HANYANG UNIVERSITY
Cyber-Physical Systems Laboratory

Introduction to Computer Systems Addressing

Prof. Kyuntae Kang

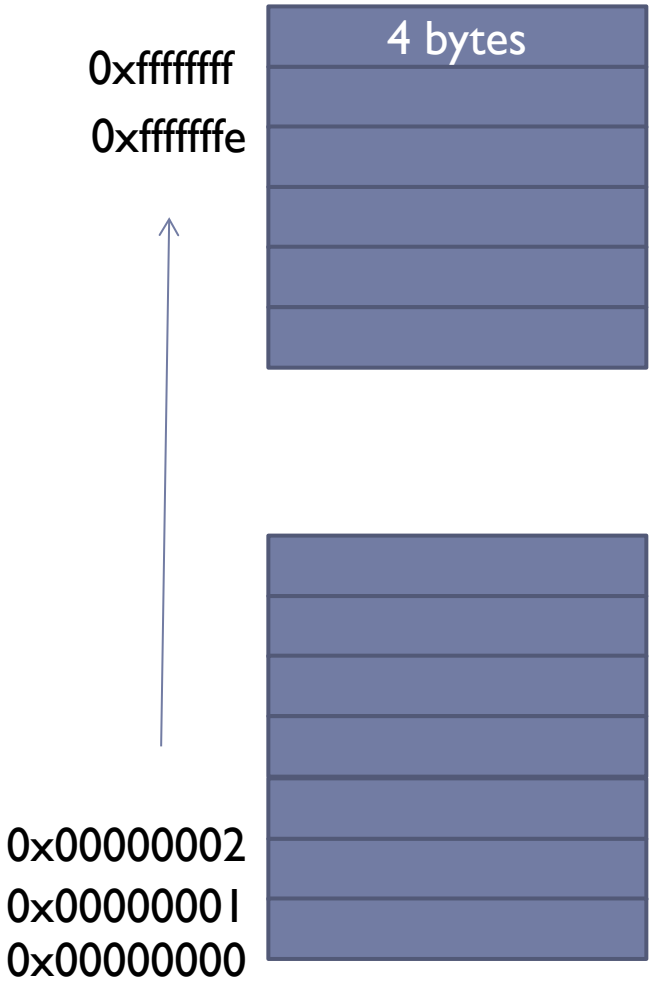
Hexadecimal Notation

- **Hexadecimal** (also **base 16**, or **hex**) is a positional numeral system with a radix, or base, of 16.
  - It uses sixteen distinct symbols, most often the symbols **0**–**9** to represent values zero to nine, and **A**, **B**, **C**, **D**, **E**, **F** to represent values ten to fifteen.
  - For example, the hexadecimal number 2AF3 is equal, in decimal, to $(2 \times 16^3) + (10 \times 16^2) + (15 \times 16^1) + (3 \times 16^0)$, or 10,995.
- Each hexadecimal digit represents four binary digits (bits).
  - For example, byte values can range from 0 to 255 (decimal) but may be more conveniently represented as two hexadecimal digits in the range 00 through FF.
  - Hexadecimal is also commonly used to represent computer memory addresses.

Word Addressing

- Assuming 4-byte (32 bits) word
  - Number of addressable units=2^32
  - 0x00000000~0xffffffff
  - Block size: 4 bytes
  - Total address space=

