시스템프로그래밍(한승훈)

Lee Jun Hyeok (wnsx0000@gmail.com)

June 15, 2025

목차

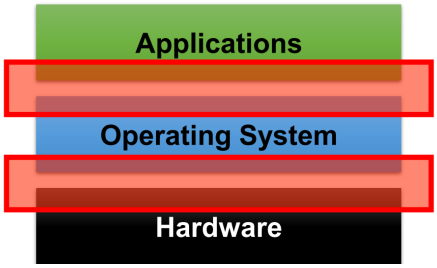
1. 서론

이 필기는 x86_64에서 돌아가는 linux를 기반으로 함.

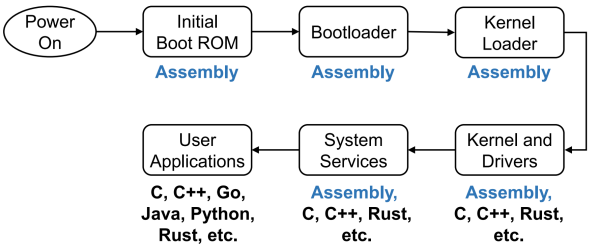
1.1. System Programming

1.1.1. System Programming

System Programming은 다른 software에게 서비스를 제공하는 OS, DB, virtualization과 같은 low-level software platform을 개발하는 분야임.



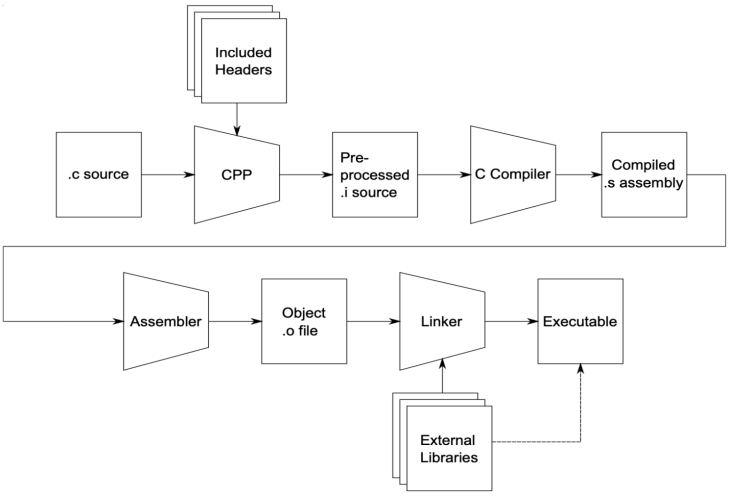
초기 system programming은 assembly로 작성되었지만, UNIX의 등장 이후로는 C로 작성되고 있음. 현재는 C 외에도 C++, go, Rust 등으로도 작성됨. 물론 지금도 assembly 수준에서의 동작을 이해하고, 또한 활용할 수 있어야 함. 예를 들어, OS의 booting 과정은 아래와 같이 assembly 수준에서 수행됨. 이는 hardware가 제공하는 기능을 잘 활용하고, 또한 최적화하기 위함임.



1.1.2. C

1. C

C는 system programming을 위해 설계된 programming language임. os kernel은 대체로 c로 되어있음. 물론 일부 시스템은 c 대신 Rust 등을 시도해 구현되고 있지만, 여전히 완전히 대체하지는 못하고 있음.



2. C Toolchain
C toolchain은 실행 가능한 c 프로그램을 개발하기 위한 소프트웨어 모음임. 이는 아래와 같은 구조를 가짐.

1) CPP(C Preprocessor) : 전처리기에 따라 header file을 추가하고, define된 부분을 대치하는 등의 preprocessing을 수행하는 부분. 이 작업은 .c 파일로 .i 파일을 생성함.

gcc에서는 -E 옵션으로 CPP까지만 수행할 수 있음.

2) C Compiler : .i 파일의 c 코드를 machine에 종속적인 assembly(.s) 코드로 변환하는 부분. 이후 내부적으로 assembler를 활용해 assembly(.s) 코드로 object(.o) file을 생성함.

gcc에서는 -S 옵션으로 assembly까지만 변환할 수 있음.

3) Assembler : assembly 코드를 machine에서 실행 가능한 instruction(기계어)로 변환하는 부분. 현대의 compiler toolchain에서 assembler는 단순히 기계적인 1대1 변환만을 수행함.

gcc에서는 -c 옵션으로 object file까지만 변환할 수 있음.

