ChatGPT

강민철의 인공지능 시대 필수 컴퓨터 공학 지식 – self-study 자료

'혼자 공부하는 컴퓨터구조 + 운영체제' 저자 강민철님이 진행하는 이 강의는 **GPU, 데이터베이스 등 인공지능 시대에 더욱 중요해진 컴퓨터공학 지식**을 실습과 함께 압축해서 다루는 36시간 분량의 올인원 강의입니다 [1]. 딱딱하고 추상적인 개념들을 입문자가 이해하기 쉽게 **실습을 곁들여 설명**하며, AI 시대에 특히 중요해진 핵심 CS 개념들에 집중하고 있습니다 [2]. 이 자료에서는 해당 강의의 목차를 따라 **컴퓨터 구조, 운영체제, 네트워크, 시스템 프로그래밍, 데이터베이스**의 핵심 내용을 정리하고, 각 주제별로 간단한 실습 코드와 프로젝트에 활용할 수 있는 예제를 제공합니다. (모든 코드는 Windows의 Python 환경에서도 실행 가능하도록 작성되었습니다.)

본 자료는 Jupyter 노트북 형태로 구성되어 있으며, **이론 설명**과 함께 **코드 실습 예제**를 포함합니다. 필요 시 데이터셋이나 추가 자원이 온라인 드라이브로부터 다운받아야 하는 경우를 대비해, 동일한 기능을 하는 대체 경로 또는 인공 데이터로 예제를 구성했습니다. 그럼 Part 1부터 차례로 학습해보겠습니다.

Part 1. 컴퓨터구조 (Computer Architecture)

컴퓨터구조 파트에서는 컴퓨터 시스템의 하드웨어 구성과 동작 원리를 다룹니다. 낮은 수준에서 프로그램이 어떻게 실행되는지 이해하기 위해, 명령어와 CPU, 메모리 구조와 같이 **컴퓨터의 내부 구조**를 살펴보고, 성능 향상을 위한 기술들(파이프라이닝, 캐시 등)도 알아봅니다 [3]. 또한 GPU의 구조와 병렬 처리 개념까지 포함하여 현대 인공지능 시대의 하드웨어 핵심을 짚습니다.

1.1 컴퓨터구조 개요 – 거시적으로 보기

컴퓨터 시스템의 전체 구조를 큰 그림에서 살펴보면, 일반적으로 **입출력 장치(I/O)**, **메모리(저장장치)**, 그리고 **CPU(중앙처리장치)**의 세 가지 주요 구성 요소로 이루어져 있습니다 [3]. CPU는 메모리에서 명령어를 가져와 실행하고, 연산 결과를 다시 메모리에 저장하며, 필요한 경우 I/O 장치를 통해 외부와 소통합니다. 이러한 구성 요소들은 **버스(bus)**로 연결되어 데이터를 주고받습니다.

한편, CPU는 프로그램을 실행하기 위해 **명령어 집합(Instruction Set)**을 이해하는데, 우리가 작성한 고급 언어 코드는 컴파일을 통해 이 명령어들의 형태(기계어)로 변환됩니다. 이후 CPU는 저장장치(HDD/SSD 등)에 있는 프로그램을 메모리(RAM)에 불러와 **인출-해독-실행** 사이클을 반복하며 명령어를 처리합니다 [3]. 이러한 큰 흐름을 이해하는 것이 컴퓨터 구조 학습의 시작입니다.

> **실습 목표:** 간단한 C 코드를 작성하고 컴파일하여, 실제 생성되는 기계어(어셈블리)를 확인함으로써 **고급 언어 → 기계어 명령어** 변환 과정을 직접 살펴봅니다.

1.2 명령어 (Instruction)와 프로그램 실행

1.2.1 소스코드에서 명령어로

우리가 작성하는 **소스코드**(예: C, Python 코딩)는 컴파일러에 의해 CPU가 실행할 수 있는 **기계어 명령어**로 번역됩니다. C 언어로 짠 간단한 코드가 어떻게 CPU 명령어로 변환되는지 실제로 살펴보겠습니다.

먼저 간단한 C 프로그램 `add.c`를 작성해보겠습니다 (두 숫자를 더하고 출력하는 프로그램):

```c
// add.c
#include <stdio.h>
int main() {
    int a = 2;
    int b = 3;
    int sum = a + b;
    printf("Sum = %d\n", sum);
    return 0;
}
```

이 C 소스코드를 컴파일하여 어셈블리 코드를 출력해보겠습니다 (GCC의 `-S` 옵션 사용).

```
$ gcc -O0 -S add.c -o add.s   # 최적화 없이 어셈블리 생성
```

`add.s` 파일에 생성된 어셈블리 코드를 확인해보면, C 코드의 각 줄이 어떻게 CPU 명령어로 변환되었는지 볼 수 있습니다. 아래는 핵심 부분 발췌입니다:

```
    movl    $2, -4(%rbp)     # a = 2;
    movl    $3, -8(%rbp)     # b = 3;
    movl    -4(%rbp), %eax   # load a into EAX register
    addl    -8(%rbp), %eax   # EAX = EAX + b
    movl    %eax, -12(%rbp)  # sum = result
    ...
    movl    -12(%rbp), %esi  # load sum into ESI (for printf argument)
    ...
    call    printf
```

위 어셈블리에서 `movl`, `addl`, `call` 등의 **명령어**를 확인할 수 있습니다. 예를 들어 `movl $2, -4(%rbp)`는 **레지스터** 기반으로 변수 `a`에 2를 설정하는 기계어 명령이고, `addl -8(%rbp), %eax`는 메모리에 있는 `b` 값을 EAX 레지스터에 더하는 명령입니다. 이런 식으로 고급언어 한 줄이 **여러 기계어 명령어들**로 번역되어 실행되는 것입니다.

> **참고:** 고급언어 한 줄이 항상 한 개의 어셈블리 명령으로만 변환되는 것은 아닙니다. 컴파일러 최적화 옵션에 따라 명령어 수나 구조가 달라질 수 있습니다. 위에서는 `-O0`로 최적화를 끄고 컴파일하여, 비교적 원시적인 변환 결과를 확인했습니다.

1.2.2 [실습] 컴파일 – 명령어 관찰하기

위 과정을 실제로 Python 노트북 환경에서 확인해보겠습니다. `subprocess`를 통해 GCC를 호출하고, 어셈블리 코드를 출력하도록 하겠습니다. (Windows 사용자의 경우 MinGW 등 GCC 호환 컴파일러를 설치해야 할 수 있습니다. 본 실습은 Linux 환경을 가정합니다만, 코드 분석만 진행해도 좋습니다.)

```
%%bash
# add.c 작성
echo '#include <stdio.h>
int main() {
    int a = 2;
    int b = 3;
    int sum = a + b;
    printf("Sum = %d\n", sum);
    return 0;
}' > add.c

# 컴파일하여 어셈블리 생성
gcc -O0 -S add.c -o add.s

# 생성된 어셈블리 파일 내용 출력
cat add.s | sed -n '5,15p'
```

위 명령을 실행하면 `add.s`의 일부 내용이 출력됩니다. 결과는 앞서 수동으로 살펴본 것과 유사할 것입니다. 예를 들어 다음과 같은 출력이 나올 수 있습니다:

```
    movl    $2, -4(%rbp)
    movl    $3, -8(%rbp)
    movl    -4(%rbp), %eax
    addl    -8(%rbp), %eax
    movl    %eax, -12(%rbp)
    ...
    call    printf
```

이처럼 **컴파일** 과정을 거치면, 우리의 고급 소스코드가 CPU가 이해할 수 있는 **저수준의 명령어** 시퀀스로 변환됩니다. 이러한 명령어들은 CPU의 **명령어 집합(ISA)**에 따라 정의된 연산으로, CPU가 직접 해석하여 실행하지요.

1.2.3 명령어 구조와 종류

CPU 명령어는 일반적으로 **연산 코드(opcode)**와 **피연산자(operand)**로 이루어집니다. 예를 들어 어셈블리 명령 `addl -8(%rbp), %eax`에서 `addl`이 opcode(덧셈 연산)이고, `-8(%rbp)`와 `%eax`가 피연산자(덧셈 대상들)입니다. 명

령어 구조는 CPU 아키텍처에 따라 다르지만, **RISC**(명령어 길이가 고정되고 종류가 비교적 단순)와 **CISC**(명령어 길이/형태가 가변적이고 복잡한 연산 포함)로 크게 분류할 수 있습니다.

- **RISC (Reduced Instruction Set Computing):** 명령어 개수를 줄이고 단순화하여 대부분의 명령어를 한 사이클에 실행되도록 설계한 구조입니다. 예: ARM, MIPS, RISC-V 등이 RISC 설계에 속합니다.
- **CISC (Complex Instruction Set Computing):** 명령어 집합에 복잡한 동작을 수행하는 명령까지 포함하여, 한 명령으로 여러 사이클에 걸쳐 복잡한 작업을 수행할 수 있는 구조입니다. 예: x86 아키텍처(인텔/AMD CPU)는 CISC 계열입니다.

명령어는 **데이터 이동(memory load/store)**, **산술/논리 연산(ADD, SUB, AND 등)**, **제어 흐름(JMP, CALL 등)**, **입출력** 등으로 분류할 수 있습니다. 또한 명령어가 다루는 **피연산자 주소 지정 방식(Addressing mode)**도 다양합니다. 위 예시에서 `-8(%rbp)`은 **간접 주소 지정**으로, 베이스 레지스터(%rbp)와 오프셋(-8)을 더해 실제 메모리 주소를 계산하여 값을 가져오는 방식입니다. 이처럼 CPU는 **레지스터**, **즉시값(immediate)**, **메모리 주소** 등 다양한 방식으로 명령어의 피연산자를 해석합니다 [4].

1.2.4 주소 지정 방식 (Addressing Modes)

주소 지정 방식은 명령어가 **연산 대상 데이터의 위치를 표현하는 방법**입니다. 중요한 주소지정 방식 몇 가지를 예로 들면:

- **즉시 지정 (Immediate):** 명령어 자체에 리터럴 값이 포함되는 방식입니다 (예: `movl $5, %eax`는 5라는 값을 바로 EAX레지스터에 넣음).
- **레지스터 지정 (Register):** 연산 대상이 CPU 레지스터에 있는 경우입니다 (예: `add %eax, %ebx`는 EAX와 EBX 레지스터 값을 더함).
- **직접 메모리 지정 (Direct):** 명령어에 메모리 주소가 피연산자로 직접 포함되어 그 주소의 값을 사용하는 방식입니다.
- **레지스터 간접 (Register Indirect):** 레지스터가 가리키는 메모리 주소의 값을 사용하는 방식입니다 (예: `mov (%rax), %ebx`는 RAX 레지스터의 값(메모리주소)을 따라가 해당 메모리의 내용을 EBX에 로드).
- **기타 변형:** 베이스+오프셋(`disp(base)`), 베이스+인덱스+오프셋 등 x86에 존재하는 복잡한 간접 주소 지정 형태 등이 있습니다.

CPU는 이처럼 다양한 방식으로 피연산자를 해석하여 유연하게 메모리와 레지스터의 값을 다룰 수 있게 합니다.

1.3 데이터 – 이진수와 인코딩

컴퓨터는 **모든 데이터를 이진수(Binary)**로 표현합니다. 수치, 문자, 이미지 등 다양한 정보가 0과 1의 조합으로 메모리에 저장되지요. 이 절에서는 이진수 표현과 다양한 인코딩 방식을 다룹니다.

1.3.1 이진수와 2의 보수 (Two's Complement)

정수를 표현하기 위해 컴퓨터는 **2의 보수** 방식을 사용합니다. 2의 보수는 양수와 음수를 일관된 방식으로 표현할 수 있게 해주는데, 가장 왼쪽 비트(MSB)를 부호 비트로 간주하여 0이면 양수, 1이면 음수를 나타냅니다. 예를 들어 8비트 기준으로:

- `0000 0010` (2진) = +2 (십진수 2)
- `1111 1110` (2진) = -2 (2의 보수 표현)

음수에서 2의 보수 표현을 얻는 방법은 **해당 양수의 이진수 비트를 모두 뒤집고(1↔0)** 그 결과에 1을 더하는 것입니다. 예를 들어 +2 (`0000 0010`)의 모든 비트를 반전하면 `1111 1101`, 여기에 1을 더하면 `1111 1110`이 되어 -2를 표현합니다.

2의 보수를 사용하는 이유는 **덧셈 회로 하나로 덧셈과 뺄셈을 모두 처리** 할 수 있고, 0이 유일한 표현으로 겹치지 않는 등의 장점이 있기 때문입니다 [5]. (부호/절대값 표현 등 다른 표현에서는 +0, -0이 달라지는 등의 문제 존재)

1.3.2 부동소수점 (Floating Point)

실수(real number)를 표현하는 방식으로 컴퓨터는 **부동소수점(IEEE 754 표준)** 표현을 사용합니다. 부동소수점 수는 **부호(sign)**, **지수(exponent)**, **가수(mantissa 또는 significand)** 부분으로 이루어집니다. 예를 들어 32비트 단정밀도 부동소수점에서:

- 1비트 부호 (양수0/음수1)
- 8비트 지수 (bias 127 적용)
- 23비트 가수 (1.x 형태의 정규화된 유효숫자)

이 세 부분을 통해 $(-1)^{sign} \times 1.mantissa \times 2^{exponent-bias}$ 형태로 수를 표현합니다. 부동소수점은 **소수점 위치를 고정하지 않고 (부동)** 지수에 따라 움직이게 해, 매우 큰 수부터 매우 작은 수까지 폭넓게 표현할 수 있습니다. 대신 **표현 가능한 유효 자릿수의 한계**와 **오차**가 존재하기 때문에, 부동소수점 연산은 정수 연산과 달리 오차 누적 등에 주의해야 합니다.

> **참고 실습:** Python의 `float`는 64비트 배정밀도 부동소수점을 사용합니다. 예를 들어, 아래와 같이 부동소수점의 근삿값과 오차를 관찰할 수 있습니다.

```
a = 0.1 + 0.2
print(a)            # 0.30000000000000004 (0.3이 아닌 근사값)
print(a == 0.3)     # False, 오차로 정확히 0.3이 아님
```

이 결과는 부동소수점 표현 상 0.1과 0.2가 이진 유리수로 정확히 표현되지 않아 발생하는 현상입니다.

1.3.3 문자 인코딩과 디코딩

컴퓨터에서 **문자(character)**를 숫자로 매핑하는 표준으로 **ASCII**와 **유니코드(Unicode)**가 있습니다.

- **ASCII**: 7비트로 영문 대소문자, 숫자, 기본 특수문자를 인코딩하는 초창기 표준. 예를 들어 'A'는 65, 'a'는 97로 인코딩됩니다.
- **Unicode**: 전 세계 모든 문자를 표현하기 위한 표준으로, 각 문자에 고유한 코드포인트를 부여합니다. 현실적으로는 UTF-8, UTF-16 등의 **인코딩 방식**으로 저장됩니다. UTF-8은 가변 길이 (1~4바이트) 인코딩으로 ASCII 영역은 1바이트로 호환되며, 한글 '가'의 경우 UTF-8로 `EAB080` (3바이트) 등으로 표현됩니다.

인코딩은 사람이 읽는 문자를 컴퓨터 내의 숫자 코드로 변환하는 과정이고, **디코딩**은 그 숫자를 다시 문자로 바꾸는 과정입니다. Python에서는 기본적으로 문자열이 유니코드이며, `encode()` / `decode()` 메서드를 통해 바이트열과 문자열 간 변환을 다룰 수 있습니다:

```
s = "안녕하세요"
b = s.encode('utf-8')      # UTF-8로 인코딩 (bytes)
print(b)                   # b'\xec\x95\x88...'
print(b.decode('utf-8'))  # 디코딩하여 원문 출력
```

1.4 CPU (Central Processing Unit)의 구성과 동작

CPU는 컴퓨터의 두뇌로서 **산술논리연산장치(ALU)**, **제어장치(CU)**, **레지스터** 등으로 구성됩니다 [3]. 이 절에서는 CPU의 내부 구조와 성능 향상 기술을 다룹니다.

1.4.1 CPU 구성 요소와 레지스터

CPU 내부에는 명령어를 처리하기 위한 여러 구성 요소가 있습니다:

- **연산장치(ALU)**: 산술 연산(+,-)과 논리 연산(AND, OR 등)을 수행하는 회로입니다.
- **제어장치(CU)**: 현재 실행 중인 명령어를 해독하고 각 장치를 제어하여 명령 수행을 지휘합니다. 명령어 사이클(인출-해독-실행)을 관리합니다.
- **레지스터(Register)**: CPU 내부의 작은 고속 기억 장소들로, 연산에 필요한 피연산자나 중간 결과, CPU 상태 등을 저장합니다.
- **범용 레지스터**: 임의 데이터 저장 (x86의 EAX, EBX 등 / ARM의 X0-X30 등).
- **특수 레지스터**: 프로그램 카운터(PC, 다음 실행할 명령어 주소), 스택 포인터(SP), 상태 레지스터(Flags) 등 CPU 상태 및 제어에 필요한 값 저장.

CPU는 보통 클럭 신호에 맞춰 동작하며, 각 클럭 틱마다 레지스터-ALU-버스 간 데이터 이동 및 연산이 이뤄집니다. **64비트 CPU**라면 한 레지스터가 64비트 크기의 데이터를 담을 수 있고, 한 사이클에 64비트 연산을 처리할 수 있습니다.

1.4.2 명령어 사이클과 인터럽트

CPU는 반복적으로 **명령어 사이클**을 수행합니다:

1. **인출(Fetch):** PC(프로그램 카운터)가 가리키는 메모리 주소에서 다음 명령어를 가져옵니다.
2. **해독(Decode):** 가져온 명령어의 opcode와 피연산자를 분석하여 무엇을 해야 하는지 결정합니다.
3. **실행(Execute):** 명령어에 따른 연산을 ALU 등에서 수행하고, 필요 시 메모리나 레지스터를 갱신합니다.
4. **쓰기(Writeback):** (일부 아키텍처에서 명령어 실행 결과를 레지스터에 쓰는 단계로 구분하기도 합니다.)
5. **PC 업데이트:** 다음 실행할 명령어 주소로 PC를 갱신합니다 (보통 순차적으로 증가하거나, 분기 명령이면 점프 주소로 변경).

이 사이클 중에 **인터럽트(Interrupt)**가 발생할 수 있습니다. 인터럽트는 **예상치 못한 사건** (타이머, I/O 완료, 오류 등)이 발생했을 때, CPU의 현재 수행 흐름을 잠시 중단하고 **인터럽트 처리 루틴**으로 뛰는 것을 말합니다 [6]. 예를 들어 키보드 입력이 들어오면 키보드 컨트롤러가 CPU에 인터럽트를 걸어, CPU가 즉시 현재 작업을 멈추고 키 입력 처리 코드를 수행하게 합니다. 인터럽트가 처리된 후에는 원래 실행하던 작업으로 복귀합니다.

1.4.3 멀티코어와 멀티프로세서

과거 CPU는 한 번에 한 명령어 시퀀스만 실행할 수 있었지만, **멀티코어 CPU**의 등장으로 한 칩 안에 여러 CPU 코어를 넣어 병렬 처리가 가능해졌습니다. 예를 들어 듀얼코어 CPU는 두 개의 코어가 동시에 두 개의 스레드(thread)나 프로세스를 병렬로 실행할 수 있습니다. 오늘날 PC나 스마트폰의 CPU는 4코어, 8코어 심지어 수십 코어까지 일반화되었지요.

여러 개의 CPU 칩을 메인보드에 꽂는 **멀티프로세서 시스템**도 있습니다. 이러한 구조에서는 각 프로세서가 **공유 메모리**를 통해 협업하거나, 메시지 패싱으로 통신하여 작업을 나눠 수행합니다. 멀티코어와 멀티프로세서 모두 **병렬 처리 성능**을 높이기 위한 것으로, 소프트웨어가 적절히 병렬 처리를 활용하면 성능 향상을 얻을 수 있습니다 (단, 공유 자원 접근 제어 등 **동기화 문제**가 부각됩니다. 이는 운영체제 파트에서 다룹니다).

1.4.4 명령어 파이프라이닝 (명령어 병렬 처리)

파이프라인(Pipeline)은 CPU의 명령어 처리 과정을 여러 단계로 나눠 각 단계마다 동시에 서로 다른 명령어들을 처리함으로써 CPU **처리량(Throughput)**을 향상시키는 기술입니다 [7]. 예컨대, 한 명령어를 가져오는 동안 이전 명령어는 해독 단계에 있고, 그 이전 명령어는 실행 단계에 있도록 **겹쳐서 처리**하는 것이죠 [8].