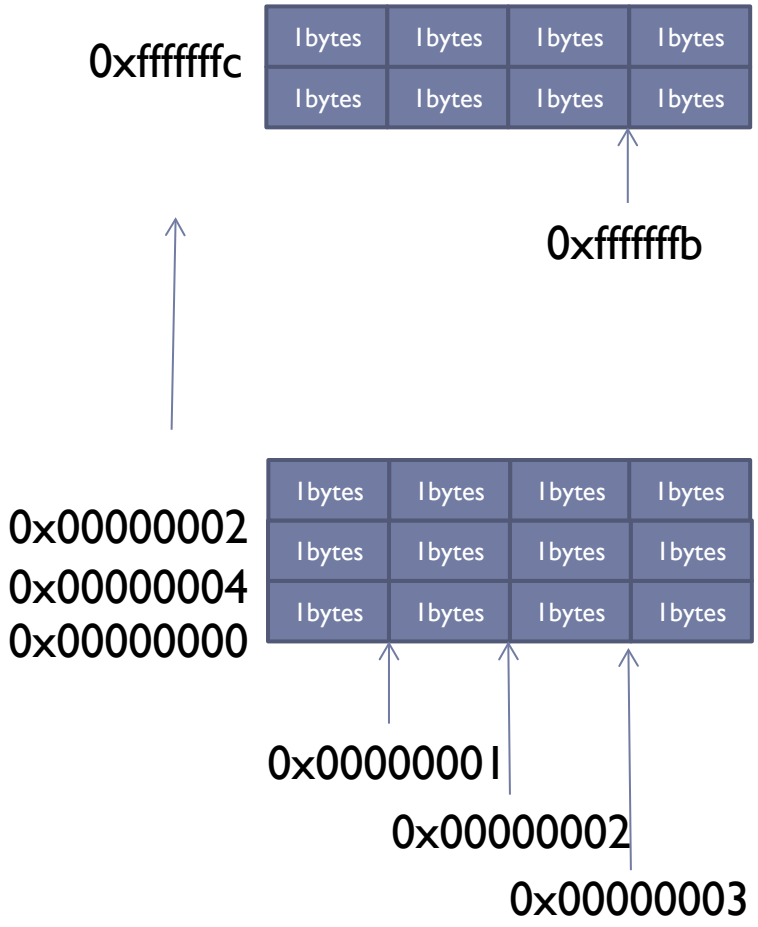
    4 bytes*(2^32) = 16 GB



Byte Addressing

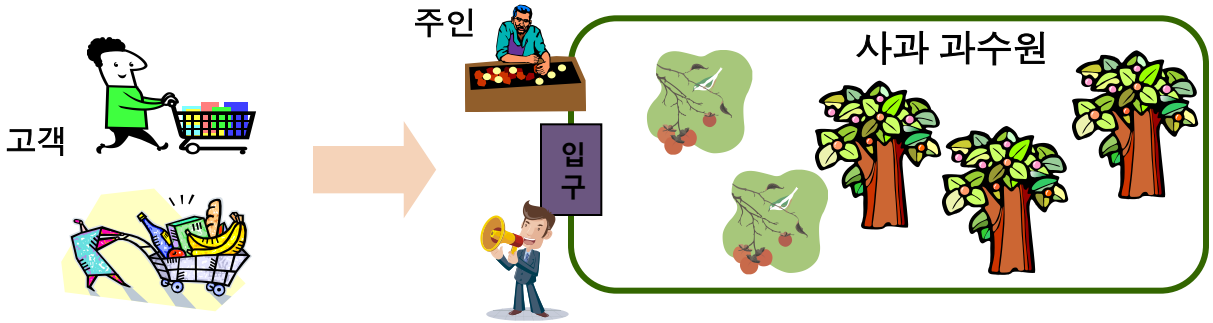
- Assuming 4-byte (32 bits) word
  - Number of addressable units=2^32
  - 0x00000000~0xffffffff
  - Block size: 1 Byte
  - Total address space=

    1 bytes*(2^32) = 4 GB



API

- ‘Proprietary source’ vs. Open source vs. API
  - proprietary source
  - open source
  - API



API Usage Example

- C library call
  - 시스템 콜을 호출하지 않는 라이브러리 함수
    - 특권 명령을 수행할 필요가 없는 기능을 수행
      - 예) sqrt() function

        var = sqrt(4);
  - 시스템 콜을 호출하는 라이브러리 함수
    - 사용자의 편의성을 제공하기 위한 의도
      - 예) printf() 함수는 사용자가 정의한 형태로 버퍼를 조작하여 표준 출력 (디스크립터 1번)에 씀 (write() 시스템 콜 호출)

        printf("grade : %c\n", sp_grade);
    - 시스템 콜 래핑 함수 (system call wrapping function)
      - 전달한 인자들의 유효성 검사
      - 호출한 시스템 콜의 리턴 값에 따라 errno 라이브러리 변수 에 에러 값 설정 (errno.h에 선언됨). 함수 호출 뒤 에러의 종류를 파악하기 위해 이 변수를 참조.
      - 예) read() 라이브러리 함수는 인자들의 유효성 검사 후 read() 시스템 콜을 호출 후 리턴 값을 가공하여 errno 변수에 저장

        read(fd, read_buffer, read_size);