4) Linker : object file을 executable file로 합치는 부분. 각 object file들의 symbol table을 합치고, unsolved symbol에 해당되는 부분이 다른 object file 또는 라이브러리에 존재하는지 확인함. 또한 symbol reference를 concrete address로 변환함.

c 파일을 compile할 때는 아래와 같은 옵션을 줄 수 있음. 특히 -Wall(warning all), -Werror(warning as error)까지는 쓰는 게 권장된다고 함.

- -Wall: Turn on all warnings
- -Werror: Treat all warnings as errors
- -O2: Turn on moderate optimization
- -g: Include debugging information
- -std=c99: Use the 1999 ISO C Standard
- -o helloworld: Call the output helloworld
- helloworld.c: Compile the file helloworld.c

이에 따라 gcc 등을 활용하는 compiling은 CPP, compiler, assembler, linker의 step으로 수행됨. 이때 입력으로 넣는 파일의 종류(확장자)로 시작 step을, compile option으로 종료 step을 지정할 수 있음. 종료 step을 지정하지 않는 경우 기본적으로 linker까지 모둣 수행됨.

3. Object File
object file은 아래와 같은 부분을 포함하고 있음. 여기에서 Symbol은 변수명과 함수명을 말함. 또한 translation unit은 해당 object file로 이해할 수 있음.

1) Constant Data : 변하지 않는(constant) 값을 저장하는 부분.

2) Static Symbol : locally-defined이고, global하게 접근이 불가능한 변수와 함수를 저장하는 부분.

3) Locally-defined Global Symbol : global하게 접근이 가능하고, 해당 translation unit 안에 완전한 정의가 존재하는 변수와 함수를 저장하는 부분.

4) Unsolved Symbol : global하게 접근이 가능하지만, 해당 translation unit 안에 완전한 정의가 존재하지 않는 변수와 함수를 저장하는 부분.

windows의 WSL(Window Subsystem for Linux)에서도 POSIX를 지원함. 과거 윈도우는 posix를 지원하지 않았는데, 이에 따라 os별로 코드를 나누어 쓰는 등의 불편함이 존재했었음.

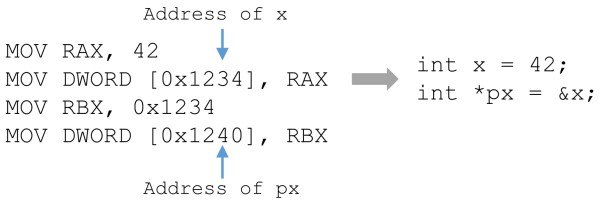
posix가 지원되지 않는 시스템을 고려해야 한다면, 시스템 별로 별도의 코드를 짜야 함.

1.1.3. Pointer

1. Pointer
Pointer는 memory location에 대한 address와 해당 location에서의 type을 포함하는 개념으로, 해당 address에 해당 type으로 접근할 수 있도록 함. c에서는 아래와 같이 pointer를 선언해 활용할 수 있음. java, 파이썬 등에서는 직접적인 pointer 대신 object reference로 유사한 기능을 제공함.

```
char *str = &a;
a = *str;

const char *str; // 해당 location이 const인 경우
```

pointer는 assembly로부터 유래된 것으로, c에서의 pointer는 아래와 같이 어셈블리어와 대응됨.



null은 pointer 값이 존재하지 않음을 나타내는 키워드로, c에서는 0을 값으로 가짐. 이는 주로 초기화 값으로 사용하는데, 실제로 주소가 0인 지점을 참조할 수도 있으므로 대신 -1로 초기화하기도 한다고 함.

2. Dereferencing

*c에서 pointer는 *, ->, []에 의해 dereference될 수 있음.*

이때 pointer는 []로도 dereference가 가능하고, 배열 이름은 첫 번째 원소를 가리키는 pointer로 생각할 수 있지만, 아래의 예시와 같이 실제로 pointer와 배열은 서로 다른 개념임. pointer는 단순 참조를 하는 반면 배열은 동일한 자료형의 데이터들로 구성되어 있음. 이에 따라 pointer와 배열을 혼용하는 것은 좋지 않음.

```
a = ptr[i]
a = *(ptr + i)
```

```
char arr[] = "string";
char arr2[] = arr;
```

"error: invalid initializer"

```
char arr[] = "Hello World";
char *ptr = arr;
```

ptr points to arr[0].