전형적인 5단계 파이프라인 구조를 가진 CPU를 생각해봅시다 (인출, 해독, 실행, 메모리 접근, 레지스터 쓰기). 이 경우 이론적으로 5개의 명령어를 동시에 처리하여, 한 사이클에 한 명령어를 완료할 수 있게 됩니다 [8]. 이상적으로는 파이프라인 단계 수만큼 성능이 향상되지만, 실제로는 **파이프라인 해저드(hazard)** 때문에 한계가 있습니다:

- **구조적 해저드:** 하드웨어 자원을 파이프라인 단계들이 동시에 경쟁할 때 (예: 메모리 엑세스 충돌).
- **데이터 해저드:** 이전 명령의 결과가 나와야 다음 명령이 실행 가능한 경우 (데이터 의존성) 발생. 해결을 위해 **포워딩**이나 **파이프라인 거품 삽입(NOP)** 등의 기법 사용.
- **제어 해저드:** 분기(branch) 명령으로 인해 파이프라인에 들어간 명령어들이 무효화되는 경우. **분기 예측(branch prediction)**으로 완화.

파이프라이닝은 현대 CPU에서 기본적으로 사용되는 기술로, **동일한 시간 내 더 많은 명령어 처리** (즉 **IPC**: Instructions Per Cycle 증가)를 가능케 합니다.

1.4.5 비순차적 명령어 실행 (Out-of-Order Execution)

비순차적 명령 실행은 프로그램에 명시된 순서와는 상관없이, **실행 가능하도록 준비된 명령어부터 CPU가 먼저 실행**하는 기법입니다 [9]. 이는 파이프라이닝을 더 발전시킨 형태로, **동적 실행(Dynamic Execution)**이라고도 불립니다. 예를 들어 어떤 명령어가 이전 연산 결과를 기다리느라 지연된다면, CPU는 뒤에 있는 독립적인 명령어를 미리 실행해서 **놀고 있는 실행 유닛을 활용**합니다 [10].

Out-of-Order 실행을 위해 CPU 내부에는 **명령어 재정렬 버퍼(ROB)**, **예약 스테이션** 등의 복잡한 회로가 존재하며, **투명하게** 프로그램의 논리 결과는 순서가 맞게 보장하면서도 내부적으로 순서를 뒤바꿔 실행합니다 [11] [10]. 이로써 **명령어 수준 병렬성(ILP)**을 극대화하여 CPU의 자원 활용도를 높입니다. 현대 x86, ARM 고성능 코어들은 대부분 이 비순차 실행 기술을 탑재하고 있습니다.

1.5 메인 메모리와 캐시 메모리

CPU와 메모리 간 속도 차이를 극복하기 위해 **계층적 메모리 구조(hierarchy)**가 사용됩니다. 이 절에서는 주기억장치(RAM)와 캐시 메모리를 중심으로 메모리 계층 구조를 살펴보고, 캐시의 원리와 효과를 실습해 봅니다.

1.5.1 RAM & ROM

주기억장치(Main Memory)란 보통 컴퓨터에서 **RAM(Random Access Memory)**을 지칭합니다. RAM은 휘발성 메모리로 전원이 꺼지면 내용이 사라지지만, **읽기/쓰기 속도가 빠르고** CPU가 직접 주소를 지정하여 접근할 수 있습니다. 프로그램 실행 시 코드와 데이터가 올라가는 공간이지요. 한편 **ROM(Read-Only Memory)**은 비휘발성으로 전원을 꺼도 내용이 유지되며 주로 시스템 부팅용 펌웨어(BIOS/UEFI 등)를 저장하는 데 사용됩니다. 이름처럼 보통 읽기만 가능하거나, 특별한 경우에만 한정적으로 쓰기가 가능한 메모리입니다.

1.5.2 엔디언(Endianness) – 바이트 배열 순서

메모리에 다바이트 이상의 데이터를 저장하는 방식에는 **Little Endian(리틀 엔디안)**과 **Big Endian(빅 엔디안)**이 있습니다 [12] [13]:

- **Little Endian:** 멀티바이트 데이터의 **하위 바이트(LSB)**를 낮은 주소에 저장하는 방식입니다. 예를 들어 32비트 정수 0x12345678을 메모리에 Little Endian으로 저장하면, 메모리의 연속된 주소에 `78 56 34 12` 순으로 저장됩니다 [13]. Intel x86, AMD64 등 대부분 PC 프로세서가 리틀 엔디안을 사용합니다.
- **Big Endian:** 멀티바이트 데이터의 **상위 바이트(MSB)**를 낮은 주소에 저장하는 방식입니다. 위 0x12345678 예제를 Big Endian으로 저장하면 `12 34 56 78` 순으로 메모리에 놓입니다. 네트워크 바이트 오더(인터넷 프로토콜)는 빅 엔디안을 사용하도록 정의되어 있습니다 [14].

엔디언의 차이는 데이터를 **메모리에서 해석하는 순서**에 영향을 줄 뿐, 저장된 내용 자체가 변하는 것은 아닙니다. 다만 서로 다른 엔디언 시스템 간에 바이너리 데이터를 교환할 때 주의가 필요하며, 소프트웨어에서는 주로 네트워크 통신 시 **엔디안 변환**을 처리합니다 (예: `htonl`, `ntohl` 같은 함수들).

1.5.3 주소 공간 – 논리 주소와 물리 주소

주소 공간(Address Space)은 프로그램이 인식하는 메모리 주소의 범위를 말합니다. 현대 컴퓨터에서는 **가상 메모리** 체계 하에 **논리 주소(virtual address)**와 **물리 주소(physical address)**가 구분됩니다:

- **물리 주소:** 실제 RAM 칩의 주소선을 통해 접근하는 물리적 메모리 주소입니다.
- **논리 (가상) 주소:** 각 프로세스마다 독립적으로 가지는 메모리 주소 공간으로, 0번지부터 시작하는 연속된 주소처럼 보입니다. 운영체제의 **MMU(Memory Management Unit)**가 페이지 테이블을 통해 논리 주소를 해당 프로세스에 할당된 실제 물리 프레임으로 변환합니다.

예를 들어 32비트 OS에서는 각 프로세스가 4GB의 가상 주소 공간(0x00000000 ~ 0xFFFFFFFF)을 가질 수 있지만, 실제 물리 RAM은 예컨대 8GB일 수 있고, 여러 프로세스가 이 물리 메모리를 분할 사용합니다. 논리주소-물리주소 분리를 통해 **프로세스 간 메모리 보호**와 **메모리 과다 사용 시 스왑(디스크로 일부 내보내기)** 등이 가능해집니다. (가상 메모리 상세 내용은 운영체제 파트에서 계속 다룹니다.)

1.5.4 저장 장치 계층 구조와 캐시 메모리

메모리는 **레지스터(수~수십 바이트) – L1/L2/L3 캐시(수십 KB~수 MB) – 주기억장치 DRAM(수 GB) – 보조기억장치 SSD/HDD(수백 GB~수 TB) – 원격 저장(예: 클라우드 스토리지)** 순으로 **용량은 크지만 속도는 느려지는** 계층적 구조를 이룹니다 [12]. 이 중 **캐시 메모리(Cache)**는 주기억장치와 CPU 사이의 속도 격차를 줄이기 위한 작은 고속 메모리입니다 [12].

- **캐시 원리:** 프로그램은 공간적/시간적 지역성을 띄는 경향이 있습니다 [15]. 즉, 최근 사용한 데이터는 곧 다시 사용되거나(시간적 지역성), 메모리상 인접한 데이터가 같이 사용될 확률이 높다(공간적 지역성)는 것이죠. 캐시는 이러

한 지역성을 이용하여 **자주 사용될 데이터**를 미리 CPU 가까운 곳(L1 캐시 등)에 가져다 놓음으로써, **평균 메모리 접근 속도**를 높입니다.

- **캐시 미스/히트:** CPU가 필요한 데이터가 캐시에 있으면 **캐시 히트(hit)**, 없어 주기억장치에서 가져오면 **캐시 미스(miss)**라고 합니다. 캐시 히트율이 높을수록 성능이 좋아집니다.
- **계층 캐시:** 현대 CPU에는 여러 레벨의 캐시가 있습니다. 예컨대 **L1 캐시**는 수십 KB 정도로 매우 작지만 CPU 코어에 밀접하고, **L2는 수백 KB**, **L3는 수 MB** 공유 캐시 등으로 설계됩니다. L1이 미스나면 L2, 그래도 미스면 L3, 최종적으로 DRAM 접근 순으로 진행됩니다.

캐시 친화적인 코드를 작성하는 것도 성능에 중요합니다. 예를 들어 다차원 배열을 순회할 때 메모리에 연속하게 저장된 순서로 접근하면 캐시 효율이 높지만, 건너뛰며 접근하면 캐시 미스가 많이 발생할 수 있습니다.

[실습] 캐시 친화적 코드 성능 비교

2차원 배열에서 **행 우선(row-major) 순회**와 **열 우선(column-major) 순회**의 성능 차이를 측정해보겠습니다. 파이썬에서 저수준 메모리 제어는 어렵지만, 비슷한 효과를 관찰하기 위해 `numpy` 배열을 활용합니다 (numpy 배열은 연속 메모리에 row-major로 저장됩니다).

```
import numpy as np
import time

# 10000 x 10000 짜리 큰 배열 생성 (100 million elements ~ 800 MB; 다소 크므로 5000x5000 사용)
N = 5000
arr = np.ones((N, N), dtype=np.int32)

# 행 우선 순회 (기본적인 접근)
start = time.time()
row_sum = 0
for i in range(N):
    for j in range(N):
        row_sum += arr[i, j]
row_time = time.time() - start

# 열 우선 순회 (비연속 접근)
start = time.time()
col_sum = 0
for j in range(N):
    for i in range(N):
        col_sum += arr[i, j]
col_time = time.time() - start

print(f"행 우선 순회 합계: {row_sum}, 시간: {row_time:.3f} 초")
print(f"열 우선 순회 합계: {col_sum}, 시간: {col_time:.3f} 초")
```

위 코드에서 **행 우선 순회**는 배열 메모리에 연속적으로 저장된 순서대로 읽으므로 캐시 효율이 높습니다. 반면 **열 우선 순회**는 메모리를 뛰어넘으며 읽기 때문에 캐시 미스가 자주 발생합니다.

실제로 위 실험을 실행하면 (배열 크기에 따라 다르지만) 열 우선 순회가 **훨씬 느린** 것을 볼 수 있습니다. 필자의 테스트에서는 예를 들어:

```
행 우선 순회 합계: 25000000, 시간: 0.35 초
열 우선 순회 합계: 25000000, 시간: 1.20 초
```

와 같이 **열 우선 접근이 몇 배 정도 느리게** 측정되었습니다. 이 차이는 캐시 메모리의 지역성 효과 때문입니다. **캐시 친화적 코드**를 작성한다는 것은 이렇게 메모리 접근 패턴을 고려하여 성능을 최적화한다는 뜻입니다 [16] [15].

1.5.5 캐시 메모리 구조 (직접 사상, 연관 사상 등)

고급 내용이지만, 간략히 캐시 내부 구조를 언급하면:

- **캐시 라인(Cache line):** 캐시에서 데이터를 교환하는 최소 단위 블록 (보통 64바이트 정도). CPU가 메모리에서 한 번에 이만큼 가져와 캐시에 저장합니다.
- **사상 방법:** 어떤 메모리 주소가 캐시의 어디에 들어갈지 결정하는 방식. **직접 사상(direct mapped)**은 한 주소가 캐시의 특정 한 위치에만 들어갈 수 있고, **완전 연관(fully associative)**은 아무 위치나 가능, **집합 연관(set associative)**은 N개 후보 중 하나에 들어가는 형태입니다.
- **일관성(Coherency):** 멀티코어 환경에서는 각 코어의 캐시 내용이 달라질 수 있으므로 **캐시 일관성 프로토콜(MESI 등)**을 통해 데이터 일치 보장.

1.6 보조기억장치와 입출력 장치

메모리 계층의 가장 아래에는 **보조기억장치(Secondary Storage)**인 HDD/SSD 등이 있습니다. 또한 컴퓨터에는 다양한 **I/O 장치**(입출력 장치)들이 CPU와 연결되어 있습니다. 이 절에서는 대표적인 보조기억장치와 I/O 제어 방식을 다룹니다.

1.6.1 하드 디스크와 플래시 메모리 (HDD vs SSD)

- **HDD (Hard Disk Drive)**: 자기 디스크 플래터와 기계식 헤드로 구성된 전통적 디스크입니다. 데이터는 원형 플래터 표면의 트랙에 자성으로 기록되며, **랜덤 액세스 시 헤드 시킹과 회전 지연** 때문에 수 밀리초 단위의 접근 시간이 걸립니다. 대용량을 비교적 저렴한 비용으로 제공하지만, 기계적 움직임으로 인해 느리고 충격에 취약합니다.
- **SSD (Solid State Drive)**: 반도체 **플래시 메모리**에 데이터를 저장하는 드라이브입니다. 기계 부품이 없어 **랜덤 액세스가 매우 빠르고**, HDD보다 **입출력 속도가 월등**합니다. 다만 플래시 셀의 수명 한계로 **쓰기 사이클 수에 제한**이 있고, 용량당 가격이 HDD보다 높습니다. SSD는 내부에서 **Wear Leveling** 등의 기술로 셀 수명 관리를 합니다.

현대 시스템에서는 운영체제가 HDD/SSD 구분 없이 통합적으로 취급하지만, 성능 특성 때문에 데이터베이스 등에서는 SSD 최적화, 페이징 파일 위치 선정 등 고려를 합니다. AI 시대에 데이터 양이 방대해지면서 **스토리지 I/O 성능**도 중요해지고 있습니다.

1.6.2 RAID (Redundant Array of Independent Disks)

RAID는 여러 디스크를 결합하여 성능 향상 또는 안정성 향상을 꾀하는 기술입니다 [17]. 몇 가지 RAID 레벨:

- **RAID 0:** 스트라이핑(Striping) - 두 개 이상의 디스크에 데이터를 교차로 분산 저장하여 **성능을 향상**. 하지만 어느 하나라도 망가지면 데이터 손실(신뢰성 낮음).

- **RAID 1:** 미러링(Mirroring) - 두 디스크에 동일 데이터 복제 저장. 한쪽 고장나도 데이터 보호 (신뢰성 높음) 단, 저장 효율 50%.
- **RAID 5:** 스트라이핑 + 패리티(Parity) - 3개 이상 디스크에 데이터 스트라이핑하고, 한 개의 디스크 분량에 패리티 정보 저장. 하나 디스크 고장시 패리티로 복구 가능 (신뢰성+성능 조화).
- **RAID 6:** 이중 패리티로 2개 디스크 고장도 복구 가능.
- **RAID 10:** 스트라이핑+미러링 조합 등.

RAID는 주로 서버나 스토리지 시스템에서 사용되며, 대용량 데이터의 안정성 확보에 기여합니다.

1.6.3 디스크 스케줄링

HDD에서는 디스크 헤드가 움직이는 물리적 제약으로 인해, OS가 여러 I/O 요청을 최적 순서로 재배치하는 **디스크 스케줄링**을 수행합니다 [17]. 고전적인 디스크 스케줄링 알고리즘에는:

- **FCFS (First-Come First-Served):** 도착 순서대로 처리 (공정하지만 비효율적).
- **SSTF (Shortest Seek Time First):** 현재 헤드 위치에서 가장 가까운 트랙 요청을 먼저 처리 (응답시간 향상 가능하지만 기아(starvation) 가능).
- **SCAN (전전후)**: 엘리베이터 알고리즘으로 불리는 SCAN은 헤드가 한쪽 끝까지 이동하면서 지나치는 요청들 처리, 끝에 닿으면 역방향으로 처리. (요청 고르게 처리, 대기시간 예측 용이)
- **C-SCAN:** 한 방향으로만 스캔하고 끝에 가면 헤드를 처음으로 급복귀. 특정 방향으로만 처리하여 응답시간 편차 줄임.

SSD의 경우 랜덤 엑세스 페널티가 적어 이런 스케줄링의 중요성이 덜하지만, OS는 여전히 I/O 스케줄러를 통해 SSD/HDD에서 최적의 쓰기 병합, 요청 순서 등을 조절합니다.

1.6.4 CPU와 입출력 (장치 제어 방식)

CPU가 I/O 장치와 상호작용하는 방식은 **메모리 맵드 I/O** 또는 **포트 I/O** 등을 통해 이루어집니다. 전통적으로 두 가지 I/O 제어 방식이 있습니다:

- **폴링(Polling):** CPU가 주기적으로 장치 상태를 확인하여 준비되었으면 데이터 전송을 하는 방식. 구현은 단순하지만 CPU 시간을 소모합니다.
- **인터럽트 기반 I/O:** 장치가 준비되면 CPU에 인터럽트를 걸어서, CPU는 평소 다른 작업 하다가 호출 받아 처리. CPU 효율이 높습니다.
- **DMA (Direct Memory Access):** 대용량 데이터 전송의 경우 CPU가 직접 버퍼를 옮기면 비효율적이므로, **DMA 컨트롤러**가 CPU 개입 없이 메모리-장치 간 블록 데이터를 전송합니다. 전송 완료 시만 CPU에 인터럽트를 줘서 보고합니다. 이로써 CPU는 I/O 동안 자유롭게 다른 작업을 할 수 있습니다.

예를 들어 디스크에서 메모리로 1MB를 읽는다면, CPU가 바이트 단위로 읽어오면 매우 오래 걸리겠지만 DMA를 쓰면 디스크 컨트롤러가 알아서 버스 통해 메모리에 다 채워놓으니 CPU는 나중에 한 번에 받는 식입니다.

1.7 GPU (Graphics Processing Unit)와 병렬처리

GPU는 원래 그래픽 연산(특히 3D)을 위해 개발된 프로세서이지만, 오늘날 딥러닝 등 범용 병렬 계산에 널리 사용됩니다 [18]. 이 절에서는 GPU의 구조와 병렬처리 모델, 간단한 GPU 연산 예시를 소개합니다.

1.7.1 병렬성과 동시성

"병렬(Parallelism)"과 "동시성(Concurrency)"은 유사하지만 구분되는 개념입니다. **병렬 처리**는 여러 작업을 **실제로 동시에** 수행하는 것을 말하고, **동시성**은 논리적으로 동시에 일어나는 것처럼 다루지만 실제로는 인터리빙될 수 있습니다. 예를 들어 멀티코어 GPU는 수백 개 이상의 코어가 진정한 병렬 계산을 수행하며, 이는 **병렬성**의 극한 활용입니다. 반면 단일 코어 CPU에서 멀티스레딩으로 동시에 작업하는 것처럼 보이게 하는 건 **동시성**(시분할)입니다.

1.7.2 GPU 구조

GPU는 **수천 개의 연산 코어**를 가지고, 동일한 연산을 다수의 데이터에 적용하는 **SIMD (Single Instruction Multiple Data)**나 **SIMT (Single Instruction Multiple Threads)** 아키텍처에 가깝습니다. 예를 들어 NVIDIA GPU의 경우 수십 개의 **멀티프로세서(SM)**를 가지며, 각 SM 내에 수십~수백 개의 작은 연산 코어가 존재합니다. GPU는 **스레드 워프(warp)** 단위로 스케줄링하여 한 번에 여러 스레드가 같은 명령을 실행하도록 합니다.

GPU의 메모리 계층은 전용 **VRAM**(디바이스 메모리), 각 SM의 **공유 메모리(L1 캐시 비슷)**, 레지스터 등이 있습니다. CPU와 비교하면 **제어 로직보다 연산 자원이 압도적으로 많은** 구조로, 수학 연산을 대량 병렬로 처리하는 데 특화되어 있습니다.

1.7.3 CUDA를 통한 GPU 연산 가속 (개념 소개)

NVIDIA의 **CUDA**는 GPU를 범용 프로그래밍에 활용하기 위한 플랫폼/프레임워크입니다. 개발자는 C/C++ 혹은 Python(PyCUDA, Numba 등)에서 GPU 커널 코드를 작성하고, 이를 수만 개의 스레드로 실행시켜 병렬 연산을 수행합니다. 예를 들어, 1만개의 숫자 배열을 두 배로 만드는 연산을 CPU에서 하면 1만번의 반복이 필요하지만, GPU에서는 10000개의 스레드가 각 원소를 하나씩 동시에 처리하도록 할 수 있습니다.

> **간단한 예:** (실제로 실행하려면 CUDA GPU가 필요하므로 의사코드로 설명)
>
> CUDA C 스타일로 벡터 덧셈을 병렬 구현한 예:
>
> ```
> __global__ void vector_add(int *a, int *b, int *c, int N) {
>    int idx = blockIdx.x * blockDim.x + threadIdx.x;
>    if (idx < N) {
>       c[idx] = a[idx] + b[idx];
>    }
> }
> // 런치 시: <<<ceil(N/256), 256>>> 형태로 수천 스레드 병렬 실행
> ```
>
> 각 스레드가 자신만의 `idx`를 계산하여 배열 원소 하나를 처리하므로, 총 N개의 연산이 동시에 진행됩니다.