API Usage Example

- system call
  - 유저 레벨 프로그램에서 호출 할 수 있는 커널 서비스 함수
    - 유저 레벨 프로그램은 시스템 콜을 제외한 커널 함수를 직접 호출 할 수 없음
  - 안전함
    - 예) write() 시스템 콜은 데이터가 저장될 디스크의 물리 위치를 지정할 수 없고 운영체제가 정해 줌. write() 시스템 콜로 물리 주소를 지정할 수 있거나, 실제 섹터를 명시하여 디스크 I/O을 위해 호출하는 ll_rw_block() 커널 함수를 유저 레벨에서 직접 호출 할 수 있다면 부트 섹터도 안전 할 수 없음

      syscall(SYS_write, fd, write_buffer, write_size);
- kernel functions
  - 디바이스 드라이버나 커널 코드에서 호출할 수 있는 함수
  - 커널 코드에서 메모리 할당이 필요할 때는 vmalloc(), kmalloc() 등의 커널 함수를 사용할 수 있고 이 외의 커널 함수도 수천 개가 넘음

    ptr = vmalloc(2*1024*1024); // 2M의 메모리를 할당

System Call

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

#define BUFSIZE 10

int main()
{
    int fd;
    char buf[BUFSIZE]={0, };
    fd = open("/etc/passwd",
      O_RDONLY);
    read(fd, buf, BUFSIZE-1);
    printf("%s\n", buf);
    close(fd);
    return 0;
}
```

Library Call ver.

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

#define BUFSIZE 10

int main()
{
    int fd;
    char buf[BUFSIZE]={0, };
    fd = syscall(SYS_open, "/etc/passwd", 0 /* O_RDONLY */);
    syscall(SYS_read, fd, buf, BUFSIZE-1);
    printf("%s\n", buf);
    syscall(SYS_close, fd);
    return 0;
}
```

```
#define __NR_open        5
```
<asm/unistd.h>

```
#define SYS_open __NR_open
```
<bits/syscall.h>

```
0x08048439 <main+53>:   movl   $0x5,(%esp)
0x08048440 <main+60>:   call   0x80482d8
<syscall@plt>
```

Indirect System Call

System Call

```
#include <stdio.h>

int main()
{
    int pid;

    pid = getpid();

    printf("pid : %d\n", pid);
    return 0;
}
```

Library Call Ver.

```
#include <stdio.h>

int main()
{
    int pid;

    __asm__ ("movl $20, %%eax\n\t"
             "int $0x80\n\t"
             "movl %%eax, %0"
             : "=m" (pid)
            );

    printf("pid : %d\n", pid);
    return 0;
}
```

```
#define __NR_getpid   20
```