또한 pointer에 정수 덧셈/뺄셈을 적용해 해당 pointer의 자료형만큼 주소 값을 이동할 수 있음.

pointer는 실제 데이터가 어떤지에 상관없이 해당 자료형이 뭔지에 따라 데이터를 읽음. 컴파일러는 이에 따른 오동작을 방지하기 위해 다른 타입의 pointer를 pointer에 할당하면 오류를 발생시키는데, casting을 통해 이를 해결할 수 있음.

3. Dynamic Allocation

아래와 같이 dynamic memory를 allocation하고 free할 수 있음.

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void ptr);
```

이때 free()는 해당 pointer의 값과, 가리키고 있던 데이터를 수정하지 않음. 이에 따라 free() 이후에도 pointer에 주소 값이 남아 있으므로 해당 memory 공간에 접근하면 이전 데이터를 참조할 수 있게 되는 문제가 생기는데, 이를 Dangling Pointer Problem이라고 함.

1.1.4. POSIX

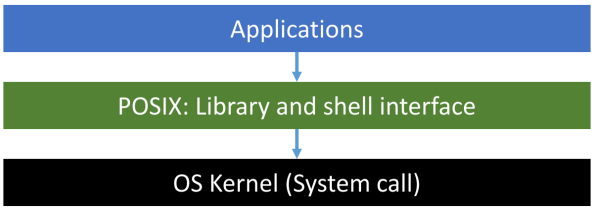
1. POSIX

POSIX(Portable Operating System Interface for UNIX)는 application과 os 사이에서의 interface로, UNIX 계열 os에서의 portability를 위한 것임. 이때 POSIX는 api로 기능하므로, 이를 POSIX API 라고 함. 즉, cpu 및 os에 따라 지원되는 연산이나 system call이 달라질 수 있으므로 unix 계열 POSIX 시스템에서는 POSIX를 표준 API로 사용하는 것임.

POSIX는 library와 shell 명령어 등에 대한 interface를 정의하고, system call이나 binary format(ex. ELF)은 정의하지 않음. 물론 POSIX에서 정의하는 함수들이 대체로 system call을 일대일로 매핑하기는 하는데, 이는 POSIX가 os에 대한 api임을 고려하면 자연스러움.

POSIX API를 활용해 코드를 작성하면 다른 시스템에서도 해당 코드를 recompile하여 바로 실행시킬 수 있음. 당연하게도 POSIX를 지원한다고 해도 실제로 cpu가 다르다면 코드 상에서의 수정은 필요하지 않아도 compile은 다시 해야 실행시킬 수 있음.

POSIX 시스템에서는 c library에서 POSIX API를 구현함. 이때 c standard library와 일부 POSIX function은 겹치는 부분이 존재함.



2. POSIX API 예시

POSIX에는 아래와 같은 함수에 대한 interface들이 정의되어 있음.

1) open() : file open.
2) fork() : process fork.
3) connect() : network connection 생성.
4) exit() : 현재 process 종료.
5) tcsetattr() : terminal에 대한 설정 지정.
6) time() : 현재 시간 반환.

1.1.5. I/O

C와 POSIX에서의 I/O 처리를 알아보자.

1. Stream

*POSIX에서는 file 형태의 stream을 지원함. stream은 자료형이 FILE*이고, 모든 POSIX process는 stdin(0), stdout(1), stderr(2) stream(file)이 default로 열려 있음.*

stdout으로 출력할 때는 출력 버퍼를 사용하는데, 너무 자주 flush하면 비효율적이므로 특정 시간이 지나거나 버퍼가 꽉 차면 flush하는 식으로 동작함. stdout으로 출력했는데 출력이 아예 안돼서 프로그램 초반에 에러가 났나 할 수도 있지만 실제로는 버퍼에서 나오지 못한 것일수도 있음. 반면 stderr는 버퍼를 활용하지 않고 바로 화면에 출력함. 이에 따라 디버깅이나 긴급한 에러를 나타낼 때 stderr를 사용하면 좋음.

2. I/O 함수 예시

C와 POSIX는 다양한 I/O 함수들을 제공하고, 그 일부는 서로 겹침. 아래의 함수들은 stdio.h에 포함되는 것들임.

*1) int puts(const char *s) : s 끝에 개행문자를 추가하여 stdout에 출력함.*

*2) int fputs(const char *s, FILE *fp) : s을 끝에 개행문자를 추가하여 stream fp에 출력함.*

*3) int printf(const char *format, ...) : format을 stdout에 출력함. 임의의 개수의 conversion을 포함할 수 있는 formatting을 활용할 수 있음. 출력된 문자의 개수를 반환함.*