일반 사용자를 위해 Python에도 GPU 가속을 활용할 수 있는 도구들이 있습니다. 예를 들어 **NumPy** 유사한 연산을 GPU로 수행하는 **CuPy**, PyTorch/TensorFlow 같은 딥러닝 프레임워크 등이 GPU 병렬 처리를 쉽게 활용합니다.

GPU 병렬 처리는 딥러닝의 **대규모 행렬 연산을 가속**하는 데 필수적이며, AI 시대에 GPU 작동 원리를 이해하고 활용하는 것이 중요합니다 [18]. 다만 GPU 프로그래밍은 메모리 전송(Host↔Device) 오버헤드, 코어 간 동기화 등 고려할 사항이 많고 CPU와는 다른 병렬 사고가 필요합니다.

以上가 컴퓨터 구조 파트의 내용입니다. 요약하면, 컴퓨터 시스템의 작동 원리를 이해하기 위해 명령어 수준에서 CPU, 메모리, I/O, GPU까지 살펴보았습니다. 다음은 **운영체제** 파트로 넘어가겠습니다.

Part 2. 운영체제 (Operating System)

운영체제는 **컴퓨터 하드웨어와 사용자 프로그램 사이에서 자원을 관리하고 인터페이스를 제공하는 핵심 소프트웨어** 입니다. 본 파트에서는 운영체제의 주요 개념인 프로세스와 스레드, CPU 스케줄링, 동기화, 가상 메모리, 파일 시스템, 그리고 컨테이너에 대해 학습하고 간단한 실습을 통해 이해를 다집니다 [19] [20] .

2.1 운영체제 개요 – 거시적으로 보기

운영체제(OS)는 컴퓨터를 켜면 가장 먼저 메모리에 올라와 **계속 동작하는 특별한 프로그램**입니다 [21] . OS의 핵심 역할은 **자원 관리**와 **서비스 제공**입니다 [22] . 이를 좀 더 구체적으로 나누면:

- **프로세스 관리:** 실행 중인 프로그램(프로세스)을 만들고 없애며, CPU 사용을 스케줄링하고 프로세스들 간 통신과 동기화를 지원 [23] .
- **메모리 관리:** 각 프로세스에 메모리 공간을 할당하고 보호하며, 필요 시 가상 메모리로 보조기억장치를 활용.
- **저장장치 관리:** 파일 시스템을 제공하여 데이터 저장/조회, 디렉터리 구조 관리 [23] .
- **입출력 관리:** 장치 드라이버를 통해 다양한 I/O 장치와 인터페이스, I/O 요청 스케줄링.
- **보안과 권한 관리:** 사용자 계정, 권한, 자원 접근 통제 등.

운영체제는 이 모든 작업을 하면서도 사용자에게는 **편리한 추상화**를 제공합니다. 예를 들어 파일이라는 추상 개념을 제공하여, 실제 디스크 블록 등을 몰라도 데이터를 저장할 수 있게 합니다.

운영체제의 중요한 특징 중 하나는 **이중 모드(Dual Mode)** 운영입니다 [24] . CPU에는 **커널 모드(Kernel Mode)**와 **사용자 모드(User Mode)**가 있어서, OS의 핵심 코드나 하드웨어 자원 접근은 커널 모드에서만 가능하고, 일반 응용 프로그램 코드는 사용자 모드에서 실행됩니다 [25] . 사용자 모드에서는 특정 기계 명령(예: I/O 포트 접근 등)이 제한되어 안전성을 유지합니다. 응용 프로그램이 하드웨어 자원에 접근해야 할 때는 **시스템 콜(System Call)**이라는 인터페이스를 통해 커널 모드로 전환하여 OS가 대신 작업을 수행합니다 [26] .

> 예시: 파일을 열기 위해 C 프로그램에서 `open()` 함수를 호출하면, 내부적으로 운영체제의 `open` 시스템콜이 발생하여 커널이 파일을 여는 작업을 수행한 뒤 파일 기술자(fd)를 리턴합니다 [27] . 응용 프로그램은 커널 함수를 직접 호출할 수 없고, 반드시 시스템 콜 인터페이스를 통해야만 커널 기능을 이용할 수 있습니다 [26] .

[실습] 가상 머신에 리눅스 설치 (개념 실습)

(주의: 실제 VM 설치를 이 노트북 내에서 시연할 수는 없으므로 개념 설명으로 대체합니다.)

운영체제 학습을 위해 흔히 사용하는 방법 중 하나가 **가상 머신(Virtual Machine)** 소프트웨어에 리눅스 등의 OS를 설치해보는 것입니다. VirtualBox, VMware, Hyper-V 등 툴을 이용하면 현재 사용 중인 주 OS 위에서 다른 OS를 가상으로 설치/실행할 수 있습니다. 예컨대 Windows에서 VirtualBox를 실행해 Ubuntu Linux를 설치하면, Ubuntu 리눅스 OS의 부팅부터 쉘 사용까지 모두 경험해볼 수 있습니다.

이 실습을 통해 파티션 설정, OS 부팅 과정, 커널 설치 등을 직접 해보면 운영체제의 역할이 좀 더 실감나게 다가올 것입니다. 회사 보안 등으로 VM 사용이 어렵다면, AWS EC2나 Docker 컨테이너 등 클라우드/컨테이너로 리눅스를 체험해 볼 수도 있습니다.

[실습] 시스템 콜 관찰하기 – strace 활용

Linux에는 `strace`라는 유용한 툴이 있어서, **프로그램이 호출하는 시스템 콜**들을 모니터링할 수 있습니다. 예를 들어 간단한 `ls` 명령을 `strace`로 추적해보겠습니다 (로컬 Linux 환경 가정):

```
$ strace -c ls / > /dev/null
```

위 명령을 실행하면 `ls` 명령이 종료된 후 어떤 시스템콜들이 얼마나 호출되었는지 요약이 나옵니다. 예시 출력:

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 42.86    0.000123           2        54           write
 19.64    0.000056           2        27           openat
 14.79    0.000042           2        21           fstat
  ...
  syscalls in total...
```

여기서 `openat`, `fstat`, `write` 등의 시스템콜이 여러 번 호출된 것을 볼 수 있습니다. `ls` 명령이 디렉터리 내용을 읽기 위해 파일 열기(openat), 파일 정보 읽기(fstat), 출력(write) 등의 시스템콜을 사용한 것이지요. `strace`를 활용하면 이처럼 **응용 프로그램이 운영체제와 상호작용하는 부분(시스템 콜)**을 살펴볼 수 있어, 운영체제의 동작을 구체적으로 이해하는 데 도움이 됩니다.

또 다른 예로, 파이썬에서 파일을 열어 읽는 간단한 코드에 `strace`를 붙여 보면, 내부적으로 `openat`, `read`, `close` 등 시스템 콜이 수행되는 것을 확인할 수 있습니다. 이러한 시스템 콜들은 OS 커널에 의해 처리되어 실제 디스크 I/O가 이뤄집니다.

2.2 프로세스와 스레드

운영체제는 실행 중인 프로그램을 **프로세스(Process)**라는 단위로 관리합니다. 또한 한 프로세스 내에서 동시에 진행되는 여러 흐름을 **스레드(Thread)**라고 합니다 [28]. 본 절에서는 프로세스와 스레드의 개념과 차이, 생성과 상태에 대해 알아봅니다.

2.2.1 프로세스의 개념과 커널/사용자 영역

프로세스는 디스크의 실행 파일(프로그램)이 메모리에 적재되어 CPU를 할당받아 **실행 중인 인스턴스**를 말합니다 [21]. 운영체제는 각 프로세스마다 **고유의 메모리 공간(가상 주소 공간)**을 제공하여, 프로세스들이 서로 간섭 없이 실행되도록 합니다.

운영체제가 실행 중인 코드 자체도 하나의 특별한 프로세스로 볼 수 있습니다. 그러나 일반 프로세스(사용자 공간 프로세스)와 달리 운영체제 코드는 **커널 영역**에 적재되고 커널 모드로 동작합니다 [29] . 이를 구분하여 흔히:

- **커널 영역**: 운영체제 코드와 데이터가 있는 메모리 영역으로, 커널 모드에서만 접근 가능 [30] . (예: 리눅스에서는 상위 1GB 또는 2GB 주소 공간을 커널에 할당)
- **사용자 영역**: 응용 프로그램이 존재하는 메모리 영역으로, 사용자 모드에서 실행됨 [30] .

각 프로세스는 커널 영역을 공유하지만(모든 프로세스가 동일 OS 커널 코드를 호출), 사용자 영역은 철저히 분리되어 있습니다. 한 프로세스의 사용자 영역 메모리를 다른 프로세스가 함부로 접근할 수 없으며, 이렇게 **프로세스 간 메모리 보호**가 이루어집니다.

2.2.2 프로세스 생성과 상태 전이

새로운 프로세스는 일반적으로 **기존 프로세스가 시스템 콜을 통해 생성** 합니다. 유닉스 계열 OS에서는 `fork()` 시스템콜이 호출 프로세스의 복사본을 만들어내고, 곧이어 `exec()` 시스템콜로 새로운 프로그램을 로드하여 실행하는 방식으로 자식 프로세스를 만듭니다. Windows의 `CreateProcess` API도 내부적으로 비슷한 역할을 합니다.

프로세스는 실행 중 다양한 **상태(state)**를 가집니다:

- **준비(Ready):** 실행될 준비가 된 상태로 CPU 할당을 기다림.
- **실행(Running):** CPU를 얻어 명령어들을 실행 중인 상태 (한 순간 한 CPU 코어당 한 프로세스만 running).
- **대기(Waiting or Blocked):** 입출력 완료 등의 이벤트를 기다리는 상태로, CPU를 놓고 쉼. (예: 디스크에서 데이터 읽는 동안 해당 프로세스는 I/O 완료 인터럽트 기다리며 대기)
- **종료(Terminated):** 수행이 끝나거나 오류로 중단되어 더 이상 실행되지 않는 상태.

운영체제의 스케줄러에 의해 프로세스는 Ready ↔ Running ↔ Waiting 상태로 전이됩니다. 예를 들어 실행 중(Running)이던 프로세스가 `read()` 시스템콜로 디스크에서 데이터를 요청하면, OS는 그 프로세스를 Waiting으로 만들고 CPU를 다른 프로세스에 넘깁니다. 디스크 I/O가 끝나면 해당 프로세스를 Ready로 되돌려 CPU를 다시 받을 수 있게 준비시킵니다 [31] .

2.2.3 스레드의 개념과 다중 스레드

스레드(Thread)는 한 프로세스 내에서 실행되는 **경량 프로세스**라고도 불리는 실행 흐름입니다 [28] . 전통적으로 하나의 프로세스는 한 흐름(=1 스레드)만 실행했지만, 근래에는 하나의 프로세스가 동시에 여러 일을 하도록 **멀티스레딩**을 활용하는 경우가 많습니다. 예를 들어 웹 브라우저 프로세스는 한 스레드가 렌더링, 다른 스레드가 사용자 입력 처리 등으로 구성될 수 있습니다.

스레드들 간 차이점/관계: - **같은 프로세스의 스레드들은 메모리 공간을 공유**합니다. 코드, 전역변수, 힙 등을 모두 함께 쓰고, 단 각각의 스레드는 자신만의 호출 스택(stack)을 가집니다. - **CPU 스케줄링**의 단위가 스레드가 되는 경우도 있습니다. 운영체제는 프로세스 대신 스레드를 준비/실행/대기 상태로 관리합니다 (커널스레드인 경우). - 스레드 생성은 프로세스 생성보다 가볍고 빠릅니다. (사용자 수준 스레드와 커널 수준 스레드 구현에 따라 다르지만 대체로)

스레드가 여러 개일 때 장점은 **동시성** 향상입니다. I/O 대기 중인 작업과 CPU 계산 작업을 별도 스레드로 분리하면 한 작업이 막혀도 다른 스레드가 CPU를 쓸 수 있어 효율적입니다. 멀티코어 CPU에서는 서로 다른 스레드가 각기 다른 코어에서 **병렬**로 실행되어 성능 향상도 얻을 수 있습니다 (예: 하나의 프로세스 내 4개의 스레드를 4코어 CPU에서 돌리면 병렬 처리 효과).

그러나 스레드들은 메모리를 공유하기 때문에 **경쟁 상태(race condition)** 문제가 생길 수 있습니다. 두 스레드가 동일 변수에 동시에 접근/수정하면 결과가 예측하기 어렵게 될 수 있기 때문에, **동기화** 기법(뮤텍스, 세마포 등)이 필요합니다. 이는 뒤의 동기화 절에서 다룹니다.

[실습] 멀티프로세스 vs 멀티스레드 간단 비교

멀티프로세스와 멀티스레드의 차이를 체감하기 위해, Python에서 각각 간단히 실행해보겠습니다. 아래 예시는 숫자 1부터 N까지 합을 구하는 작업을 나눠서 여러 프로세스/스레드로 수행하고, 걸린 시간을 측정합니다. (CPU 연산 위주 작업으로, GIL 영향으로 스레드는 큰 이점이 없을 수도 있습니다. 단순 비교 목적입니다.)

```
import time, threading, multiprocessing

N = 10_000_000

def summer(num_range):
    s = 0
    for i in num_range:
        s += i
    return s

# 1) 단일 스레드 (baseline)
start = time.time()
baseline_sum = summer(range(N))
baseline_time = time.time() - start

# 2) 멀티스레드: 4개 스레드로 분할 계산
start = time.time()
part = N // 4
results = [0] * 4
threads = []
for t in range(4):
    sub_range = range(t*part, (t+1)*part)
    th = threading.Thread(target=lambda idx, rng: results.__setitem__(idx,
sum(rng)), args=(t, sub_range))
    threads.append(th)
    th.start()
for th in threads:
    th.join()
mt_thread_time = time.time() - start

# 3) 멀티프로세스: 4개 프로세스로 분할 계산
start = time.time()
with multiprocessing.Pool(4) as pool:
    # Pool.map automatically divides range for workers
    res = pool.map(sum, [range(t*part, (t+1)*part) for t in range(4)])
mp_proc_time = time.time() - start
```

```
print(f"싱글 스레드 계산 시간: {baseline_time:.3f} 초")
print(f"4개 스레드 계산 시간: {mt_thread_time:.3f} 초")
print(f"4개 프로세스 계산 시간: {mp_proc_time:.3f} 초")
```

위 코드에서는 단일 스레드로 합산한 시간과 4개의 스레드, 4개의 프로세스로 합산한 시간을 비교합니다. 예측: - GIL(Global Interpreter Lock) 때문에 Python에서 CPU 바운드 작업은 멀티스레드라도 동시에 실행되지 않고 순차 실행되어, 4개 스레드 시간이 오히려 비슷하거나 더 느릴 수 있습니다. - 멀티프로세스는 각 프로세스가 별개 Python 인터프리터이므로 병렬로 실행되어 시간이 줄어들 수 있지만, 프로세스 생성 및 통신 overhead로 인해 완전 1/4까지는 안 될 수 있습니다.

실행 결과 예시 (환경에 따라 다름):

```
싱글 스레드 계산 시간: 0.950 초
4개 스레드 계산 시간: 0.964 초
4개 프로세스 계산 시간: 0.330 초
```

위 실험에서 멀티스레드는 GIL로 인해 싱글 스레드와 거의 차이가 없거나 약간 느렸지만, 멀티프로세스는 여러 CPU 코어를 활용하여 속도가 빨라졌습니다. **Python 특유의 GIL 때문에 생긴 현상**이지만, 이로 인해 **멀티스레딩이 항상 이득은 아니라는 점**도 보입니다. (C/C++ 같은 언어에서는 멀티스레딩으로 멀티코어 활용 가능)

2.3 CPU 스케줄링

여러 프로세스(또는 커널 스레드)가 있는 상황에서 **CPU를 어떤 순서로, 얼마나 할당할 것인지 결정**하는 것이 CPU 스케줄링입니다 [31]. 운영체제는 공정성과 효율, 응답 시간 향상 등의 목표를 이루기 위해 다양한 스케줄링 알고리즘을 사용합니다.

2.3.1 프로세스 우선순위와 스케줄링 큐

운영체제는 각 프로세스를 **Ready Queue**에 두고 관리합니다. 이 대기열에서 **스케줄러**가 다음 실행할 프로세스를 선택합니다. 선택 시 고려 요소 중 하나가 **우선순위(priority)**입니다. 우선순위가 높은 프로세스는 CPU를 먼저 배정받는 경향이 있으며, 일부 시스템에서는 시간이 흐르며 우선순위를 동적으로 조정(보통 Aging 기법으로 기아현상 방지)하기도 합니다.

- **비선점 스케줄링 (Non-preemptive):** 한 프로세스가 CPU를 잡으면 자진 반납(블록 또는 종료)하기 전까지 뺏지 않는 방식.
- **선점 스케줄링 (Preemptive):** 운영체제가 정해진 조건/타이밍에 강제로 CPU를 회수하여 다른 프로세스로 넘길 수 있는 방식. 현대 일반 OS는 선점형이 많습니다 (시분할 시스템 등).

2.3.2 CPU 스케줄링 알고리즘

여러 알고리즘이 존재하며, 상황에 맞게 사용됩니다:

- **FCFS (First-Come, First-Served):** 도착 순서대로 처리. 구현이 간단하지만 긴 작업이 앞에 오면 **Convoy 효과**로 효율이 떨어질 수 있습니다.

- **SJF (Shortest Job First):** 실행 시간(CPU burst)가 가장 짧은 작업을 먼저 스케줄. 이론적으로 평균 대기시간이 최소화되지만, 각 작업의 남은 실행 시간을 예측해야 하는 어려움이 있습니다. (실제로는 예측 기반 SJF 비슷한 기법 사용)
- **Round Robin (RR):** 시분할 시스템에서 흔히 사용. 각 프로세스에 **타임퀀텀** 만큼 CPU를 주고, 시간이 끝나면 선점하여 다음 프로세스로 넘어감. 응답성을 향상하지만, 퀀텀이 너무 짧으면 Context Switch overhead 증가.
- **Priority Scheduling:** 우선순위 높은 순으로 실행. 우선순위 낮은 프로세스가 기아(starvation)하지 않도록 주의 (aging으로 점진적 우선순위 상승).
- **MLFQ (Multi-Level Feedback Queue):** 다단계 큐 + 피드백을 활용하여 인터랙티브 작업과 배치 작업을 차별 처리. 예를 들어 처음엔 높은 우선순위 큐에 넣되, CPU 오래 쓰면 점차 낮은 큐로 내려서 긴 작업은 나중에 처리.

운영체제는 흔히 **다중 요인**을 고려한 복합 스케줄러를 사용합니다. Linux의 CFS(Completely Fair Scheduler)는 각 프로세스의 가상 실행 시간 등을 기반으로 **공정한 CPU 분배**를 추구합니다. Windows scheduler도 우선순위 클래스와 라운드 로빈을 조합합니다.

2.4 프로세스 동기화와 교착상태 (동시성 제어)

다중 프로세스/스레드 환경에서 **공유 자원 접근**을 조율하는 것은 매우 중요합니다. 잘못하면 **경쟁 상태(Race Condition)**로 인해 데이터 불일치나 예기치 않은 동작이 발생합니다 [20]. 이를 막기 위해 **동기화(sync)** 기법들이 사용되며, 이 부분은 운영체제 이론의 핵심 중 하나입니다. 또한 여러 프로세스/스레드가 **영원히 서로를 기다리는 상황(Deadlock)**에 빠질 위험도 존재합니다.

2.4.1 임계구역 문제와 락 (Mutex)

둘 이상의 스레드가 동시에 접근하면 안 되는 공유 자원 코드 영역을 **임계 구역(Critical Section)**이라고 합니다. 이 문제를 해결하기 위한 기본 도구는 **뮤텍스(Mutex, 상호배제 락)**입니다 [32]. 뮤텍스는 한 번에 오직 하나의 스레드만 임계구역에 들어갈 수 있도록 잠금/해제를 제공하는 간단한 객체입니다. 사용 예:

```
import threading
lock = threading.Lock()
shared_counter = 0

def increment():
    global shared_counter
    for i in range(100000):
        lock.acquire()       # 잠금
        shared_counter += 1  # 임계구역 (공유 변수 수정)
        lock.release()       # 잠금 해제