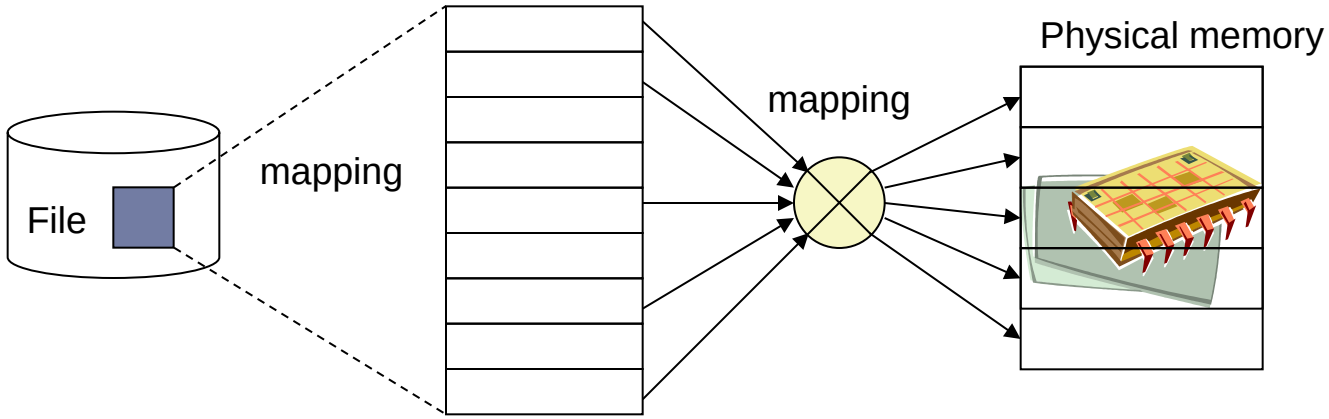
<asm/unistd.h>

Direct System Call

-x86의 경우 시스템 콜 번호는 eax 레지스터에 저장하고 인자는 ebx, ecx, edx, esi, edi, ebp 순으로 저장하여 "int 0x80"을 수행
- 인자의 수가 6개가 넘는 경우는 인자를 스택에 저장하거나 인자들을 구조체로 만들어서 구조체의 주소를 레지스터 인자로 전달

Bonus

- File mapping



Compiler

- Compiler vs. Interpreter
- Syntax vs. Semantic
- Intermediate code generation
  - lexical analysis, scanner
  - syntactic analysis, parsing
  - semantic analysis
- Code optimization
  - Machine dependent optimization
  - Machine independent optimization
- Code generation

Interpreter

- 기능
  - 원시 프로그램의 각 행을 차례로 읽어들여 해석
    - lexical/syntactic analysis는 컴파일러와 유사
  - 해석 후, 원시 프로그램을 내부형식으로 변환하여 실행 : subroutine calls
    - 인터프리터는 원래의 실행문(statement)을 기계어로 번역하여 실행하지 않음
- 인터프리터 실행의 예: C := A + B
  - 1. 우선 이 문장을 스캔하고, 치환문임을 알아냄
  - 2. 우변의 식을 계산하기 위한 루틴을 호출
    - 2-1. 이 루틴은 기호 A와 B를 받아들여, 저장장소를 파악
    - 2-2. 그들의 현재 값을 인출하여 덧셈 후 결과값을 반환
  - 3. 인터프리터는 이 결과값을 기호 C의 저장장소에 저장

Interpreter

- 인터프리터 특성
  - 문법: 각 행의 문장이 독립적으로 변환후 실행이 가능하도록 설계
    - 예: if-then-else statement (QBasic)

```
if condition1 then                          {line 1}
   [statements]                             {line 2..k}
[else                                       {line k+1}
   [statements]]                            {line (k+2)..(n-1)}
endif                                       {line n}
```

    - 해석
      - 쉽게 새로운 행을 확인할 수 있음.
      - endif는 if-then-else 구문을 다른 문장으로부터 분리시킴
      - then은 조건 절(condition clause)을 효과적으로 종료
  - 실행시: 현재의 조건만 만족시키면 됨
    - 예: 위의 예에서 condition1이 만족되지 않으면 line 2..k는 변환.수행될 필요가 없음. (컴파일러에서는 번역함)

Interpreter

- 인터프리터의 장점
  - 작성 용이: 코드 생성이 불필요
  - 프로그램 개발에 유용
    - 편집, 컴파일, 링크/로드, 실행 등이 한 단계에 수행됨
    - 디버깅
      - 기호표, 원시프로그램의 행 번호를 실행시 추적가능
      - 예: 실행시, 오류가 생긴 곳의 행과 변수가 반짝임
  - 기계에 독립적인 면: 기계어 코드 생성 불필요.
    - 예: 새 컴퓨터에 새로운 인터프리터가 필요하면 기존 인터프리터를 컴파일해서 쓰면 됨. (단, 컴파일러가 존재한다면)
- 인터프리터의 단점
  - 원시 프로그램의 실행이 느림: 10 – 100 배
    - 현재 입출력 속도에 좌우되는 경우가 많고, CPU 속도가 빨라지므로 이 문제는 점점 중요해지지 않음
  - 실행시, 대상 프로그램의 원시코드가 필요. (메모리 공간이 요구됨)
  - 실행시, 인터프리터가 필요. (메모리 공간이 요구됨)
  - 모듈러 프로그래밍 불가능.