*4) int fprintf(FILE *fp, const char *format, ...) : format을 stream fp에 출력함. 출력된 문자의 개수를*

반환함.

*5) char *gets(char *s) : stdin에서 임의의 길이의 문자열을 입력받음. 버퍼 오버플로우 취약점이 존재하므로 사용하는 것이 권장되지 않음.*

*6) char *fgets(char *s, int size, FILE *fp) : stream fp에서 s를 최대 size-1 바이트만큼 단일 line을 입력받음. 이때 맨 끝에 null을 추가함. end of file 또는 오류인 경우 null을 반환하고, 성공 시에는 s(저장한 주소)를 반환함.*

*7) int scanf(const char *format, ...) : format을 stdin에서 입력받음. 성공적으로 입력받은 conversion의 개수를 반환함.*

*8) int fscanf(FILE *fp, const char *format, ...) : stream에서 format을 입력받음. 성공적으로 입력받은 conversion의 개수를 반환함.*

I/O 장치 처리 방식은 memory mapped io를 사용하는 방식과, instruction을 사용하는 방식으로 나눌 수도 있음. 이때 memory mapped io를 사용하려면 직접 주소를 활용해 메모리를 조작해야 하므로 pointer를 사용해야 함.

1.2. Memory Representation

1.2.1. Memory Representation

memory는 단순히 비트의 나열(syntax)이고, 해당 비트열을 어떻게 해석하는지를 결정하는 자료형(semantic)은 소프트웨어에서의 역할임. 즉, 소프트웨어에서는 memory representation을 정의함.

1. Word

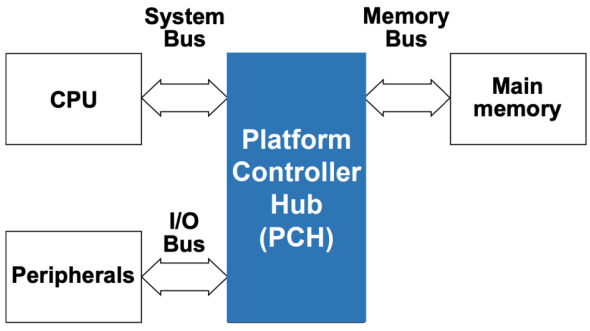
Word는 해당 시스템에서의 native integer size로, 시스템이 처리하는 데이터의 기본 단위임. 이는 시스템 및 cpu에 하드웨어적으로 지원되는 정수 타입임. 컴퓨터 시스템에서는 word 단위 또는 그 배수 단위로 데이터를 처리하는 것이 자연스러움. 이에 따라 memory는 word의 집합으로 이해할 수도 있음.

word로는 cpu가 한 번에 처리하는 데이터 크기인 cpu word(시스템 word)도 존재하지만, memory bus의 width인 memory word도 존재함. 시스템의 word는 width와 관련되어 있고, cpu의 word는 register 크기와 관련되어 있음.

소프트웨어는 convention에 따라 word에 대해 semantic을 부여하여 특정 자료형으로 활용함.

2. Word와 Bus

cpu와 memory를 포함하는 하드웨어적인 구조는 아래 그림과 같음. clock cycle 차이 등에 의해 cpu와 메모리는 직접 연결될 수 없고, PCH(Platform Controller Hub)라고 하는 중간 장치를 통해 bus로 연결됨. 과거에는 PCH 대신 여러 개의 bridge를 사용했는데, 지금은 하드웨어의 발전에 의해 PCH 하나로 처리함.



프로그래머의 시점에서 단순화했을 때, bus는 단순히 비트 별 line으로 구성되어 있음. 예를 들어, line이 8개면 bus width는 8비트임. cpu가 보는 memory의 word size는 memory bus의 width임. system bus와 memory bus의 크기가 맞아야 성능 저하가 발생하지 않지만, 실제로 system bus에 비해 memory bus가 더 작을 수 있는데 이 경우에는 PCH 또는 bridge가 적절히 fetch하는 것으로 이를 중재함. 즉, memory로부터 cpu로 데이터를 가져올 때 데이터는 memory word(memory bus width) 만큼씩 fetch

되며, 해당 데이터들은 cpu word width만큼씩 cpu의 register에 저장됨.

현대의 architecture에서는 caching 등을 이유로 memory word size가 cpu word size보다 크다고 함. 특히 지역성을 활용하기 위해 64바이트의 배수로 읽어온다고 함.

3. Exploring Representation

아래와 같은 코드로 특정 변수의 memory representation을 확인할 수 있음. 쉽게 확인할 수 있는 것처럼, 이 코드는 바이트 단위로 해당 변수의 비트열을 출력함.