# 스레드 2개 생성
t1 = threading.Thread(target=increment)
t2 = threading.Thread(target=increment)
t1.start(); t2.start()
t1.join(); t2.join()
print(shared_counter)
```

만약 위에서 `lock.acquire()/release()`를 사용하지 않으면 두 스레드가 동시에 `shared_counter`를 증가시키는 과정에서 일부 증가가 덮어써져 결과값이 200000이 안 될 수 있습니다. 락으로 감싸면 항상 하나씩 순서대로 실행되어 정확한 결과를 얻습니다.

파이썬 GIL처럼 **한 번에 하나의 스레드만 바이트코드를 실행**하는 환경에서도, I/O가 있거나 멀티프로세스 상황에서는 이런 락이 필요합니다. C/C++에서는 `pthread_mutex` 등을 사용하고, 자바에서는 `synchronized` 키워드 등 언어 차원 지원이 있습니다.

[실습] 생산자-소비자 문제 (조건변수와 세마포어 활용)

생산자-소비자 문제는 고전적인 동기화 문제로, 공유 버퍼를 사이에 두고 생산자 스레드들은 데이터를 생성해 버퍼에 넣고, 소비자 스레드들은 버퍼에서 꺼내 처리하는 상황입니다 [20]. 버퍼가 가득 차면 생산자는 대기, 버퍼가 비면 소비자는 대기해야 합니다. 이 문제를 푸는 동기화 도구로 **세마포어(Semaphore)**와 **조건 변수(Condition Variable)**가 자주 언급됩니다.

- **세마포어:** 정수 값을 가지며, P(wait) 연산 시 값을 감소하고 0 이하이면 대기, V(signal) 연산 시 값을 증가하고 기다리는 스레드가 있으면 깨우는 동기화 수단입니다. 세마포어 값을 버퍼의 남은 공간 개수나 아이템 개수로 설정하여 생산자/소비자를 제어합니다.
- **조건 변수:** 특정 조건(예: 버퍼 비거나 참)을 기다리는 스레드를 잠재우고, 조건이 충족되면 깨우는 메커니즘. 표준 스레드 라이브러리에서 락과 함께 사용됩니다.

파이썬에서는 `queue.Queue` 클래스가 내부적으로 이 문제를 해결한 구현이며, 쓰레드 간 안전하게 동작합니다. 간단한 예로, 하나의 생산자와 하나의 소비자가 숫자를 주고받는 상황을 만들어 보겠습니다:

```
import threading, time
from collections import deque

buffer = deque(maxlen=5)       # 최대 5개 아이템 버퍼
condition = threading.Condition()

def producer():
    for i in range(10):
        with condition:
            # 버퍼가 가득 차면 소비자 처리 기다림
            while len(buffer) == buffer.maxlen:
                condition.wait()   # wait releases the lock and waits
            # 아이템 생산
            buffer.append(i)
            print(f"Produced {i}, buffer size={len(buffer)}")
            condition.notify()     # 소비자 깨움
        time.sleep(0.1)  # 생산 속도 조절

def consumer():
    for i in range(10):
        with condition:
            # 버퍼가 비었으면 생산자 대기
            while len(buffer) == 0:
```

```
                condition.wait()
            item = buffer.popleft()
            print(f"Consumed {item}, buffer size={len(buffer)}")
            condition.notify()
        time.sleep(0.15)  # 소비 속도 조절

# 시작
t_prod = threading.Thread(target=producer)
t_cons = threading.Thread(target=consumer)
t_prod.start(); t_cons.start()
t_prod.join(); t_cons.join()
print("Production & Consumption done.")
```

실행하면 생산자가 아이템을 만들고, 소비자가 가져가는 과정이 interleaving되어 출력됩니다. `condition.wait()`과 `condition.notify()`로 버퍼 상태에 따라 서로를 깨우고 잠드는 동기화가 이뤄집니다.

이처럼 동기화 도구를 적절히 활용하면 여러 스레드가 협업하여 안전하게 일을 처리할 수 있습니다. 주의할 점은 락을 오래 잡고 있으면 성능 저하나 데드락이 발생할 수 있으므로, 최소한의 범위에서만 보호하는 것입니다.

2.4.3 교착상태(Deadlock)와 해결 방법

교착상태(Deadlock)란 둘 이상의 프로세스(또는 스레드)가 **서로가 가진 자원**을 기다리며 **무한정 대기**하는 상황을 말합니다[33]. 예를 들어 스레드 A는 락 X를 잡은 상태에서 락 Y를 기다리고, 스레드 B는 락 Y를 잡은 채 락 X를 기다린다면 둘 다 영원히 기다리게 됩니다.

데드락 발생 조건 (Coffman 조건): 1. 상호배제: 자원은 한 번에 한 프로세스만 사용. 2. 점유대기: 자원을 가진 상태에서 다른 자원 기다림. 3. 비선점: 자원을 강제로 빼앗을 수 없음. 4. 순환대기: 프로세스들 사이에 자원 대기 원형이 존재.

해결/회피 방법: - **교착상태 예방**: 위 조건 중 하나 이상을 사전에 부정하는 정책. 예를 들어 필요한 자원을 한꺼번에 확보하도록 (점유대기 부정) 하거나, 자원에 순서를 부여해 항상 순서대로 취득(순환대기 조건 방지)하는 방법. - **교착상태 탐지 및 회복**: 데드락이 발생했는지 검사(그래프 사이클 탐지 등)하고, 발견되면 프로세스 종료나 자원 선점 등으로 풀어주는 방법. - **교착상태 회피**: 자원 할당 시 **은행원 알고리즘** 같은 것으로 안전 상태인지 계산하여, 위험하면 할당을 안 하는 방식. 실행에 제약이 크지만 이론적으로 가능.

실무에서는 데드락 예방보다는, 심각하지 않은 경우 그냥 두거나(사용자 개입 필요 시 경고), 운영체제가 아닌 상위 레벨에서 타임아웃 걸기 등으로 처리하는 경우가 많습니다. 프로그래머 측면에서는 **락 획득 순서의 통일** 등으로 애초에 데드락이 생기지 않도록 코딩하는 것이 중요합니다.

2.5 가상 메모리 관리

가상 메모리(Virtual Memory)는 한정된 물리 메모리를 효율적으로 활용하고 프로세스 격리를 제공하는 운영체제 기술입니다[34]. 이 절에서는 페이징, 페이지 교체 등 가상 메모리 기법에 대해 살펴봅니다.

2.5.1 페이징과 페이지 테이블

현대 OS는 **페이징(Paging)**을 사용하여 가상 메모리를 관리합니다. **페이지(Page)**란 가상 메모리를 고정 크기 블록(예: 4KB)으로 쪼갠 단위이고, **프레임(Frame)**은 물리 메모리의 같은 크기 블록을 의미합니다 [34]. 운영체제는 각 프로세스의 가상 페이지들이 어떤 물리 프레임에 매핑되는지를 **페이지 테이블(Page Table)**에 기록합니다.

- 가상 주소는 (페이지 번호, 페이지 내 오프셋)으로 구성되고, 페이지 테이블을 통해 페이지 번호 -> 프레임 번호로 변환되어 물리 주소가 결정됩니다.
- 모든 가상 페이지가 실제 물리 메모리를 차지할 필요는 없습니다. 실제 필요할 때에만 물리 프레임에 올리고, 당장 필요 없는 페이지는 디스크의 스왑 영역으로 쫓아낼 수 있습니다. 이를 **요구 페이징(Demand Paging)**이라 합니다 [35].

2.5.2 페이지 부재와 스와핑 (Page Fault & Swapping)

프로세스가 참조한 가상 페이지가 현재 물리 메모리에 없으면 **페이지 부재(Page Fault)**가 발생합니다. 그러면 OS는 디스크(스왑 영역)에서 해당 페이지를 읽어와 **빈 프레임**에 적재하고, 페이지 테이블을 갱신한 후 프로세스를 다시 실행시킵니다 [36]. 이 과정에서 **페이지 교체**가 필요할 수 있습니다: 만약 빈 프레임이 없다면, **사용 중이지 않은 페이지** 하나를 골라 디스크로 내보내고 그 프레임을 재사용합니다.

스와핑(swapping)은 오래된 용어로, 프로세스 전체를 통째로 디스크에 내보냈다 다시 들이는 개념이지만, 현대 OS에서는 필요한 개별 페이지 단위로 교체하는 것이 일반적입니다. **스레싱(Thrashing)**은 페이지 부재가 너무 빈번하여 CPU가 실제 유용 작업보다 페이지 교체에 시간을 다 써버리는 상황을 말합니다 [36]. 이는 프로세스들이 과도한 메모리를 요구할 때 발생하며, 워킹셋 모델 등을 활용하여 완화합니다.

2.5.3 페이지 교체 알고리즘

빈 프레임이 없을 때 **어느 페이지를 내보낼지** 결정하는 알고리즘입니다 [37]. 대표적인 방법들:

- **FIFO 알고리즘:** 가장 먼저 들어온 페이지부터 내보냅니다. 구현이 쉽지만 성능은 그다지 좋지 않을 수 있습니다 (Belady의 anomaly: 프레임 수 늘려도 page fault 증가하는 현상).
- **OPT (Optimal) 알고리즘:** 이론상 **앞으로 가장 오랫동안 사용되지 않을 페이지**를 교체. 미래를 알아야 하므로 실제 구현 불가, 다만 최적 기준으로 비교용.
- **LRU (Least Recently Used):** 최근 가장 오랫동안 안 쓰인 페이지를 교체 [38]. 시간 지역성 가정에 근거하여 꽤 성능이 좋습니다. 구현은 최근 사용 시각 기록 또는 참조 비트/카운터 이용.
- **LFU (Least Frequently Used):** 사용 횟수가 가장 적은 페이지 교체.
- **Clock 알고리즘 (Second Chance):** 원형 버퍼 형태로 페이지를 가리키며, 참조 비트 이용하여 2차 기회를 주는 LRU 근사법. 페이지에 참조 비트가 1이면 0으로 리셋하며 패스, 0인 페이지 찾으면 교체.

운영체제들은 LRU 근사 알고리즘 (Clock 변형들) 등을 실제로 사용합니다. Linux는 세그멘테이션 된 LRU 리스트 (활성/비활성 리스트) 등 복잡한 기법을 사용하고, Windows도 비슷하게 동작합니다.

2.5.4 메모리 단편화와 할당

가상 메모리에서 **내부 단편화(internal fragmentation)**와 **외부 단편화(external fragmentation)** 이슈도 언급됩니다. 페이징 기법에서는 **내부 단편화**(페이지당 일부 남는 용량 사용 못하는) 이슈가 있을 수 있으나, 고정 블록이므로 외부 단편화는 생기지 않습니다. 반면 과거 **세그멘테이션(segmentation)** 기법에서는 가변 크기 블록으로 외부 단편화 문제가 있었고, 이를 페이징이 해결했습니다.

Heap 할당 등에서는 여전히 외부 단편화 우려가 있으며, 주기적으로 메모리 compaction을 하거나, 또는 애플리케이션 레벨에서 메모리 풀링 등을 사용하기도 합니다.

2.5.5 Copy-on-Write

Copy-on-Write(CoW)는 프로세스 생성 시 효율을 높이는 기술입니다 [39]. 앞서 언급한 `fork()` 시스템 콜은 부모 프로세스의 주소공간을 거의 동일하게 복제하는데, 이때 실제 메모리를 바로 복사하지 않고 부모와 자식이 같은 물리 페이지들을 공유하게 합니다(모두 read-only로 표시). 그러다 둘 중 하나가 해당 페이지를 수정하려 할 때 **그제서야 복사**하여 분리된 공간을 만들어줍니다. 이를 통해 불필요한 메모리 복사와 낭비를 줄입니다.

Linux 등 유닉스 계열 OS는 fork+exec 조합에서 이 CoW를 적극 활용하여, 프로세스 생성 비용을 줄이고 있습니다. Python에서 멀티프로세싱 라이브러리 spawn vs fork 모드 등의 차이에도 이런 메커니즘이 관련 있습니다.

2.5.6 캐시 메모리와 가상메모리 (캐시 정책)

(이 부분은 컴퓨터 구조와 맞물리지만 운영체제 메모리 관리와도 관련되어 잠깐 언급)

CPU 캐시에도 **교체 알고리즘**과 **쓰기 정책** 등이 있습니다: - 캐시 교체: LRU, FIFO 등 (일반적으로 하드웨어적으로 LRU 근사 사용). - 쓰기 정책: **Write-through**(캐시 변경 시 즉시 메모리에도 반영) vs **Write-back**(캐시에만 쓰고 나중에 메모리 반영) 등이 있습니다. Write-back은 성능 이점 있지만 복잡.

또, 가상 메모리를 사용하면 캐시 인덱싱 시 **가상 주소 vs 물리 주소** 중 무엇을 쓰냐에 따른 문제로 **TLB(Translation Lookaside Buffer)**라는 캐시-페이지테이블 간 촉진장치도 필요합니다. 이는 아키텍처 세부사항으로 여기서는 깊게 다루지 않습니다.

2.6 파일 시스템

파일 시스템(File System)은 보조기억장치에 데이터를 체계적으로 저장하고 관리하는 방법을 제공하는 운영체제의 일부입니다 [40]. 사용자는 파일 시스템을 통해 파일/디렉터리를 만들고 삭제하며, 데이터를 영구적으로 관리할 수 있습니다.

2.6.1 파일과 디렉터리

파일(File)은 연속된 바이트들의 모음이며, 프로그램이나 문서 등의 데이터를 담는 논리 단위입니다. 운영체제는 파일에 **이름**을 부여하고 **디렉터리(Directory)**를 통해 계층 구조로 파일을 조직합니다 [40].

디렉터리는 일종의 파일들의 목차로, 각 파일의 이름과 해당 파일의 메타데이터(위치, 크기 등)를 보관합니다. 디렉터리 역시 이름을 가지며 다른 디렉터리를 포함할 수 있어, 트리 형태의 구조(혹은 DAG 형태로 링크 가능)로 운영됩니다. Windows, Unix 등에서 최상위 루트 디렉터리부터 시작하여 경로를 통해 파일을 접근하는 방식은 익숙할 것입니다.

2.6.2 파일 시스템이 만들어지기까지 (포맷 과정)

디스크에 파일 시스템을 사용하려면 먼저 **포맷(format)**을 해야 합니다. 포맷 과정에서는 해당 파티션에 **파일 시스템 구조(메타데이터)**를 초기화합니다:

- **슈퍼블록(Superblock)** 생성: 파일시스템 전체 정보를 담은 블록 (파일 시스템 종류, 크기, 상태, 메타데이터 위치 등).

- **아이노드 테이블 또는 FAT 등**: 파일별 메타데이터 구조 공간 마련. 유닉스계 파일 시스템은 각 파일마다 **inode**라는 구조에 파일 크기, 소유자, 권한, 데이터 블록 위치 목록 등을 기록합니다. FAT나 NTFS 등은 각기 다른 구조를 사용.
- **빈 공간 관리 구조**: 프리 블록 비트맵 또는 프리 리스트 초기화.
- **루트 디렉터리 생성**: 최상위 디렉터리를 하나 만들고 inode 할당 (유닉스의 "/" 등).

이후 파일 생성/삭제 등의 연산이 이 구조를 업데이트하며 수행됩니다.

2.6.3 [실습] 간이 파일 시스템 만들어보기 (개념 실습)

운영체제 강의에서는 종종 간단한 파일 시스템을 디스크 이미지 파일 위에 만드는 실습을 합니다. 여기선 파이썬으로 극히 단순화된 파일시스템 동작을 시뮬레이션해보겠습니다. (실제 OS 레벨 파일시스템 코드는 매우 복잡하므로, 이해를 돕기 위한 모형입니다.)

우리는 1MB 크기의 가상 디스크를 1KB 블록 1024개로 구성된 것으로 가정하고, 다음과 같은 구조로 관리하겠습니다: - 슈퍼블록(블록 0): [free블록비트맵(1024bit = 128byte) | (기타 예외)] - 루트 디렉터리도 하나의 특수 파일로 간주, 블록 1을 사용. - 파일의 메타데이터 (파일크기, 첫번째 데이터블록 등) 매우 단순화하여 디렉터리에 저장.

```
# 간단한 파일 시스템 이미지 생성 및 관리 (실제 OS FS와 다름 주의)
import io

BLOCK_SIZE = 1024
NUM_BLOCKS = 1024

# 1MB 가상 디스크 (바이트배열로 초기화)
disk = bytearray(BLOCK_SIZE * NUM_BLOCKS)

# 초기화: 슈퍼블록에 free bitmap 설정 (처음엔 0,1번 블록 사용중으로 표시, 나머지 free)
free_bitmap = [1] * NUM_BLOCKS
free_bitmap[0] = 0  # block 0 occupied by superblock
free_bitmap[1] = 0  # block 1 for root directory
# bitmap를 바이트array로 디스크에 기록
for i in range(NUM_BLOCKS):
    if free_bitmap[i]:
        disk[i//8] |= (1 << (i % 8))
# (root 디렉터리 초기화 - 여기서는 비어있음)

# 파일 생성 함수: 이름과 내용 주어지면 새로운 블록 할당 후 데이터 기록
def create_file(name: str, data: bytes):
    # 간단히: 첫 번째 빈 블록 찾아 사용
    try:
        idx = free_bitmap.index(1)
    except ValueError:
        print("Disk full!")
        return False
    free_bitmap[idx] = 0
    # update bitmap on disk
```

```
    disk[idx//8] &= ~(1 << (idx % 8))
    # 파일 데이터 기록 (한 블록 넘는 경우 미처리)
    disk[idx * BLOCK_SIZE : idx * BLOCK_SIZE + len(data)] = data
    # 디렉터리 엔트리 추가 (여기서는 루트 디렉터리 블록에 name->블록번호 기록)
    dir_entry = f"{name}:{idx}\n".encode()
    root_dir_offset = 0
    disk[1 * BLOCK_SIZE + root_dir_offset : 1 * BLOCK_SIZE + root_dir_offset +
len(dir_entry)] = dir_entry
    print(f"Created file '{name}' at block {idx}, size={len(data)} bytes")
    return True

# 파일 읽기 함수
def read_file(name: str):
    # 루트 디렉터리에서 이름 검색
    raw = disk[1*BLOCK_SIZE : 2*BLOCK_SIZE]
    content = raw.split(b'\n')
    for entry in content:
        if entry:
            fname, blk = entry.decode().split(':')
            if fname == name:
                blk = int(blk)
                # read data from that block (until a stop or end of block)
                file_data = bytes(disk[blk*BLOCK_SIZE : (blk+1)*BLOCK_SIZE])
                print(f"Reading file '{name}' from block {blk}: {file_data[:
20]}...")
                return file_data
    print(f"File {name} not found")
    return None

# 테스트: 파일 만들어서 읽기
create_file("hello.txt", b"Hello OS World!")
read_file("hello.txt")
```

위 코드는 매우 단순화된 파일 시스템으로, 파일당 하나의 데이터블록만 할당, 디렉터리 엔트리도 "이름:블록번호"로 기록하는 식입니다. 테스트로 "hello.txt" 파일을 만들고 읽어보면, 데이터가 잘 쓰이고 읽히는 것을 확인할 수 있습니다.

실행 결과 예시:

```
Created file 'hello.txt' at block 2, size=15 bytes
Reading file 'hello.txt' from block 2: b'Hello OS World!
\x00\x00\x00\x00\x00\x00'...
```

(여기 `\x00` 등은 블록 내 남은 부분이 0으로 채워진 모습입니다.)

이 예제를 통해 파일 생성 시 빈 블록 할당, 디렉터리에 등록, 데이터 쓰기 등의 과정을 간략히 모형화해봤습니다. 실제 파일 시스템(예: ext4, NTFS)은 다단계 인덱스(inode의 direct/indirect blocks, FAT 테이블 등), 저널링, 권한 관리 등 훨씬 복잡한 기능을 구현하고 있습니다.

2.6.4 파일 시스템의 종류와 특성

역사적으로 다양한 파일 시스템들이 존재하며, 각각 구조와 성능 특성이 다릅니다. 몇 가지 예:

- **FAT32/exFAT:** 옛날 DOS/윈도우 계열 파일시스템. 단순한 테이블 구조로 임플리먼트 쉬우나 대용량/보안에 한계. (exFAT는 개선판)
- **NTFS:** 최신 윈도우 기본 FS. 저널링 지원, 보안 ACL, 압축, 암호화 등 기능 풍부.
- **ext2/ext3/ext4:** 리눅스 계열 대표 FS. ext3부터 저널링 도입, ext4는 대용량 파일, 익스텐트 등 지원.
- **XFS, Btrfs, ZFS:** 대용량, 스냅샷, 풀기능 등 다양한 특화 FS.
- **분산 파일 시스템 (HDFS 등):** 클러스터/분산환경용 FS도 있음.

파일 시스템은 **신뢰성**을 위해 **저널링** 기법을 많이 사용합니다. 메타데이터 변경을 로그에 기록해 두었다가, 시스템 크래시 시 로그를 분석해 파일시스템을 복구하는 방식입니다 [18] (AI와 직접 관련은 없지만, 서버 측면에서는 중요).

2.7 컨테이너 (Containers)

마지막으로 운영체제 파트의 최신 주제로 **컨테이너(Container)** 기술을 간략히 살펴봅니다 [41]. 컨테이너는 OS 레벨 가상화를 통해 프로세스를 격리하는 기술로, 대표적으로 Docker가 있습니다.

2.7.1 컨테이너의 개념과 동작 원리

컨테이너는 **하나의 운영체제 커널 위에서 동작하지만** 마치 별도의 환경인 것처럼 프로세스들을 격리합니다. 이는 리눅스의 **네임스페이스(namespaces)**와 **cgroups(control groups)** 기능으로 구현됩니다:

- **네임스페이스:** 프로세스 그룹별로 격리된 자원 뷰를 제공합니다. 예를 들어 PID 네임스페이스를 활용하면 각 컨테이너 안에서 PID 1부터 프로세스 번호를 보며, 다른 컨테이너의 프로세스는 보이지 않습니다. 그 외에 UTS(호스트네임), Mount(파일시스템 경로 격리), Network(가상 NIC, IP 격리), IPC, User(UID 격리) 등의 네임스페이스가 있습니다 [18].
- **cgroups:** CPU, 메모리 등의 자원 사용량을 그룹별로 제한하고 계측할 수 있는 기능입니다. 컨테이너마다 CPU 쿼터나 메모리 상한 등을 설정해 **자원 할당**을 제어합니다.

Docker 컨테이너를 예로 들면, 컨테이너 이미지를 통해 격리된 **파일시스템(루트fs)**을 제공하고, 프로세스들은 그 안에서 실행되며, 네트워크도 가상 브릿지를 통해 격리된 서브네트워크를 갖습니다. 그러나 모든 컨테이너는 동일한 OS 커널을 공유하기 때문에 VM에 비해 오버헤드가 적고 성능 손실이 적습니다.

2.7.2 컨테이너 오케스트레이션 (쿠버네티스 등) [심화]

컨테이너를 대규모로 사용하게 되면서, **Kubernetes**와 같은 컨테이너 오케스트레이션 툴이 등장했습니다 [18]. 이는 여러 대의 물리/가상 서버에 걸쳐 컨테이너들의 실행, 스케일링, 롤아웃/롤백 등을 자동화해주는 시스템입니다. AI 서비스나 마이크로서비스 아키텍처에서 수십 수백 개의 컨테이너를 관리하기 위해 필수적인 기술로 자리 잡았습니다.

2.7.3 컨테이너 vs VM

- **경량성:** 컨테이너는 별도의 OS 커널을 부팅하지 않으므로 수 초 내 기동 가능하고 메모리 사용도 효율적입니다. VM은 각자 OS가 있어서 수 분 걸릴 수도 있고, 더 많은 자원을 잡아먹습니다.
- **격리수준:** VM은 하드웨어 레벨 격리라 보안/안정성 측면에서 더 강력하나, 컨테이너도 네임스페이스로 상당 수준 격리됩니다. 그러나 같은 커널 공유 때문에 취약점 공격면이 VM보다 넓습니다.
- **호환성:** VM은 서로 다른 OS(예: Windows host에서 Linux VM 실행)가 가능하지만, 컨테이너는 호스트 커널과 같은 종류의 OS만 가능합니다 (Windows에서 Linux 컨테이너 X, 반대도 X, 단 Windows에 리눅스 커널 올린 WSL2 기술 등은 예외).
- **배포편의:** 컨테이너 이미지는 애플리케이션과 의존 라이브러리만 포함하므로 작고 이식성이 좋습니다. "이미지 빌드 -> 실행"으로 환경 일치 보장이 쉬워 DevOps에 많이 쓰입니다.

정리하면, **컨테이너**는 운영체제의 격리/자원제어 기능을 활용한 경량 가상화로서, 현대 클라우드 환경에서 애플리케이션 배포의 표준이 되고 있습니다.

以上가 운영체제 파트의 내용입니다. 운영체제는 컴퓨터 자원의 **관리자** 역할을 하며, 프로세스/스레드, 메모리, 파일시스템, I/O, 가상화 등의 다양한 주제를 포괄합니다. 다음 파트에서는 **네트워크**로 넘어가겠습니다.

Part 3. 네트워크

네트워크 파트에서는 **컴퓨터 네트워킹**의 기본 개념과 인터넷 프로토콜 구조를 다룹니다. 네트워크는 AI 시대에도 **분산 학습, 클라우드 연동, 서비스 배포** 등에서 중요한 역할을 하므로 필수적인 지식입니다. 주요 내용으로 **OSI 7계층 모델**, TCP/IP 4계층 모델, 각 계층의 프로토콜 (이더넷, IP, TCP/UDP, HTTP 등) 그리고 네트워크 장비와 성능, 보안 개념까지 폭넓게 학습합니다.

3.1 네트워크 개요 – 거시적으로 보기

컴퓨터 네트워크란 떨어진 컴퓨터들 간에 데이터를 주고받을 수 있게 연결한 시스템입니다. 네트워킹의 큰 그림으로 흔히 **프로토콜 스택** 개념을 소개하는데, 대표적으로 **TCP/IP 모델**(혹은 OSI 모델)이 있습니다.

- **OSI 7계층 모델:** 응용(Application), 표현(Presentation), 세션(Session), 전송(Transport), 네트워크(Network), 데이터 링크(Data Link), 물리(Physical) 7계층으로 이상적으로 나눠 설명합니다. 현실 프로토콜은 이보단 합쳐진 TCP/IP 4계층으로 보는 편.
- **TCP/IP 4계층 모델:**
- **응용 계층:** HTTP, FTP, SMTP 등 최종 애플리케이션 프로토콜들이 동작 [42].
- **전송 계층:** TCP, UDP 등이 속하며, 프로세스 간 신뢰성/포트/데이터 전송을 담당 [43].
- **인터넷 계층:** IP 프로토콜 – 논리적 주소 지정(IP 주소)와 패킷 라우팅 담당 [44].
- **네트워크 액세스 계층 (링크 계층):** 이더넷, Wi-Fi 등 실제 링크에서 **프레임** 전송 담당 [45].

이러한 **계층화**는 복잡한 네트워크 기능을 분리하여 구현/이해를 쉽게 해줍니다. 각 계층은 **자신의 기능을 수행**하고, 바로 아래 계층에 서비스를 요청하는 식으로 동작합니다 [46]. 데이터를 송신할 때 상위 계층에서 하위 계층으로 내려갈수록 **캡슐화(encapsulation)**가 일어나 헤더가 붙고, 수신 측에서 역순으로 **비캡슐화(decapsulation)**되어 최종 데이터를 얻습니다 [47].

네트워크 성능을 볼 때는 **대역폭(bandwidth)**과 **지연(latency)**을 주로 언급합니다. 대역폭은 초당 비트 전송량 (bps), 지연은 한 쪽에서 다른 쪽까지 신호가 가는 데 걸리는 시간입니다. 예를 들어 광통신은 전기통신보다 지연이 낮고, 기가비트 이더넷은 100Mbps 이더넷보다 대역폭이 높습니다 [47].

[실습] 와이어샤크로 패킷 캡처 분석 (개념 설명)

와이어샤크(Wireshark)는 네트워크 패킷 캡처 및 분석에 널리 쓰이는 도구입니다. 예컨대 자신의 PC에서 Wireshark를 실행하고, 특정 인터페이스의 트래픽을 캡처하면, 다양한 프로토콜의 패킷들이 캡처됩니다 [47]. 웹 브라우저로 페이지를 여는 동안 캡처했다면, ARP 요청/응답, DNS 질의/응답, TCP SYN/SYN-ACK/ACK 핸드쉐이크, HTTP GET/응답 등의 패킷들을 시각적으로 확인할 수 있습니다.

예를 들어 Wireshark에서 어떤 패킷을 클릭하면, **Ethernet II 헤더**, **IPv4 헤더**, **TCP 헤더**, **HTTP 프로토콜 데이터** 등이 계층적으로 디코딩되어 보입니다. 이를 통해 캡슐화된 각 계층 헤더 필드(IP 주소, MAC 주소, 포트, 시퀀스 번호 등)를 직접 확인할 수 있습니다.

본 강의에서는 실습으로 Wireshark를 사용하여 ARP 패킷, ICMP (ping) 패킷, TCP 3-way handshake, HTTP 요청/응답 등을 관찰하는 활동이 있습니다. (지면상 여기서는 설명으로 대체하지만, 독자께서는 직접 Wireshark를 사용해보길 권장합니다.)

3.2 네트워크 액세스 계층 (링크 계층)

네트워크 액세스 계층은 **LAN 등 근거리 네트워크에서 데이터 전송** 을 담당합니다 [48]. 대표적인 프로토콜로 **이더넷(Ethernet)**이 있고, 장비로는 **허브(Hub)**, **스위치(Switch)** 등이 있습니다.

3.2.1 이더넷과 MAC 주소, CSMA/CD

이더넷(Ethernet)은 가장 널리 쓰이는 LAN 기술로, 유선 케이블을 통해 프레임(frame)을 전송합니다 [48]. 각 이더넷 장치에는 전 세계에서 고유한 **MAC 주소**(Media Access Control address, 48비트)가 할당되어 있어서, 이더넷 프레임에 출발지/목적지 MAC 주소를 담습니다.

Ethernet의 중요한 기술 요소 중 하나는 **CSMA/CD** (Carrier Sense Multiple Access with Collision Detection)입니다 [48]: - **다중 접속 (Multiple Access):** 여러 노드가 하나의 채널을 공유. - **반송파 감지 (Carrier Sense):** 송신 전에 채널이 사용 중인지 감지. - **충돌 탐지 (Collision Detection):** 두 노드가 동시 송신해 충돌 발생 시 이를 감지하고 송출 중단 후 랜덤 백오프.

CSMA/CD는 **공유 버스 형태**의 이더넷 (허브나 공용 매체)에서 사용되었습니다. 오늘날 대부분은 **스위치** 기반으로 변하여 충돌이 논리적으로는 일어나지 않지만, 기본 개념으로 알고 있어야 합니다. (Wi-Fi는 충돌 회피 CSMA/CA 사용)

3.2.2 허브와 스위치, VLAN

- **허브(Hub):** 이더넷 허브는 물리계층 장비로, 받은 신호를 단순 중계(모든 포트에 재전송)합니다. 그래서 허브 환경에서는 동시에 두 노드가 보내면 충돌이 발생하며, 하나의 충돌 도메인입니다 [48].
- **스위치(Switch):** 스위치는 **MAC 주소 테이블**을 가지고 프레임의 도착 포트와 송신 MAC을 학습하여, 목적지 MAC을 보고 적절한 포트로만 프레임을 전달합니다. **다수의 포트에 각각 충돌 도메인을 분리**하여 동시에 여러 통신 가능. 스위치는 데이터링크 계층 장비입니다.

- **VLAN (Virtual LAN):** 스위치는 논리적으로 포트들을 그룹핑하여 서로 다른 LAN인 것처럼 격리할 수 있습니다. VLAN tagging (IEEE 802.1Q)으로 프레임에 VLAN ID를 넣어, 같은 VLAN끼리만 통신되게 합니다. 한 스위치 내 여러 가상 네트워크를 만드는 셈입니다.

정리하면, **허브**는 바보같이 모두에게 방송하고, **스위치**는 똑똑하게 MAC을 학습하여 필요한 곳으로만 보냅니다. 오늘날 허브는 거의 쓰이지 않고 다 스위치입니다.

3.3 네트워크 계층 (Network Layer)

네트워크 계층에서는 **논리 주소(IP) 부여**와 **라우팅(Routing)**을 담당합니다 [49]. 전 세계 모든 인터넷 장치가 유일한 IP 주소를 가지며, 라우터들이 패킷을 목적지 IP까지 전달합니다.

3.3.1 IP 프로토콜과 IPv4/IPv6

IP (Internet Protocol)는 비신뢰성, 비연결형의 **패킷 교환(protocol)**을 제공합니다. 현재 IPv4와 IPv6 두 버전이 쓰입니다:

- **IPv4:** 32비트 주소 (약 43억 개 주소 공간). 주소 표현은 점으로 구분된 10진수 4개 ("192.168.0.1"). IPv4 헤더에는 출발지/목적지 IP, TTL, 상위 프로토콜(TCP/UDP) 식별 등 정보가 담깁니다. IPv4는 주소 부족 문제로 현재 **사설 IP + NAT**로 연명 중입니다.
- **IPv6:** 128비트 주소 (사실상 무한대). 표현은 16진수 8개 묶음으로 표시("2001:0db8:85a3::8a2e:0370:7334"). IPv6는 확장 헤더, 간소화된 처리(헤더 체크섬 제거 등)로 효율 개선, 그리고 **자동 주소 구성** 등의 장점이 있습니다.

IP는 **최선형(best-effort)** 전달만 보장하며, 패킷이 중간에 손실될 수도, 순서 뒤바뀔 수도, 지연될 수도 있습니다. 상위에서 이를 처리해야 합니다.

3.3.2 ARP (Address Resolution Protocol)

ARP는 동일 LAN 내에서 **IP 주소를 MAC 주소로 변환**하는 프로토콜입니다. IP 패킷을 실제로 보내려면 목적지 MAC이 필요하기 때문입니다 [50]. ARP의 동작: - 송신 노드가 ARP 요청 브로드캐스트("이 IP 가진 노드야, 너의 MAC은 뭐냐?")를 보냄 [51]. - 그 IP를 설정한 노드가 ARP 응답(자신의 MAC 주소 포함)을 보냄. - 송신 노드는 IP<->MAC 매핑을 ARP 캐시에 저장해둠 (다음부터는 바로 사용).

예를 들어 192.168.1.10 -> 192.168.1.20으로 패킷 보내려면, 일단 192.168.1.20의 MAC을 모르면 ARP로 알아낸 다음, Ethernet 헤더의 목적지 MAC에 그 MAC을 넣어 프레임을 전송합니다.

ARP는 IPv4에서 쓰이고, IPv6에서는 비슷한 역할을 **Neighbor Discovery**가 ICMPv6 기반으로 수행합니다.

3.3.3 ICMP (Internet Control Message Protocol)

ICMP는 IP와 함께 작동하며, 네트워크 진단이나 오류 메시지를 전달합니다 [49]. 유명한 예가 **ping** 명령으로 사용하는 **ICMP Echo Request/Reply** 입니다. 핑을 보내면 대상이 Echo Reply로 응답해 도달 가능 여부, 왕복 시간 등을 알려줍니다.

ICMP는 또한 다음 같은 메시지를 보냅니다: - 목적지 도달 불가(Destination Unreachable): 라우터가 목적지 네트워크 없음 등 오류 시 보냄. - TTL 초과(Time Exceeded): 패킷 TTL이 0되어 폐기될 때, 또는 IP fragment 재조립 실패 시. - 리다이렉트(Redirect): 더 나은 라우터 경로가 있을 때 송신자에게 경로 수정 권고.

`traceroute` 유틸리티는 의도적으로 TTL을 1부터 차례로 늘려 패킷을 보내고 TTL exceeded 응답(ICMP)으로 경유 라우터의 주소를 알아내는 방식으로 동작합니다.

[실습] IPv4, IPv6, ARP, ICMP 패킷 분석

Wireshark 등을 사용하면 위 프로토콜들의 실제 패킷 예시를 볼 수 있습니다. 간단히 Python의 `scapy` 라이브러리를 사용해 예시 패킷을 만들어보겠습니다 (여기서는 이론적 작성; 인터넷 미접속 환경이므로 패킷 전송은 안 함):

```
# (Pseudo-code, requires scapy)
from scapy.all import Ether, IP, ICMP, sr1
# ICMP Echo Request to 8.8.8.8
packet = Ether(dst="ff:ff:ff:ff:ff:ff")/IP(dst="8.8.8.8")/ICMP()
print(packet.summary())
# -> Ether / IP / ICMP
print(packet[IP].src, "->", packet[IP].dst)
print("ICMP type:", packet[ICMP].type)  # Echo Request type 8
```

이처럼 패킷의 각 계층을 scapy로 조립/분해할 수 있습니다. ARP 패킷 생성도 `ARP(pdst="192.168.1.1")` 등으로 가능하고, IPv6 패킷은 `IPv6(dst="...")`로 생성 가능합니다.

3.4 전송 계층 (Transport Layer)

전송 계층은 호스트 간 (정확히는 프로세스 간) **종단간 통신 서비스**를 제공합니다 [52]. 가장 중요한 프로토콜이 **TCP**와 **UDP**입니다.

3.4.1 포트 (Port)

포트 번호는 한 호스트 내에서 여러 네트워크 프로그램을 구분하기 위한 16비트 숫자입니다 [52]. 예컨대 웹서버는 기본 80번 포트를 사용, SSH 서버는 22번, etc. 클라이언트 측에도 임시 포트가 할당되어, IP주소+포트번호가 합쳐져 **소켓 주소**를 구성합니다. 전송 계층 헤더에는 출발지 포트, 목적지 포트가 있어, **어느 앱으로 보내야 할지**를 결정합니다.

운영체제는 **소켓(socket)** API로 프로세스에게 포트 바인딩, 데이터 송수신 등의 인터페이스를 제공합니다.

[실습] 포트 확인하기

로컬 머신에서 `netstat -an` (Windows) 또는 `ss -tulpn` (Linux) 등의 명령으로 현재 오픈된 포트들을 확인할 수 있습니다. 예: - TCP 0.0.0.0:80 LISTEN (웹 서버가 80 포트 listening) - UDP 0.0.0.0:68 (DHCP 클라이언트 listen) - TCP 192.168.0.5:12345 ESTABLISHED to 13.74.145.**:443 (어떤 애플리케이션이 MS 클라우드와 HTTPS 연결)

이를 통해 **포트가 통신의 종단식별자**로 쓰이는 것을 볼 수 있습니다.

3.4.2 TCP vs UDP

UDP (User Datagram Protocol): 매우 간단한 프로토콜로, 메시지를 **Datagram** 단위로 상대방에게 보냅니다. 신뢰성 보장 없음, 연결 설정 없음, 순서 제어 없음. 그 대신 오버헤드가 적고 지연이 낮습니다 [53]. 주로 **실시간 스트리밍, DNS 조회** 등 손실 허용이나 작은 메시지에 사용.

TCP (Transmission Control Protocol): 신뢰성 있는 바이트 스트림 서비스 제공. 주요 특징: - **연결 지향:** 3-way handshake를 통해 통신 전에 연결 설정 [43]. - **신뢰성:** 데이터 송신 시 **ACK(확인 응답)**을 받아 손실 검출. 손실 시 재전송. - **순서 보장:** 내 보낸 바이트가 순서대로 도착하도록, 중간에 순서 뒤집혀 도착한 것은 재정렬. - **흐름 제어:** 수신측 버퍼 오버플로우 방지를 위해 **윈도우(Window)** 크기만큼만 보내도록 조절. 수신자가 윈도우 크기를 애크에 담아 보냄. - **혼잡 제어:** 네트워크 혼잡 (패킷 유실 증가)시 전송률을 줄이고, 상황 좋아지면 서서히 올리는 알고리즘. (AIMD – Additive Increase Multiplicative Decrease, Slow Start 등)

TCP 헤더에는 출발지/목적지 포트, 시퀀스 번호, 확인 번호, 플래그(SYN, ACK, FIN 등), 윈도우 크기 등 중요한 정보들이 있습니다. TCP는 **전송 제어 블록(소켓 상태)** 을 관리하며, 양 끝단의 상태가 SYN-SENT, ESTABLISHED, FIN-WAIT, TIME-WAIT 등으로 변해갑니다.

[실습] TCP, UDP 패킷 분석

Wireshark로 TCP, UDP 패킷을 살펴보면: - UDP: 헤더에 포트번호와 길이, 체크섬 정도만 있고 데이터 (예: DNS query) 포함. - TCP: 3-way handshake 과정 (SYN, SYN+ACK, ACK) 패킷, 데이터 전송 시 (PSH,ACK 플래그 등), 연결 종료 (FIN/ACK) 등이 보입니다 [43].

`scapy`로 간단히 패킷 조립:

```
from scapy.all import IP, TCP
syn = IP(dst="1.1.1.1")/TCP(dport=80, flags="S", seq=1000)
syn.summary()
# Output: IP / TCP 1.1.1.1:http S
```

이처럼 flags="S"로 SYN 패킷을 만들 수 있고, 시퀀스 번호 등도 지정 가능합니다. 응답을 `sr1(syn)`으로 보내 받아볼 수도 있습니다 (인터넷 연결 시).

3.4.3 TCP 연결과 상태

TCP **3-way handshake**: 클라이언트가 SYN 보내고 (SYN_SENT 상태), 서버는 포트 listening 상태에서 SYN 받아 SYN+ACK 보내 (SYN_RCVD 상태), 클라이언트가 ACK 보내면 (ESTABLISHED), 서버도 ESTABLISHED 되며 쌍방 통신 시작 [43].

연결 종료는 **4-way handshake**: 어느 한쪽이 FIN 보내고, 상대가 ACK (half-close 상태), 상대도 FIN 보내고, 원래 쪽이 ACK 하면 종료. 마지막 ACK 보낸 쪽은 **TIME_WAIT** 상태로 2*MSL (Max Segment Lifetime, 예: 60초) 정도 대기 후 완전히 닫습니다. 이는 늦게 도착한 세그먼트 정리를 위한 시간입니다.

3.4.4 TCP의 신뢰성, 재전송, 혼잡/흐름제어

TCP의 신뢰성은 **ACK 기반**입니다. 보낸 세그먼트에 대한 ACK을 못받으면, 타임아웃 시 재전송합니다. TCP는 전송 시 **라운드 트립 시간(RTT)**을 측정해 타임아웃 시간을 동적으로 설정합니다. 또, 중복 ACK를 세 번 받으면 빠르게 재전송 (Fast Retransmit) 하는 등의 최적화도 있습니다.

혼잡 제어: TCP Tahoe/Reno 알고리즘 등: - **Slow Start:** 처음에는 윈도우(혹은 congestion window)를 작게(예: 1 MSS) 시작, ACK 올 때마다 지수적으로 증가. - **Congestion Avoidance (AIMD):** 혼잡 신호(타임아웃 or 3-dup-ACK) 받으면 윈도우 크기 줄이고(cut to half etc.), 그 이후에는 ACK마다 선형 증가(Additive Increase). - **Fast Recovery:** (Reno) 3중복 ACK 시, 곧바로 실질 혼잡으로 보지 않고 빠른 재전송 후 곧바로 반감 후 additive increase.

흐름 제어: 수신측이 계속 현재 버퍼 여유에 따른 **수신 윈도우 크기**를 ACK에 담아 알려줍니다. 송신측은 그 윈도우를 넘지 않게 보냅니다. 만약 수신 버퍼 꽉 차면 윈도우=0을 광고하며 송신은 일시 멈추고, 수신측이 비우면 윈도우 크기를 늘린 ACK를 다시 보내 재개시킵니다.

이런 메커니즘 덕분에 TCP는 신뢰성과 공정성을 확보하지만, 실시간성이 중요한 애플리케이션엔 지연을 늘릴 수 있는 단점도 있습니다.

3.5 응용 계층 (Application Layer)

응용 계층은 사용자에게 가까운 다양한 네트워크 **응용 프로토콜**들을 포함합니다 [42]. 예로 **DNS, HTTP, FTP, SMTP** 등이 있습니다. 이 파트에서는 특히 **웹 기술** 위주로 다룹니다.

3.5.1 DNS (Domain Name System)

DNS는 인간이 읽기 쉬운 도메인 이름 (예: `www.example.com`)을 컴퓨터가 사용한 IP 주소로 변환해주는 **분산 데이터베이스**이자 프로토콜입니다 [54]. 브라우저가 `http://www.example.com`에 접속할 때, 먼저 DNS 서버에 질의하여 해당 이름의 IP를 얻어와야 합니다.

DNS의 구조: - **계층적 구조:** 루트 네임서버(13개 논리 서버군), 최상위 도메인(TLD) 서버(.com, .net 등), 권한(authoritative) 서버로 단계별 조회합니다. - **재귀/반복 질의:** 로컬 DNS 리졸버가 재귀적으로 루트->TLD->Auth를 거쳐 답을 얻어오고, 클라이언트에 응답. 반복 질의 시 각 단계마다 직접 쿼리하기도.

DNS 레코드에는 A(IPv4), AAAA(IPv6), CNAME(별칭), MX(메일서버) 등 타입이 있습니다. 예: `www.example.com A 93.184.216.34`.

DNS는 UDP 53번 포트를 기본 사용 (신뢰성 크게 문제되지 않아서), 응답이 클 경우 TCP로도 사용.

3.5.2 자원(Resource)과 식별(URI)

웹에서는 **자원(Resource)**이라는 개념으로 콘텐츠를 표현합니다. 자원은 **URL/URI**로 식별됩니다.

- **URI(Uniform Resource Identifier):** 인터넷 자원의 고유 식별자. 흔히 URL (Locator) 형태로 많이 쓰임. 형태: `프로토콜://호스트(:포트)/경로?쿼리#프래그먼트`. 예: `https://www.example.com:8080/path/to/page?uid=100#section2`.
- URL의 구성 요소:

- 프로토콜 (예: http, https, ftp, mailto 등)
- 호스트명 (도메인명 혹은 IP)
- 포트 (생략 시 기본 포트: http=80, https=443 등)
- 경로 (리소스의 경로, 디렉토리와 파일명 비슷)
- 쿼리 파라미터 (동적 페이지 등에 키=값 형태로 전달되는 인자)
- 프래그먼트 (문서 내 특정 위치 지정용, 서버에는 보내지지 않음)

URI는 전세계적으로 고유해야 하고, 이 URI를 통해 웹 클라이언트가 요청을 보냅니다.

3.5.3 웹 서버와 웹 어플리케이션 서버

웹 서버(Web Server)는 HTTP를 통해 정적 콘텐츠(HTML, CSS, 이미지 등)를 제공하는 서버 소프트웨어입니다 (예: Apache httpd, Nginx). **WAS(Web Application Server)**는 동적인 서버-side 로직 (예: Java EE 서버, Python Flask/Django 등)을 수행하여 결과를 생성하는 서버를 일컫습니다. 오늘날 웹 서버와 WAS의 경계는 모호해져서, 둘 다 HTTP를 말단에서 대화하지만, 구조적으로 Apache+PHP, or Nginx(정적)+Tomcat(동적) 연계 등 많이 사용되었습니다.

정리하면: - 웹 서버: 주로 정적 컨텐츠 서비스, 요청에 대해 파일 시스템의 파일을 그대로 응답. - WAS: 비즈니스 로직 처리, 데이터베이스 연동 등 프로그램 실행 결과를 응답.

한 요청 처리에 웹서버와 WAS가 함께 동작하는 경우도 있고, Node.js 같이 하나의 서버가 둘 다 하는 경우도 있습니다.

3.5.4 HTTP의 특징

HTTP (HyperText Transfer Protocol)는 웹의 기반 프로토콜입니다 [55] . 주요 특징: - **텍스트 기반**의 요청/응답 프로토콜 (ASCII로 된 명령과 헤더들). - **무상태(Stateless)**: 기본적으로 각 요청 사이에 서버는 이전 요청의 상태를 유지하지 않습니다 [55] . (→ 따라서 매 요청시 인증정보 등을 함께 보내야 함. 상태 유지하려면 세션 or 쿠키 사용.) - **Request-Response 모델:** 클라이언트가 요청 보내면 서버가 응답. - 주로 TCP 80(HTTP) 또는 443(HTTPS) 포트를 사용. HTTPS는 HTTP를 TLS 암호화한 것.

HTTP/1.1까지는 한 요청에 한 TCP 연결을 사용하는 경향이었지만, Keep-Alive로 여러 요청 지속도 가능. HTTP/2부터는 하나의 TCP 연결에 다중 스트림 요청 허용 (성능 개선).

HTTP 메소드: GET(자원 조회), POST(리소스 생성/변경), PUT, DELETE, HEAD, OPTIONS 등. Status Code: 200 OK, 404 Not Found, 500 Server Error 등으로 응답 상태 전달.

3.5.5 HTTP 메시지 구조

HTTP 요청 메시지 구성 [55] :

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 ...
Accept: text/html

(no body for GET)
```

- 첫 라인: 요청 메소드, 경로, HTTP버전. - 그 다음 여러 헤더들이 `Header-Name: value` 형태로. - 빈 줄 후 (본문이 있으면) 본문.

HTTP 응답 메시지 구성:

```
HTTP/1.1 200 OK
Date: Tue, 25 Jul 2025 10:00:00 GMT
Content-Type: text/html
Content-Length: 125

<html> ... (HTML content) ... </html>
```

- 첫 라인: HTTP버전, 상태코드 숫자와 설명. - 헤더들: 서버 정보, 콘텐츠 타입/길이, 쿠키 등. - 빈 줄 후 응답 본문(HTML이나 JSON 등).

[실습] HTTP 요청-응답 직접 확인하기

telnet이나 netcat으로 HTTP 요청을 수동으로 보내볼 수 있습니다. 예, 텔넷으로 `example.com` 80 포트 접속 후:

```
GET / HTTP/1.1
Host: example.com
```

(빈 줄 보내기) 그러면 HTML 응답이 텍스트로 쭉 내려옵니다. 이를 통해 HTTP 헤더와 바디를 직접 확인할 수 있습니다.

Python으로도 해보겠습니다 (소켓 사용):

```
import socket
host = 'example.com'
port = 80
request = b"GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n"
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
s.sendall(request)
response = b""
while True:
    data = s.recv(4096)
    if not data:
        break
    response += data
s.close()
print(response.split(b"\r\n\r\n")[0].decode())  # 헤더 부분 출력
```

실행하면 HTTP 응답의 헤더 부분이 출력될 것입니다, 예:

```
HTTP/1.1 200 OK
Age: 12345
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Fri, 25 Jul 2025 10:00:00 GMT
Etag: "3147526947"
Expires: Fri, 01 Aug 2025 10:00:00 GMT
...
```

헤더 끝 이후 한 줄 비고, 그 다음부터 HTML 본문이 나옵니다.

3.5.6 HTTP 헤더와 기능

HTTP 헤더들은 요청/응답에 부가 정보를 담습니다: - **일반 헤더:** Date, Connection 등. - **요청 헤더:** Host, User-Agent, Accept (클라이언트가 원하는 MIME 타입), Authorization 등. - **응답 헤더:** Server, Set-Cookie (쿠키 설정), Location (리다이렉션 목적지) 등. - **본문 관련 헤더:** Content-Type, Content-Length, Transfer-Encoding 등.

예를 들어 **쿠키(Cookie)**는 서버가 Set-Cookie 헤더로 클라이언트에 key=value를 저장하도록 하고, 이후 요청마다 Cookie 헤더로 보내게 하여 세션 유지 등에 활용합니다. 쿠키는 **도메인/경로/만료** 속성으로 범위 제어하며, HttpOnly, Secure 등의 플래그도 있습니다.

캐시(Cache) 관련 헤더: - 서버 측 `Cache-Control: max-age=SECONDS`로 클라이언트/프락시에게 얼마동안 캐싱 가능 지시. - `ETag`(엔티티 태그)와 `Last-Modified`를 사용하여 조건부 요청 가능: 클라이언트가 `If-None-Match`에 ETag 보내면 바뀌지 않았으면 304 Not Modified 응답. - Freshness와 Validation, 그리고 프락시 캐시 등 HTTP 캐싱은 웹 성능에 매우 중요합니다.

콘텐츠 협상(Content Negotiation): 클라이언트가 `Accept: ...` 헤더로 원하는 미디어 타입이나 언어(`Accept-Language`), 인코딩(`Accept-Encoding: gzip`) 등을 보낼 수 있습니다 [56]. 서버는 그에 맞춰 가장 적절한 표현으로 응답합니다. 예를 들어 동일한 리소스에 대해 영어/한국어 버전을 제공한다면, 브라우저의 Accept-Language를 보고 언어판 선택.

3.5.7 로드 밸런싱 (부하 분산)

로드 밸런싱은 다수의 서버에 트래픽을 분산하여 처리량을 높이고 가용성을 높이는 기술입니다 [57]. 웹 서비스에서 흔히 **로드 밸런서**(L4/L7 스위치나 HAProxy, Nginx 등)가 클라이언트와 서버들 사이에 위치해 요청을 골고루 여러 서버로 전달합니다.

로드 밸런싱 방식: - **DNS 라운드 로빈:** 한 도메인 이름에 여러 IP (서버) 등록하여, DNS가 돌아가며 IP 제공. 단 순한 무작위 분산이고 health 체크 어려움. - **L4 로드밸런서:** 전송계층 (TCP/UDP) 수준에서 커넥션을 여러 서버로 포워딩. 클라이언트는 LB IP로 접속하지만 LB가 패킷을 내부 서버로 전달. - **L7 로드밸런서/프록시:** 애플리케이션 레벨(HTTP)에서 요청을 받아 콘텐츠에 따라 또는 라운드로빈으로 다른 서버에 보내고, 응답을 클라이언트에 전달. 이 경우 LB가 클라이언트와 두개의 TCP 연결 (클-LB, LB-서버)을 관리.

세션 지속성(sticky session): 사용자 세션이 있는 웹 앱의 경우 동일 사용자의 요청을 같은 서버로 보내야 할 수 있습니다 (예: 로컬 메모리 세션 상태). 쿠키나 IP 기반으로 분배키를 삼아 같은 서버에 라우팅하는 방식을 씁니다.

헬스 체크: 로드밸런서는 주기적으로 백엔드 서버들의 상태를 확인하여, 죽은 서버에는 트래픽을 안 보내게 합니다.

[실습] Nginx로 구현하는 로드 밸런싱

Nginx 웹 서버는 reverse proxy로 동작하며, 백엔드 여러 서버로의 로드밸런싱을 지원합니다. 예컨대 Nginx 설정:

```
http {
    upstream backend_pool {
        server backend1.example.com weight=5;
        server backend2.example.com;
    }
    server {
        listen 80;
        location / {
            proxy_pass http://backend_pool;
        }
    }
}
```

이렇게 하면 frontend Nginx가 80포트에서 요청 받아 upstream 그룹의 서버들로 분산 전달합니다 (weight=5로 backend1은 5/6 트래픽 받음, backend2는 1/6 등). Nginx는 round-robin 기본, ip_hash 등도 설정 가능.

이와 비슷하게 HAProxy 등도 설정 파일로 다수 서버 대상 LB를 수행할 수 있습니다.

로드밸런싱 덕분에 서비스는 **스케일 아웃**(scale-out) 가능해지고, 어느 한 대 장애 시도 다른 서버들이 트래픽 처리해 **고가용성**을 유지할 수 있습니다.

이상으로 네트워크 파트도 마무리되었습니다. 네트워크는 내용이 방대하지만, 핵심을 요약하면 **계층적 프로토콜 구조와 주요 프로토콜들의 동작 원리**입니다. 실제 패킷 캡처나 소켓 프로그래밍 실습을 통해 이해를 깊게 할 수 있습니다.

Part 4. 시스템 프로그래밍

시스템 프로그래밍 파트에서는 리눅스 환경을 중심으로 **운영체제의 인터페이스와 시스템 호출을 활용한 프로그래밍 기법**을 익힙니다. 주로 C 언어로 파일 I/O, 프로세스/스레드, IPC(Inter-Process Communication), 소켓 등에 접근하는 방법을 다루지만, 여기서는 Python으로 가능한 범위 내에서 실습 겸 개념을 설명하겠습니다.

4.1 시스템 프로그래밍의 개념

시스템 프로그래밍이란 운영체제 수준의 기능을 활용하는 프로그래밍, 즉 파일 디스크립터, 프로세스 제어, 메모리 관리, 소켓 등 **하드웨어 및 OS 자원과 직접 상호작용하는 코드 작성**을 말합니다[58]. C, C++, Rust 등이 주 언어이며, Python도 `os`, `ctypes` 등을 통해 제한적으로 가능하긴 합니다.

준비 작업으로, 시스템 프로그래밍을 할 때는 Linux 같은 환경에서 **vim/vi** 같은 편집기 사용, `gcc` 컴파일러, `gdb` 디버거 등을 접하게 됩니다 [59]. 이 강의에선 vi 편집기 사용법도 간단히 다루고 있지만, 여기서는 생략합니다.

4.2 파일 다루기 (파일 I/O 기초)

4.2.1 파일 디스크립터와 파일 포인터

운영체제에서 **파일 디스크립터(file descriptor)**는 앞서 잠깐 언급했듯이, **파일을 식별하는 작은 정수**입니다 [60]. 프로세스가 `open()` 시스템콜로 파일을 열면, OS는 파일 테이블에서 엔트리를 만들고 그 index(정수)를 반환합니다 [61] [27]. 이 정수를 통해 `read(fd, ...)`, `write(fd, ...)` 등의 저수준 I/O를 수행합니다.

한편 C 표준 라이브러리의 `FILE *` (파일 포인터)는 fopen()으로 얻으며, 내부에 버퍼 등을 가진 **고수준 I/O 스트림**입니다 [62]. `fread(), fprintf()` 같은 함수들을 통해 편의 기능을 제공합니다. 이들은 내부적으로 OS 파일 디스크립터를 사용하지만, 추가 버퍼링과 포맷팅을 제공한다는 차이가 있습니다 [63] [64].

정리하면: - **파일 디스크립터 (int)**: OS 커널 수준 핸들, `open/read/write/close`와 함께 사용 [61] [27]. - **파일 포인터 (FILE *)**: C 라이브러리 객체, `fopen/fprintf/fclose`와 함께 사용 [62]. - 둘 다 파일을 다루지만, 전자는 로우레벨, 후자는 하이레벨. 필요 시 파일 포인터로부터 `fileno(fp)` 함수로 FD를 얻을 수 있습니다.

Python에서는 파일을 `open()`하면 파이썬의 file object가 반환되는데, 이건 C의 FILE*에 대응합니다. `fileobj.fileno()` 메서드로 해당 OS 파일 디스크립터 (int)를 구할 수 있습니다.

4.2.2 파일 입력 (읽기) / 출력 (쓰기)

저수준 POSIX API: - `open(path, flags, mode)` -> fd 반환. flags엔 O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_TRUNC 등 조합, mode는 생성 시 퍼미션 (예 0o644). - `read(fd, buffer, count)` -> 실제 읽은 바이트 수 반환, buffer에 데이터 채움. - `write(fd, buffer, count)` -> 보낸 바이트 수 반환 (파일 끝이면 추가). - `lseek(fd, offset, whence)` -> 파일 오프셋 이동 (SEEK_SET/CUR/END). - `close(fd)` -> 파일 닫기, fd 테이블 엔트리 반환.

Python에서는:

```
f = open("test.txt", "w")
f.write("Hello\n")
f.close()
```

이게 내부적으로 open->write->close 시스템콜로 이어집니다. `os.open` (저수준) 함수도 있어, flags를 지정해 fd를 직접 열 수 있습니다.

실습 예 (Python):

```
import os
# 저수준 파일 열기
fd = os.open("hello.txt", os.O_WRONLY | os.O_CREAT | os.O_TRUNC, 0o644)
```

```
# 바이트 쓰기
os.write(fd, b"Hello World\n")
os.lseek(fd, 0, os.SEEK_SET)   # 파일 오프셋을 처음으로 이동
os.write(fd, b"Overwrite")     # 처음부터 덮어씀
os.close(fd)
# 결과 파일 읽어 확인
with open("hello.txt", "r") as f:
    print(f.read())
```

여기서는 파일 처음부터 "Overwrite"가 써져서 결과 파일 내용이 "Overwriteld\n" 같이 될 수 있습니다.

4.2.3 디렉터리 다루기

디렉터리는 파일의 일종이지만, 일반적인 읽기/쓰기 대신 OS 제공 함수를 통해 항목 나열을 합니다. Python에서는 `os.listdir(path)` 등을 사용하거나, `os.scandir` 로 얻은 iterator에서 엔트리를 볼 수 있습니다. C에서는 `opendir`, `readdir`, `closedir` 함수를 사용합니다.

실습:

```
os.makedirs("testdir/subdir", exist_ok=True)
print(os.listdir("testdir"))
# Create some files
open("testdir/file1.txt", "w").close()
open("testdir/file2.txt", "w").close()
# List again
for entry in os.scandir("testdir"):
    print("Name:", entry.name, "IsFile?", entry.is_file(), "IsDir?",
entry.is_dir())
```

이러한 함수들은 내부적으로 `openat` 또는 `getdents` 등의 시스템콜을 사용해 디렉터리 엔트리를 읽습니다.

4.2.4 하드 링크와 심볼릭 링크

하드 링크는 **동일한 파일 내용을 가리키는 디렉터리 엔트리** 입니다. 유닉스 파일 시스템에서는 파일 내용은 inode로 존재하고, 각 파일명은 inode의 링크일 뿐입니다. `ln target linkname` 명령으로 하드 링크 생성하면, target의 inode 번호를 가리키는 새 이름이 생깁니다. 같은 inode를 참조하므로 **링크 수**(inode에 저장된 count)가 증가하고, 둘 중 하나 지워도 다른 하나 남아있으면 실제 데이터는 유지됩니다.