```
    #include <stdio.h>

    void dump_mem(const void *mem, size_t len)
    {
        const char *buffer = mem;
        size_t i;

        for(i = 0; i < len; i++)
        {
            if((i > 0) && (i % 8 == 0))
            {
                printf("\n");
            }

            printf("%02x ", buffer[i] & 0xff); // & 0xff은 sign extension을 고려
한 것임
        }
        if(i > 1)
        {
            puts("");
        }
    }

    ...

    int x = 1923;
    dump_mem(x, sizeof(x));
```

참고로, word 등 비트 값을 나타낼 때는 16진수를 주로 사용함. 16진수 두 자리로 8비트 길이의 비트열을 표현할 수 있음.

1.2.2. C Structure

1. C structure

C의 structure(구조체)는 하나 이상의 primitive type으로 구성되는 복합 자료형으로, 아래와 같이 정의하고 사용할 수 있음.

```
    struct Mystruct {
        int value;
        struct Mystruct *next;
    }

    struct Mysturct tmp;
    struct Mystruct *pc = &tmp;
    (*pc).value;
    pc->value;
```

2. Alignmnet

structure에서 각 변수는 먼저 등장하는 것부터 낮은 쪽 주소를 가지게 되고, 변수들은 정렬(alignment)됨. 이때 정렬을 위해 각 변수의 주소는 해당 자료형의 크기로 나누어 떨어져야 함. 또한 정렬 시에 비는 공간이 생기면 padding을 붙임.

어떤 변수에 대해, 그 자료형으로 나누어 떨어지지 않는 주소를 사용해 일부 데이터 조각에 접근하는 것을 Unaligned Access라고 함. 대부분의 시스템에서 unaligned access는 error로 처리됨. 또한 접근이 가능하다고 해도 데이터는 memory word 단위로 fetch되므로 중간 지점부터 접근하는 경우 여러 개의 word를 fetch한 뒤 해당되지 않는 부분을 제거하는 과정을 거치므로 속도가 느림.

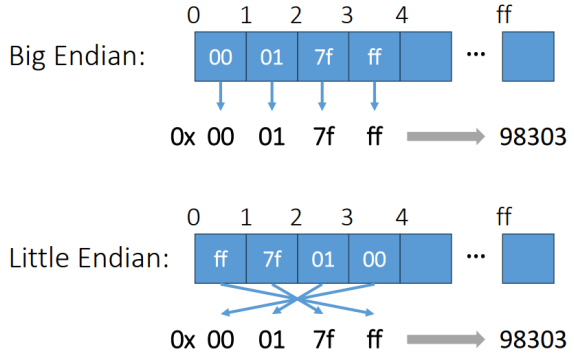
1.2.3. Byte Ordering

word들은 memory에서 바이트들로 구성되는데, Byte Ordering에 의해 해당 바이트들이 어떤 순서로 저장될지가 결정됨. 이는 os에 의해 정의되며, c 수준에서는 그 정보를 확인할 수 없음.

대표적인 byte ordering에는 big endian과 little endian이 존재함.

1) Big Endian : 낮은 자리수의 값이 낮은 주소에 저장되는 방식. 실세계에서 숫자를 쓰는 순서와 동일함.

2) Little Endian : 높은 자리수의 값이 낮은 주소에 저장되는 방식.



1.2.4. Integer

1. Integer 표현

signed 정수는 부호 비트(sign bit)를 사용해 부호를 나타냄. 주로 최상위 비트(가장 왼쪽 비트)가 부호 비트로 사용됨. 양수는 부호 비트가 0, 음수는 부호 비트가 1인데, unsigned와의 유사성을 지키기 위해 양수가 0, 음수가 1인 것임. 이때 최상위 비트인 부호 비트는 MSB(most significant bit)라고도 하고, 최하위 비트는 LSB(least significant bit)라고도 함.

정수 중에서 양수는 단순히 이진수로 나타내면 되는데, 음수는 1의 보수, 2의 보수 등 그 표현 방법이 여러 가지임. 물론 현대의 대부분의 컴퓨터에서는 2의 보수를 사용함. 참고로, 여기에서 보수는 뒤집어 표현하는 것을 말함.

1의 보수와 2의 보수 모두 부호 비트가 존재하지만, 실제로 계산 시에는 부호를 판단하는 것 이외에 이를 별도로 처리하지 않아도 됨. 즉, 부호만 판단하고 계산은 단순히 수행하면 됨.

2. 1의 보수

비트를 전부 뒤집(NOT 연산)었을 때 양수와 음수가 대응되도록 표기하는 방법을 1의 보수 표현(One's Complement Representation)이라고 함.

1의 보수로 나타낸 이진수를 10진수로 나타낼 때는, 다른 모든 비트 자리에 대해서는 양수와 동일하게 2의 거듭제곱으로 계산하고, 최상위 비트에 대해서는 $-2^{W-1}+1$로 계산할 수 있음.

1의 보수에서는 0을 표현하는 방법이 00...0과 11...1 두 가지이므로 불완전함.

예를 들어, 3은 0011, -3은 1100임.

3. 2의 보수

비트를 전부 뒤집(NOT 연산)고 1을 더했을 때 양수와 음수가 대응되도록 표기하는 방법을 2의 보수 표현(Two's Complement Representation)이라고 함.

1의 보수로 나타낸 이진수를 10진수로 나타낼 때는, 다른 모든 비트 자리에 대해서는 양수와 동일하게 2의 거듭제곱으로 계산하고, 최상위 비트에 대해서는 -2^{W-1}로 계산할 수 있음.

비대칭적인 표현 범위를 가지기 때문에 0이 00...0으로 유일하게 표현됨.

4. Sign Extension
2의 보수로 저장하는 부호 있는 정수를 해당 정수의 비트열이 필요로 하는 것보다 더 큰 공간에 저장하면 부호 비트가 복사되어 왼쪽에 추가됨. 이를 sign extension이라고 함. 이에 따라 2의 보수가 제대로 동작함.

sign extension이 적용되었더라도 2의 보수에 대한 변환 및 연산은 동일하게 수행하면 됨.

예를 들어, 0010은 0000 0010, 1010은 1111 1010로 저장됨.

1.2.5. Floating Point

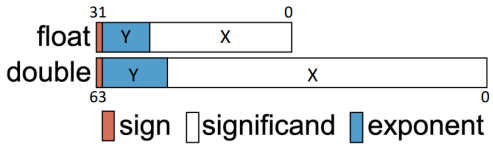
Floating Point는 float이나 double로 표현됨. IEEE 754에서는 아래와 같이 이를 정의함.

IEEE Standard 754 defines a particular floating point format.

If a floating point number is $x \times 2^y$, in IEEE 754:

- A single precision number (`float`) has a 23-bit x and 8-bit y
- A double precision number (`double`) is 52-bit x and 11-bit y

Each has a one-bit sign.



2. Linking

linking과 ELF에 대해 더 자세히 알아보자.

2.1. ELF

2.1.1. ELF

1. Binary File
binary file로는 아래와 같은 것들이 있음.

1) Relocatable Object File(.o file) : 서로 다른 relocatable object file과 합쳐져 executable object file을 구성할 수 있는 형태로 code와 data를 저장하는 file. 하나의 relocatable object file은 하나의 소스 코드 파일(.c file)로 만들어짐.

2) Executable Object File(a.out file) : memory에 복사되어 바로 실행될 수 있는 형태로 code와 data를 저장하는 file.

3) Shared Object File(.so file) : memory에 load되어 load time 또는 runtime에 dynamic하게 link될 수 있는 특수한 형태의 relocatable object file.

2. ELF
ELF(Executable and Linking Format)는 relocatable object file, executable object file, shared object file을 위한 통합된 표준 file format으로, 주로 UNIX계열 os에서 활용함. ELF executable은 이름에서도 알 수 있듯이 아래와 같은 두 가지 정보를 포함함.

1) 해당 object를 Load하고 execute하는 데에 활용하는 정보
2) 해당 object를 link하는 데에 활용하는 정보

ELF를 따르는 executable을 ELF file이라고 함.