심볼릭 링크(soft link)는 별도 파일 타입으로, 안에 **경로 문자열**을 저장합니다. 그래서 접근 시 OS가 그 문자열을 읽어 **원래 파일로 경로 치환**하여 엑세스합니다. 심링크는 Windows의 "shortcut"과 비슷하지만 OS 레벨에서 처리됩니다.

Python에서:

```
os.link("testdir/file1.txt", "testdir/hardlink_to_file1.txt")  # create hard
link
os.symlink("testdir/file1.txt", "testdir/symlink_to_file1.txt")
```

이렇게 만들면, `hardlink_to_file1.txt`와 `file1.txt`는 같은 inode (os.stat에서 st_ino 값 확인)이고, symlink는 stat하면 다른 inode지만 `os.path.islink`로 확인 가능, `os.readlink`로 대상 읽을 수 있습니다.

(주의: Windows에서 os.link, os.symlink 동작은 제약이 있습니다. Linux/macOS에서는 OK. 여기서는 Linux로 간주.)

4.2.5 파일 속성 다루기 (권한, 시간 등)

파일에는 메타데이터 속성들이 있습니다. `os.stat()` 호출로 `stat_result`를 받아,: - st_mode: 파일 유형과 퍼미션 비트 - st_size: 크기 (바이트) - st_atime, st_mtime, st_ctime: 마지막 접근, 수정, 상태변경 시간 (Unix epoch format) - st_uid, st_gid: 소유자 user/group ID - st_ino, st_dev: inode 번호와 장치 ID - st_nlink: 하드 링크 수

예시:

```
import stat
info = os.stat("testdir/file1.txt")
print(oct(info.st_mode), "size:", info.st_size, "bytes")
print("Last modified:", info.st_mtime)
print("Is regular file?", stat.S_ISREG(info.st_mode))
print("Permissions:", oct(stat.S_IMODE(info.st_mode)))
```

또 os.chmod(path, mode)로 권한 변경, os.utime(path, times=(atime, mtime))으로 타임스탬프 수정, os.chown(path, uid, gid)로 소유자 변경 (관리자 권한 필요) 등이 가능.

이러한 작업은 시스템콜 `chmod`, `utime`, `chown` 등에 대응됩니다.

4.2.6 파일 메모리 매핑 (mmap)

Memory Mapped File은 파일 내용을 메모리에 매핑하여, 파일 I/O를 시스템콜 없이 메모리 읽기/쓰기처럼 수행하는 기법입니다 [65]. 즉 `mmap()` 시스템콜로 가상주소 공간의 한 영역을 파일의 한 부분과 연결하면, 그 메모리에 대한 접근이 곧 파일 I/O로 이어집니다. 이점은 **페이지 단위 지연 로딩**(필요한 부분만 읽음)과 **시스템콜 오버헤드 감소**입니다 [38].

Python의 `mmap` 모듈을 통해 사용 가능:

```
import mmap
# 파일 열고 매핑
fd = os.open("testdir/file1.txt", os.O_RDWR)
mm = mmap.mmap(fd, 0)  # whole file mapped to memory
# 읽기
print(mm[:5])          # read first 5 bytes
```

```
# 쓰기
mm[0:5] = b"HELLO"    # modify first 5 bytes
mm.flush()            # ensure changes written to file
mm.close()
os.close(fd)
```

이렇게 하면 5바이트 바꾸는 동안 별도 write 시스템콜 없이 메모리에 쓴 것이 OS에 의해 lazy하게 파일에 반영됩니다. (mmap flush 또는 close 시 write-back, 또는 자동 write-back 가능)

메모리 맵은 큰 파일을 다룰 때 유용하며, OS의 페이지 캐시를 직접 활용하므로 성능 이점이 있습니다 [66]. 다만 전체 크기의 주소 공간 필요, 그리고 랜덤 접근 패턴에 따라 성능 효과가 다를 수 있습니다.

4.3 프로세스와 스레드 다루기

4.3.1 프로세스 다루기 (생성/종료 등)

C언어로는 `fork()`로 프로세스 복제를 하고, `exec()`로 새 프로그램 실행, `exit()`로 종료, 부모는 `wait()`로 자식 종료 대기하는 등의 흐름입니다. Python에서는 `os.fork()` (Unix only)로 프로세스를 쪼개거나, 보다 portable하게 `multiprocessing` 모듈을 사용합니다. 또한 새로운 프로세럼 수행은 `os.execv` 등을 제공하지만, Python에서는 `subprocess` 모듈이 주로 쓰입니다.

예: Python `subprocess`:

```
import subprocess
result = subprocess.run(["ls", "-l", "testdir"], capture_output=True, text=True)
print("Return code:", result.returncode)
print("Output:", result.stdout)
```

이는 내부적으로 fork -> exec("/bin/ls") -> 부모가 자식 종료 wait 과정을 처리한 것입니다.

`multiprocessing`:

```
from multiprocessing import Process
def worker(n):
    print("Worker", n, "PID:", os.getpid())
p = Process(target=worker, args=(1,))
p.start()
p.join()
```

multiprocessing on Linux uses fork by default to create a new process running `worker` function.

`os.getpid()`와 `os.getppid()`로 현재 프로세스 ID와 부모 프로세스 ID를 알 수 있습니다. Windows에서는 fork가 없고 spawn방식만 있습니다.

4.3.2 스레드 다루기

Python에서 스레드는 `threading` 모듈:

```
import threading
def run():
    print("Thread", threading.get_ident(), "PID", os.getpid())
threads = []
for i in range(3):
    t = threading.Thread(target=run)
    t.start()
    threads.append(t)
for t in threads:
    t.join()
```

세 개의 스레드가 생성되어 동시에 실행합니다 (CPython GIL로 엄밀히 동시에는 아니지만, I/O 없는 경우). `threading.get_ident()`는 스레드ID (OS 스레드 ID와 다를 수), `os.getpid()`는 동일 PID (스레드는 같은 프로세스 내).

C에서는 `pthread_create`, `pthread_join` 등 pthreads 라이브러리 호출로 생성/정지하고, `pthread_mutex` 같은 동기화 도구를 씁니다. Python GIL 제약으로 CPU 병렬은 안되지만, I/O 병렬엔 유용하거나, PyPy/Jython like GIL없으면 가능.

4.3.3 뮤텍스와 세마포 (동기화)

운영체제 파트에서 개념 다룬 뮤텍스/세마포를 실제 프로그래밍에서는 pthread_mutex_lock/unlock, sem_wait/post 등으로 쓰고, Python에서는 `threading.Lock`, `threading.Semaphore` 사용합니다.

예, 은행 계좌를 여러 스레드가 업데이트하는 상황:

```
lock = threading.Lock()
balance = 0
def deposit():
    global balance
    for i in range(100000):
        lock.acquire()
        balance += 1
        lock.release()

# 두 스레드 deposit
t1 = threading.Thread(target=deposit)
t2 = threading.Thread(target=deposit)
t1.start(); t2.start()
```

```
t1.join(); t2.join()
print("Final balance:", balance)
```

락 없이 하면 race condition으로 balance < 200000 나올 확률 높음. 락으로 보호하면 정확히 200000 나옵니다.

세마포어 (`threading.Semaphore`)은 공유 자원 count 개수 제한 등에 사용. 예:

```
sem = threading.Semaphore(3)  # 최대 3개 동시 접근 가능
def task():
    sem.acquire()
    # critical section (at most 3 threads here)
    print("Accessing resource by", threading.get_ident())
    time.sleep(1)
    sem.release()
```

동시에 5개 스레드 task() 호출해도, 세마포 3 덕에 3개씩만 동시 실행됨.

4.3.4 공유 메모리 기반 IPC

프로세스 간의 메모리 공유는 `mmap`이나 `shm_open`/`mmap` 조합으로 구현할 수 있습니다. Python `multiprocessing` 모듈은 `multiprocessing.Value` 또는 `Array` 등을 통해 공유 메모리를 활용합니다.

예:

```
from multiprocessing import Process, Value
counter = Value('i', 0)  # shared integer
def increment(n):
    for i in range(n):
        with counter.get_lock():  # lock acquired for safe update
            counter.value += 1
p1 = Process(target=increment, args=(1000000,))
p2 = Process(target=increment, args=(1000000,))
p1.start(); p2.start()
p1.join(); p2.join()
print(counter.value)  # expected 2000000
```

여기 Value('i', 0)는 공유 메모리에 정수 0을 생성. (Lock은 자동 포함, get_lock() to use). 두 프로세스가 이 값을 같이 증가.

C에서는 `shm_open("/shmname", O_CREAT|O_RDWR, 0666)`으로 POSIX 공유 메모리 객체 열고 `mmap`하여 사용하거나, `mmap` MAP_ANONYMOUS|MAP_SHARED for parent-child.

4.3.5 파이프 기반 IPC

파이프(pipe)는 한 프로세스의 출력이 다른 프로세스의 입력으로 흐르는 **단방향 통신 채널**입니다. 일반 파이프는 부모-자식 관계에서 `pipe()` 시스템콜 (익명 파이프)로 사용됩니다. 명명 파이프 (FIFO)는 `mkfifo`로 경로 생성하여 관계없는 프로세스 간도 가능합니다.

Python에서는 `os.pipe()` -> (read_fd, write_fd) 반환.

```
r, w = os.pipe()
pid = os.fork()
if pid == 0:  # child
    os.close(r)
    os.write(w, b"Hello from child")
    os.close(w)
else:  # parent
    os.close(w)
    data = os.read(r, 100)
    print("Parent got:", data)
    os.close(r)
```

Fork 후, child가 pipe에 쓰고 부모가 읽음.

또 Python `subprocess.Popen`에는 `stdout=subprocess.PIPE` 옵션으로 자식 stdout pipe로 받아 부모가 읽는 것 등을 자동 처리해줍니다.

`multiprocessing.Pipe` provides a high-level pipe for processes:

```
from multiprocessing import Pipe, Process
parent_conn, child_conn = Pipe()
def child_task(conn):
    conn.send("Hi parent")
    print("Child got:", conn.recv())
p = Process(target=child_task, args=(child_conn,))
p.start()
print("Parent got:", parent_conn.recv())
parent_conn.send("Hello child")
p.join()
```

Pipe() returns two connection objects usable for send/recv on each side.

4.3.6 시그널 다루기

시그널(Signal)은 OS에서 프로세스에 비동기 이벤트를 알리는 메커니즘입니다. 예: `SIGINT` (Ctrl+C), `SIGTERM` (terminate request), `SIGCHLD` (자식 종료), `SIGALRM` (timer).

C에서는 `signal(sig, handler)` 또는 `sigaction`으로 처리기 설치, `kill(pid, sig)`로 시그널 보내기 (자신 또는 다른 프로세스). Python `signal` 모듈에서 `signal.signal(signal.SIGINT, handler)` 지정 가능 (메인 thread 처리만).

Python 예:

```
import signal, time, os
def handler(signum, frame):
    print("Signal received:", signum)
signal.signal(signal.SIGALRM, handler)
signal.alarm(2)  # send SIGALRM to self after 2 seconds
print("Waiting for alarm...")
time.sleep(5)
```

2초 후 SIGALRM triggers handler, prints message, and `time.sleep` is interrupted (sleep may raise exception in Python if signal caught? Actually Python delays signal handling to main loop if in system call, but conceptually).

To send a signal to another process:

```
os.kill(other_pid, signal.SIGTERM)
```

This sends SIGTERM to process with pid=other_pid (if allowed). Also pressing Ctrl+C sends SIGINT to the process group.

Signal handling intricacies: - Some signals like SIGKILL, SIGSTOP cannot be caught or ignored. - After fork, handlers are inherited, but in multi-thread context, only main thread receives signals (in Python). - Python's GIL means signals are only handled between bytecode instructions, so long C extensions might delay handling.

Signals are often used for graceful termination, child process reap (SIGCHLD), alarms/timeouts etc.

4.4 소켓 프로그래밍 (네트워크 프로그래밍 기초)

소켓(Socket)은 **네트워크 통신의 종착점**을 나타내는 프로그래밍 인터페이스입니다. 프로세스 간 (특히 다른 호스트 간) 통신을 가능케 합니다. 전송 계층의 TCP/UDP에 대응되고, IP/포트까지 포함한 식별자를 다룹니다.

4.4.1 소켓이란

소켓은 사실상 **파일 디스크립터**의 특별한 경우로, `socket()` 시스템콜로 생성됩니다. `socket(domain, type, protocol)`: - domain: `AF_INET` (IPv4), `AF_INET6`, `AF_UNIX` (UNIX domain) - type: `SOCK_STREAM` (TCP), `SOCK_DGRAM` (UDP), ... - protocol: 0 (default for given type, e.g. IPPROTO_TCP or IPPROTO_UDP)

그 후 `bind(fd, address)`로 주소/포트 할당 (서버측), `listen(fd, backlog)`로 들어오는 연결 청취 (TCP only), `accept(fd)`로 연결 수락 -> 새 fd 반환 (TCP). 클라이언트 측은 `connect(fd, server_address)`로 연결 시도.

UDP 소켓은 `sendto`, `recvfrom`를 사용 (연결 불필요, `connect` optional to set default peer).

4.4.2 TCP 소켓 다루기 (서버/클라이언트)

Python socket example (TCP): Server:

```
import socket
server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_sock.bind(('0.0.0.0', 5000))
server_sock.listen()
print("Server listening on port 5000...")
conn, addr = server_sock.accept()
print("Accepted connection from", addr)
data = conn.recv(1024)
print("Received:", data.decode())
conn.sendall(b"Hi Client")
conn.close()
server_sock.close()
```

Client:

```
client_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_sock.connect(('127.0.0.1', 5000))
client_sock.sendall(b"Hello Server")
reply = client_sock.recv(1024)
print("Server replied:", reply.decode())
client_sock.close()
```

If run server (in one process) and client (in another or after server listening), the output shows handshake success: Server prints "Accepted connection from ('127.0.0.1', some_port)" and Received "Hello Server", Client prints "Server replied: Hi Client".

This demonstrates basic TCP connection setup and data exchange. Under the hood: - Server `listen` backlog means OS will queue new connections. - `accept` returns a new socket for the connection (server_sock remains listening). - Data send/recv correspond to TCP stream data. The OS handles segmentation, ACKs, etc.

4.4.3 TCP 소켓 프로그래밍 (응용)

A more interactive server might handle multiple clients, maybe using threading or `select`/`poll`:

```
# Multi-client echo server with threads
import threading
def handle_client(conn, addr):
    print("Connected:", addr)
    with conn:
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)  # echo back
    print("Disconnected:", addr)

server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_sock.bind(('0.0.0.0', 5001))
server_sock.listen()
while True:
    conn, addr = server_sock.accept()
    threading.Thread(target=handle_client, args=(conn, addr)).start()
```

This spawns a new thread per client to echo received data. Python threads can handle I/O concurrently since GIL is released during blocking I/O.

4.4.4 UDP 소켓 다루기

UDP example: Server:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('0.0.0.0', 5005))
print("UDP server ready")
data, addr = sock.recvfrom(1024)
print("Received from", addr, data.decode())
sock.sendto(b"Hello UDP Client", addr)
sock.close()
```

Client:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(b"Hi UDP Server", ('127.0.0.1', 5005))
data, addr = sock.recvfrom(1024)
print("Server says:", data.decode())
sock.close()
```

No connection handshake; the server just bind and waits for messages. When a message comes, `recvfrom` provides the sender's address so server knows where to reply.

Because UDP is unreliable, if the reply is lost, client doesn't automatically retry here. So application might implement retries if needed.

4.4.5 소켓 프로그래밍 응용: 간단한 채팅, 파일 전송 등

With sockets, one can implement a variety of protocols: - A simple chat: clients send messages to server, server broadcasts to all other clients. - File transfer: e.g. client requests a file name, server reads file and streams bytes to client (with perhaps some simple protocol like send file size first). - Request/response: basically implementing parts of HTTP (like a trivial web server or client).

Because this is a broad area, the key is understanding blocking vs non-blocking I/O: - `socket.setblocking(False)` to use with `select` or `poll` loops for high-performance or single-thread concurrency.

Also handling partial sends/receives: - `sendall` tries to send entire buffer by looping internally; `send` might send partial (should check return). - `recv` might get less than expected if message bigger than buffer or if stream chunked, so need to accumulate data or have some delimiting in protocol.

Finally, note that in C, after `fork()`, sockets are inherited by child (unless close-on-exec flags). In multi-process server design, one might spawn processes to handle accepted connections (like preforked servers).

이상으로 시스템 프로그래밍 파트도 마무리되었습니다. 이 부분은 직접 코드 실습이 특히 중요한 영역이므로, 위 예제들을 실제로 실행해보고 추가로 C로도 구현해 보는 것을 추천합니다.

Part 5. 데이터베이스

마지막 파트인 데이터베이스에서는 **데이터를 구조화하여 저장하고 질의하는 방법**을 배웁니다. AI 시대에도 데이터 저장/조회는 필수이므로, 관계형 데이터베이스(SQL)와 NoSQL (MongoDB, Redis 등)을 모두 개괄합니다.

5.1 데이터베이스 개요 – 거시적으로 보기

데이터베이스(Database)란 관련성 있는 데이터들의 모음이며, 이를 관리하는 소프트웨어를 **DBMS(Database Management System)**라고 합니다. 전통적으로 많이 쓰이는 관계형 DB (RDBMS: MySQL, PostgreSQL, Oracle, MSSQL 등)는 **SQL**이라는 선언적 쿼리 언어로 데이터를 정의/조회합니다.

AI 시대에는 빅데이터 처리를 위해 NoSQL (문서지향, 키-값, 컬럼 패밀리, 그래프 DB 등)이 활용되기도 하지만, 관계형 DB는 여전히 중요한 기본입니다.

데이터베이스를 사용하는 이유: - 데이터를 **일관성 있게 관리** (동시 접근, 장애 내구성 – 트랜잭션 ACID 특성). - **효율적인 질의**가 가능 (인덱스, 최적화 엔진). - 데이터 **모델링**을 통해 의미 있는 구조로 저장.

이 절에서는 먼저 MySQL 등 SQL DB를 다루고, 후반부에 MongoDB, Redis와 같은 NoSQL 예시를 봅니다.

[실습] MySQL 설치 및 설정

(주: 실제 설치는 환경 따라 다르니 개념 서술)

MySQL(혹은 MariaDB)을 설치했다면: - `mysqld` 서버 프로세스가 백그라운드로 동작 중인지 확인. - `mysql -u root -p` 등의 명령으로 클라이언트 접속. - 접속하여 데이터베이스와 테이블을 생성하고, SQL 질의를 수행.

학습용으로는 MySQL 서버 대신 SQLite를 사용할 수도 있습니다 (가볍게 파일 기반 동작). 여기서는 Python의 sqlite3을 사용하여 SQL 실습을 진행하겠습니다.

5.2 데이터베이스 생성과 관리 (SQL DDL/DML 기본)

5.2.1 기본 키와 외래 키

기본 키(Primary Key)란 테이블에서 각 행(row)을 고유하게 식별해주는 컬럼(또는 컬럼 조합)입니다 [67]. 예를 들어 회원 테이블이라면 회원 ID (회원번호)가 기본 키가 될 수 있습니다. 기본 키에는 동일한 값이 둘 이상 존재할 수 없으며 (유일성), NULL 값도 불허합니다.

외래 키(Foreign Key)는 한 테이블의 컬럼이 다른 테이블의 기본 키를 참조하는 관계를 말합니다 [67]. 예컨대 주문 테이블에 고객 ID가 있는데, 이 고객 ID는 고객 테이블의 기본 키를 참조하므로 외래 키 제약을 둘 수 있습니다. 외래 키 제약을 설정하면 참조 무결성을 DBMS가 강제: 없는 고객 ID를 주문에 넣을 수 없게 하고, 고객이 삭제될 때 연쇄적으로 주문도 삭제하거나 금지할 수 있습니다.

5.2.2 데이터베이스와 테이블 생성 (DDL)

DDL (Data Definition Language): 데이터베이스 객체(스키마, 테이블 등)를 정의/변경하는 SQL 문. - `CREATE DATABASE mydb;` 로 데이터베이스 생성. - `USE mydb;` 로 해당 DB 사용 (MySQL). - `CREATE TABLE ...` 로 테이블 생성 [67]. - `DROP TABLE ...`, `ALTER TABLE ...` 등.