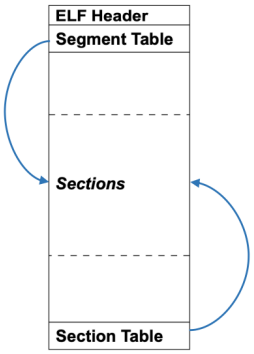
3. ELF Structure
ELF file이 가지고 있는 데이터는 section과 segment로 활용됨. 즉, ELF file은 2가지 view를 지원함.

1) Section은 linker가 해당 ELF file을 linking하기 위한 정보를 포함하며, ELF file 내부의 데이터를 논리적으로 나누는 단위임.

2) Segment는 loader가 해당 ELF file을 load하고 execute하기 위한 정보를 포함하며, ELF file이 실행될 때 memory에 load되는 단위임.

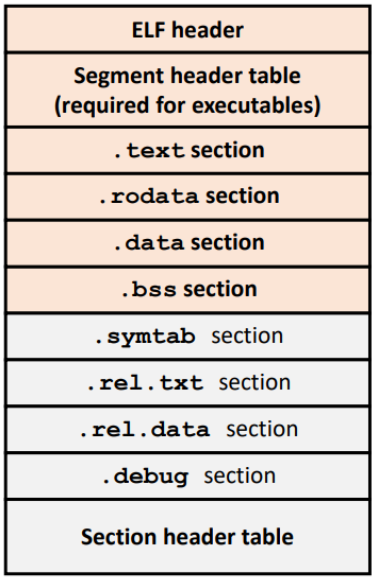
이에 따라 ELF file은 아래와 같은 구조를 가짐.

1) ELF header : 해당 file에 대한 정보(endian 등)를 포함함.
2) Segment Table : segment에 대한 정보를 포함하며, loader가 이를 활용함.
3) Section Table : section에 대한 정보를 포함하며, linker가 이를 활용함.



2.1.2. ELF Format

ELF file의 format은 아래와 같음.



1) ELF header : word size, byte ordering, file type, machine type 등을 정의하는 부분.

2) Segment header table : segment 관련 정보를 저장하는 부분. page size, segment의 virtual address,

segment size 등이 저장됨.

3) .text section : code(instruction)를 저장하는 부분.

4) .rodata section : non-code data 중 read only인 것을 저장하는 부분. 예를 들어, constant, string이 있음.

5) .data section : initialized global variable을 저장하는 부분.

6) .bss section : uninitialized global variable을 저장하는 부분. 이는 값을 가지지 않으므로 실제로 공간을 차지하는 대신 section table에만 그 정보가 저장됨. 즉, .bss는 file에서 물리적으로 존재하지 않음.

7) .symtab section : symbol table을 저장하는 부분.

8) .rel .text section : .text section에 대한 relocation 정보를 저장하는 부분. 관련 instruction들의 address가 저장됨. 대부분의 call은 상대 주소로 되어 있고, 이 경우 추가적인 relocation이 필요하지 않음. 하지만 함수 포인터를 쓰는 경우 등 절대 주소를 써야 하는 상황이 있을 수 있는데, 이때 해당 section의 정보가 활용됨.

9) .rel .data section : .data section에 대한 relocation 정보를 저장하는 부분. 관련 data들의 address가 저장됨.

10) .debug section : symbolic debugging을 위한 정보를 저장하는 부분.

11) Section header table : section 관련 정보를 저장하는 부분. 각 section에 대한 offset과 크기가 저장됨.

2.2. Linking

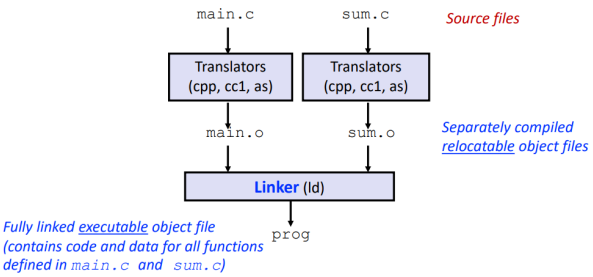
2.2.1. Linking

1. Linking

Linking은 object file들과 관련 library들을 결합하여 하나의 executable을 생성하는 과정임. compiler가 소스 코드로 object file을 생성하면, linker가 이를 활용해 linking을 수행함.

linking은 아래와 같이 두 가지로 나눌 수 있음.

1) Static Linking : object file과 관련 library 전체를 모두 포함하는 하나의 독립적인 executable을 생성하는 방식의 linking.



2) Dynamic Linking : object file과 관련 library 모두를 포함하는 대신 linking 시에는 참조만 하고, 실행 시에 필요한 shared library를 memory에 load하는 방식의 linking.