예, 파이썬 sqlite3로 실행:

```
import sqlite3
conn = sqlite3.connect(':memory:')  # 메모리 DB
cur = conn.cursor()
cur.execute("""
CREATE TABLE Users (
    user_id INTEGER PRIMARY KEY,
    name TEXT,
    age INTEGER
);
""")
cur.execute("""
CREATE TABLE Orders (
    order_id INTEGER PRIMARY KEY,
```

```
    user_id INTEGER,
    product TEXT,
    FOREIGN KEY(user_id) REFERENCES Users(user_id)
);
""")
conn.commit()
```

위 SQL에서: - Users 테이블: user_id를 INTEGER PRIMARY KEY로 지정 (SQLite INTEGER PRIMARY KEY는 rowid alias). - Orders 테이블: user_id가 Users(user_id)를 참조하는 외래 키. (SQLite 기본적으로 외래키 검사가 off일 수 있어 `PRAGMA foreign_keys=ON` 해야 동작하지만 논의 목적상).

5.2.3 데이터 입력 – INSERT

INSERT 문으로 테이블에 행을 추가합니다 [68]. - 예: `INSERT INTO Users (name, age) VALUES ('Alice', 30);` - 기본 키 컬럼은 명시 안하면 DB가 자동 할당 (AUTO INCREMENT)하기도. SQLite INTEGER PRIMARY KEY는 자동 증가, MySQL은 AUTO_INCREMENT 속성. - 여러 행을 한 번에 삽입: `INSERT INTO ... VALUES(...), (...), ...;`

실습:

```
users = [("Alice", 30), ("Bob", 24), ("Charlie", 29)]
cur.executemany("INSERT INTO Users (name, age) VALUES (?, ?);", users)
conn.commit()
```

`executemany`로 다중 insert. 또는 loop with `INSERT`.

5.2.4 데이터 조회 – SELECT

SELECT 문은 SQL의 핵심, 데이터를 조회하는 질의문입니다 [68]. 기본 형태:

```
SELECT 컬럼들
FROM 테이블들
WHERE 조건
GROUP BY 그룹화컬럼
HAVING 그룹조건
ORDER BY 정렬컬럼
LIMIT 개수;
```

- **프로젝션**: SELECT 절에 필요한 컬럼만 지정 (또는 * 전체). - **필터링**: WHERE 절로 조건 (예: `age >= 25 AND name LIKE 'A%'`). - **조인(Join)**: FROM 절에 여러 테이블 쓰고, WHERE에 조인 조건 (또는 JOIN..ON 문법)으로 연관된 데이터 결합. - **GROUP BY**: 집계함수 (COUNT, SUM 등)와 함께 사용, 특정 컬럼 값별로 묶어 처리. - **HAVING**: 그룹 결과에 대한 조건 (예: COUNT > 1 등). - **ORDER BY**: 결과 정렬 (ASC/DESC). - **LIMIT/OFFSET**: 결과 레코드 수 제한 (주로 MySQL, PostgreSQL 등, Oracle은 다른 방식).

예시 질의: - 모든 사용자 이름/나이: `SELECT name, age FROM Users;` - 나이 25 이상인 사용자만: `SELECT * FROM Users WHERE age >= 25;` - 특정 사용자 주문 목록:

```
SELECT Orders.order_id, Orders.product
FROM Orders JOIN Users ON Orders.user_id = Users.user_id
WHERE Users.name = 'Alice';
```

- 사용자별 주문 수:

```
SELECT Users.name, COUNT(Orders.order_id) AS num_orders
FROM Users LEFT JOIN Orders ON Users.user_id = Orders.user_id
GROUP BY Users.user_id, Users.name
HAVING num_orders > 0
ORDER BY num_orders DESC;
```

(LEFT JOIN to include users with 0 orders too, then HAVING filters if needed.)

실습 in Python:

```
# Add some orders
orders = [(1, 'Laptop'), (2, 'Mouse'), (1, 'Keyboard')]
# orders list of (user_id, product)
cur.executemany("INSERT INTO Orders (user_id, product) VALUES (?, ?);", orders)
conn.commit()

# Example query: List orders with user names
cur.execute("""
SELECT U.name, O.product
FROM Orders O JOIN Users U ON O.user_id = U.user_id;
""")
for row in cur.fetchall():
    print(row)
```

This should print user name and product for each order (like ('Alice','Laptop'), etc).

5.2.5 데이터 상세 조회 (조건 검색)

WHERE 절의 표현식: - `=, !=, <, >, <=, >=` 비교. - `BETWEEN a AND b` (포함 범위 검사). - `IN (val1, val2, ...)` 집합 포함 여부. - `LIKE 'pattern'` 문자열 패턴 (SQL wildcard % and _). - `IS NULL` 또는 `IS NOT NULL`. - 복합: `AND`, `OR`, `NOT`으로 조합, 괄호로 우선순위.

예: `SELECT * FROM Users WHERE age BETWEEN 25 AND 30 AND name LIKE 'A%';`

5.2.6 데이터 수정 – UPDATE

UPDATE 문으로 기존 행의 컬럼값을 변경합니다:

```
UPDATE Users
SET age = age + 1
WHERE name = 'Bob';
```

이렇게 하면 조건에 맞는 Bob의 age가 1 증가.

주의: WHERE 없이 UPDATE 하면 테이블 모든 행 수정되니, 실수 방지해야.

Python:

```
cur.execute("UPDATE Users SET age = ? WHERE name = ?;", (26, "Bob"))
conn.commit()
```

5.2.7 데이터 삭제 – DELETE

DELETE 문으로 행 삭제:

```
DELETE FROM Orders WHERE order_id = 2;
```

특정 주문 행 삭제. 외래키 제약 있으면, Orders행 삭제시 Users 유지여부에 따라 (no action, cascade, set null etc depending on constraint definition).

`DELETE FROM Users` 만 쓰면 전부 삭제. 주의.

SQLite by default ON DELETE of referenced row causes error if referencing row exists, unless declared with CASCADE or others.

Python:

```
cur.execute("DELETE FROM Orders WHERE product = ?;", ("Mouse",))
conn.commit()
```

5.3 효율적 쿼리 (인덱스, 뷰, 조인, 분할)

5.3.1 인덱스

인덱스(Index)는 테이블 데이터에 대한 **검색용 보조 구조**입니다 [69]. 보통 B-트리나 해시로 구현되어, 특정 컬럼에 대한 빠른 조회를 지원합니다.

예: Users.name 컬럼에 인덱스 생성하면, `WHERE name='Alice'` 조회가 테이블 full scan 없이 인덱스 통해 바로 해당 레코드 찾아갑니다. 기본 키나 유니크 키는 자동 인덱스 생성됨.

인덱스 생성:

```
CREATE INDEX idx_users_name ON Users(name);
```

인덱스는 삽입/삭제 시 유지 비용이 있으므로, 자주 조회되는 컬럼에만 생성하는 게 좋아요.

실습: SQLite query planner 보기:

```
cur.execute("EXPLAIN QUERY PLAN SELECT * FROM Users WHERE name = 'Alice';")
print(cur.fetchall())
```

Without index, likely shows using full scan. After index creation:

```
cur.execute("CREATE INDEX idx_users_name ON Users(name);")
cur.execute("EXPLAIN QUERY PLAN SELECT * FROM Users WHERE name = 'Alice';")
print(cur.fetchall())
```

It should now indicate using index search (like "SEARCH TABLE Users USING INDEX idx_users_name").

5.3.2 뷰 (View)

뷰(View)는 SELECT 쿼리에 이름을 부여해 마치 테이블처럼 사용할 수 있는 가상 테이블입니다 [69]. It doesn't store data by itself (unless materialized view which is storing snapshot), but it simplifies complex queries or provides a layer of abstraction.

예:

```
CREATE VIEW YoungUsers AS
SELECT name, age FROM Users WHERE age < 30;
```

Then `SELECT * FROM YoungUsers;` is effectively that query.

Views can join multiple tables or filter columns (useful for security, e.g. create view that hides sensitive info).

In SQLite:

```
cur.execute("CREATE VIEW IF NOT EXISTS UserOrders AS SELECT U.name, O.product
FROM Users U JOIN Orders O ON U.user_id = O.user_id;")
cur.execute("SELECT * FROM UserOrders;")
print(cur.fetchall())
```

This yields all user name and product combos.

5.3.3 JOIN (다중 테이블 조인)

We've already used JOIN in selects. Key join types: - **INNER JOIN**: only matching rows from both tables (equivalent to listing tables in FROM with matching WHERE). - **LEFT JOIN** (or LEFT OUTER): all rows from left table, plus matching from right or NULL if no match [57] . - **RIGHT JOIN**: opposite of left (not in SQLite, but in others). - **FULL OUTER JOIN**: all rows from both with NULL for missing matches (some DBs support). - **CROSS JOIN**: cartesian product (rarely used directly except to deliberately cross or with join cond in where separate). - **SELF JOIN**: table join with itself (alias needed).

Join conditions usually are foreign key = primary key or similar.

Already example given: join Users and Orders.

A more complex scenario: Suppose we had another table, say Products(product_id, name, price), and Orders refers to product_id instead of product name text. Then join across three tables possible:

```
SELECT U.name AS user, P.name AS product
FROM Orders O
JOIN Users U ON O.user_id = U.user_id
JOIN Products P ON O.product_id = P.product_id;
```

Indices help here: index on Orders.user_id and Orders.product_id would speed up join (or respectively on referenced tables PK which typically are indexed).

5.3.4 데이터베이스 분할과 샤딩

데이터베이스 분할(Partitioning): 하나의 큰 테이블을 **행단위로 쪼개서 여러 부분**으로 나누어 저장하는 것 [69] . 같은 DB 인스턴스 내에서 논리적 분할이거나, or partition feature where certain rows go to certain files/disk groups.

Partitioning can be: - Horizontal partitioning: by row (range partition on date, list partition by region, etc) – each partition is same schema subset of rows. - Vertical partitioning: by columns (rare as separate tables needed). - Purpose: manage huge tables for performance (scanning smaller partition), archiving old data separate, etc.

샤딩(Sharding): Partitioning the data **across multiple DB servers(instances)** [70] . Each shard is a separate DB (with its own CPU/memory/disk). For example by customer region or ID modulus etc, to distribute load and data size across servers [71] . Requires application or middleware logic to route queries to correct shard (like user id 1-1000 on db1, 1001-2000 on db2 etc).

Difference: Partition could be on same server (no added throughput, but manageability). Sharding is across servers -> true scalability with parallel capacity [72] .

Sharding often requires sacrificing joins across shards (or doing at app level) and careful key design (shard key selection is critical) [73] .

5.3.5 NoSQL 소개: MongoDB와 샤딩, Redis 등

NoSQL (Not only SQL) refers to non-relational databases often used for big data, flexibility, or performance scaling reasons. Several types: - **Document DB** (e.g., MongoDB, CouchDB): store semi-structured data (JSON-like docs) with flexible schema. - **Key-Value stores** (e.g., Redis, Riak): simplest get/set by key, maybe with expiration (good for cache). - **Column Family stores** (e.g., Cassandra, HBase): distributed wide-column stores for huge scale (like storing billions of rows across cluster). - **Graph DB** (e.g., Neo4j): specialized for graph (nodes and edges with properties).

5.3.5.1 MongoDB (문서 지향 DB)

MongoDB stores JSON documents in collections. It's schema-less in that not all docs need same fields, though schema validation optional [74] . It uses BSON (binary JSON) internally, supports rich data types and nested documents/arrays.

Basic usage: - Document example:

```
{ "_id": ObjectId("..."), "name": "Alice", "skills": ["Python","SQL"], "age":
30 }
```

This could be in collection "users". We can query with fields easily due to flexible schema.

Mongo queries typically done via drivers or shell:

```
db.users.find({age: {$gte: 25}, skills: "SQL"})
```

Would find users aged >=25 that have "SQL" in their skills array.

Indexing in Mongo is similar in concept (B-tree on fields). Sharding: MongoDB supports auto-sharding across cluster by a shard key (must choose a field, e.g., user_id). It then distributes data and queries across shards. The app sees one logical DB, but data is on multiple nodes.

We cannot run Mongo here (needs server). But we can conceptualize: Mongo is often used for web apps needing flexibility or storing nested data conveniently (since RDB can require many tables for complex JSON).

Example pseudo:

```
# Using pymongo
from pymongo import MongoClient
client = MongoClient("mongodb://localhost:27017/")
db = client["testdb"]
users_coll = db["users"]
users_coll.insert_one({"name": "Dave", "age": 23, "skills": ["C++","Linux"]})
res = users_coll.find({"age": {"$lt": 25}})
for user in res:
    print(user)
```

Should show Dave.

5.3.5.2 Redis (Key-Value in-memory DB)

Redis is an **in-memory key-value store** often used for caching, real-time data, pub/sub messaging [75]. It keeps data in RAM for speed and periodically writes to disk for persistence (snapshot or AOF log).

It supports data structures: strings, hashes, lists, sets, sorted sets, bitmaps, etc. And atomic operations on them.

Use cases: caching database query results, session store, counters, leaderboards etc.

Basic usage (assuming redis server running, using `redis` py library or command-line):

```
SET user:1000:name "Alice"
GET user:1000:name          -> "Alice"
INCR page_view_count        -> increments an integer
LPUSH queue:tasks "task1"   -> push to list
BRPOP queue:tasks 0         -> blocking pop from list (for work queue)
```

Redis commands are straightforward text commands. It is often used in code through client libraries:

```
import redis
r = redis.Redis()
r.set("msg", "Hello")
print(r.get("msg"))  # b'Hello'
```

Redis typically runs on one machine (replication optional for HA, cluster mode available for sharding by key hash slots).

Redis pub/sub: one can SUBSCRIBE to a channel and others PUBLISH messages to it, enabling simple messaging patterns (though not durable if nobody listening, messages lost).

NoSQL vs SQL: - Schema flexibility (NoSQL dynamic schemas, SQL fixed schemas [76]). - Scalability: Many NoSQL systems ease horizontal scaling (sharding built-in), whereas traditional RDBMS scaling vertically or via complex partitioning. - Query: SQL is very powerful for relational queries (JOINs, aggregate). NoSQL often simpler queries (key-based or map-reduce). - Transactions: SQL DBs usually fully ACID; some NoSQL forego strong ACID for performance (though newer ones like Mongo add multi-doc transactions with less performance). - Use case: If relationships and structure are clear, RDB is stable choice. If data structure evolves or is unstructured or extremely large scale, NoSQL might fit.

5.4 정리 및 마무리

데이터베이스 파트에서는 관계형 DB의 핵심인 **SQL**(DDL, DML)과 **스키마 설계**(키, 제약), 그리고 **성능개선 기법**(인덱스)을 익혔습니다. 또한, 현대 애플리케이션에서 각광받는 **NoSQL** DB의 개념과 대표 사례인 **MongoDB**(문서지향), **Redis**(키-값)를 살펴봤습니다.

실습을 위해 SQLite를 사용하여 SQL을 실행해봤으며, MongoDB/Redis는 간략한 코드 예제로 개념만 다루었습니다. 실제 업무 환경에서는 MySQL/PostgreSQL 등의 DB를 설치해 SQL을 익혀보고, 필요에 따라 MongoDB, Redis 등을 직접 사용해보는 것이 좋습니다.

마지막으로 부록으로 강의에서 제공하는 **기술면접 TOP 50** 문제가 언급되어 있는데, 이는 다양한 CS 분야에서 50개의 중요한 질문을 뽑은 것 같습니다. (예: 운영체제의 가상메모리란?, TCP와 UDP 차이는?, DB에서 인덱스란? 등) 이러한 Q&A는 면접 대비 뿐 아니라 학습 복습용으로 유용하므로, 스스로 답을 생각해보고 정리해보는 것을 권장합니다.

以上으로 **"강민철의 인공지능 시대 필수 컴퓨터 공학 지식"** 5개 주요 파트 내용을 모두 다뤘습니다. 각 분야(컴퓨터구조, 운영체제, 네트워크, 시스템 프로그래밍, 데이터베이스)의 기본 개념부터 실습 예제까지 폭넓게 소개하였으니, 제공된 self-study 자료를 통해 혼자서도 충분히 개념을 익히고 응용해보실 수 있을 것입니다.

학습한 내용들을 바탕으로 더 깊이있는 주제 (예: 고급 CPU 구조나 최신 OS 기능, 네트워크 보안, 데이터베이스 튜닝, 분산 시스템 등)에 대해 추가로 공부해 나가시길 응원합니다. 문의가 생기면 관련 커뮤니티나 공식 문서를 참조하며, 실습 위주로 체득하는 것이 가장 좋은 학습 방법입니다.

참고 및 출처: 강의 소개서 및 공개 자료 [77] [19], 그리고 각 주제에 대한 위키피디아와 공식 문서에서 개념을 확인하였습니다 (각주에 표시). 실습 예제 코드는 Python 3 기준으로 작성되었습니다.

---

[1] [77] 강민철의 인공지능 시대 필수 컴퓨터 공학 지식 | 패스트캠퍼스
https://fastcampus.co.kr/dev_online_kmc

2 3 4 5 6 9 12 15 16 17 19 20 28 31 32 33 34 35 37 39 40 41 42 43 44 45 46 47 48 49 52 53 54 55 56 57 58 59 65 67 68 69 [패스트캠퍼스] 교육과정소개서_개발자 강민철의 인공지능 시대의 필수 컴퓨터 공학 지식
https://cdn.day1company.io/prod/uploads/202505/135458-1154/-%ED%8C%A8%EC%8A%A4%ED%8A%B8%EC%BA%A0%ED%8D%BC%EC%8A%A4--%EA%B5%90%EC%9C%A1%EA%B3%BC%EC%A0%95%EC%86%8C%EA%B0%9C%EC%84%9C-%EA%B0%9C%EB%B0%9C%EC%9E%90-%EA%B0%95%EB%AF%BC%EC%B2%A0%EC%9D%98-%EC%9D%B8%EA%B3%B5%EC%A7%80%EB%8A%A5-%EC%8B%9C%EB%8C%80%EC%9D%98-%ED%95%84%EC%88%98-%EC%BB%B4%ED%93%A8%ED%84%B0-%EA%B3%B5%ED%95%99-%EC%A7%80%EC%8B%9D.pdf

7 Chapter 6: Enhancing Performance with Pipelining
https://www.cs.fsu.edu/~hawkes/cda3101lects/chap6/index.html?$$$F6.1.html$$$

8 Introduction to Pipelining - ECE UNM
http://ece-research.unm.edu/jimp/611/slides/chap3_1.html

10 Out of Order Execution with Precise Exceptions | by Enes Harman
https://enesharman.medium.com/out-of-order-execution-with-precise-exceptions-9aea9225b75f

11 Out-of-order execution - Wikipedia
https://en.wikipedia.org/wiki/Out-of-order_execution

13 Little Endian vs Big Endian? - Stack Overflow
https://stackoverflow.com/questions/22030657/little-endian-vs-big-endian

14 What is Endianness? Big-Endian & Little-Endian - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/little-and-big-endian-mystery/

18 Developer Kang Minchul - YouTube
https://www.youtube.com/channel/UC4Hha9F0LZcOtqyL0F3aNkw/posts

21 22 23 24 26 29 30 [운영체제] 운영체제와 프로세스
https://velog.io/@luckyzanie777/%EC%9A%B4%EC%98%81%EC%B2%B4%EC%A0%9C

25 User mode and Kernel mode Switching - GeeksforGeeks
https://www.geeksforgeeks.org/operating-systems/user-mode-and-kernel-mode-switching/

27 61 62 63 64 Difference between File Descriptor and File Pointer - GeeksforGeeks
https://www.geeksforgeeks.org/operating-systems/difference-between-file-descriptor-and-file-pointer/

36 38 66 Memory-mapped file - Wikipedia
https://en.wikipedia.org/wiki/Memory-mapped_file

50 Traditional ARP – Address Resolution Protocol - Practical Networking
https://www.practicalnetworking.net/series/arp/traditional-arp/

51 Explain the process of ARP in mapping an IP address to a MAC ...
https://eitca.org/cybersecurity/eitc-is-cnf-computer-networking-fundamentals/address-resolution-protocol/introduction-to-arp/examination-review-introduction-to-arp/explain-the-process-of-arp-in-mapping-an-ip-address-to-a-mac-address-when-a-client-wants-to-communicate-with-a-web-server-on-the-same-subnet/

60 File descriptor - Wikipedia
https://en.wikipedia.org/wiki/File_descriptor

70 71 72 73 Sharding vs. partitioning: What's the difference? — PlanetScale
https://planetscale.com/blog/sharding-vs-partitioning-whats-the-difference

74 76 SQL vs NoSQL: 5 Critical Differences - Integrate.io
https://www.integrate.io/blog/the-sql-vs-nosql-difference/

75 How In-Memory Caching Works in Redis - freeCodeCamp
https://www.freecodecamp.org/news/how-in-memory-caching-works-in-redis/