2. Linker

Linker는 linking을 수행하는 소프트웨어임. linker의 사용은 아래와 같은 두 가지 측면에서의 이점이 있음.

1) Modularity : 프로그램을 여러 개의 작은 file들로 나눠 작성할 수 있고, library 등을 활용할 수 있음.
2) Efficiency : 매번 모든 부분을 compile하는 대신, 일부만 compile할 수 있으므로 효율적임. 또한 library를 사용할 때, executable과 memory에 실제로 활용하는 부분만 올려 활용할 수 있음.

linker는 우선 symbol resolution을, 이후 relocation을 수행하는 것으로 linking을 처리함.

당연하게도 local symbol은 local variable과는 다른 개념임. 애초에 local variable은 linking 및 load 시에 file 및 memory에 존재하지 않음. linker는 local variable에 대해 알지 못함.

또한 local variable과 local static variable도 서로 다른데, local variable은 stack에 저장되고 local static variable은 .bss 또는 .data에 저장됨.

서로 다른 함수에 존재하는 동명의 local static variable은 서로 다른 memory 공간을 가지게 되고, symbol table에도 서로 다른 이름으로 저장됨.

2.2.2. Symbol Resolution

1. Symbol Resolution
Symbol Resolution은 각 object file이 가진 symbol table을 통해 symbol reference와 symbol definition을 mapping하는 과정임.

Symbol은 함수, 변수, 클래스 등의 식별자임. ELF file에서는 symbol table이 존재하여 구조체 형태로 symbol들의 이름, 상대 위치, 크기 등을 저장함. linker에서 활용하는 symbol은 아래와 같이 3가지로 나눌 수 있음.

1) Global Symbol : 특정 module에서 정의되고 다른 module에서도 참조할 수 있는 symbol. 예를 들어, staic이 아닌 함수와 global variable이 있음.

2) External Symbol : 특정 module에서 참조되는데 다른 module에서 정의된 symbol.

3) Local Symbol : 특정 module에서 참조되고 해당 module에서 정의된 symbol. 즉, 내부적으로만 활용되는 symbol임. 예를 들어, static을 사용해 정의된 함수와 global variable이 있음.

2. Duplicated Symbol 처리
linker는 symbol 중 함수와 initialized global은 Strong, uninitialized global은 Weak로 취급함. linker는 duplicated symbol(동일한 이름의 symbol)에 대해 이에 따른 처리를 수행하는데, 그 규칙은 아래와 같음. 규칙을 위반하면 linker error가 발생함.

rule1) 여러 개의 strong symbol은 허용되지 않음. 즉, 특정 이름의 strong symbol은 하나만 존재할 수 있음.

rule2) strong symbol이 하나, weak symbol이 여러 개인 경우 weak symbol들이 해당 strong symbol을 참조한다고 판단함.

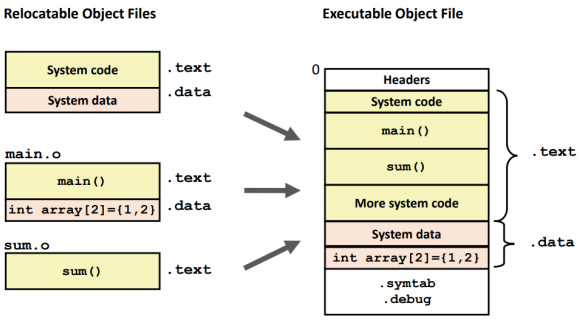
rule3) weak symbol만 여러 개인 경우 임의로 하나를 택하여 나머지는 해당 symbol을 참조한다고 판단함. 이는 compiler에 따라 동작이 달라질 수 있음.

gcc는 -fno-common이 기본인데, 이 옵션이 지정되어 있으면 rule3에 해당하는 상황을 오류로 처리함.

strong, weak를 섞어 쓰지 말고 명확하게 표기하는 것이 좋음. 특히 rule2의 상황에서 strong과 어떤 weak의 자료형이 다른 경우, 그리고 rule3의 상황에서 각 weak의 자료형이 다른 경우 오버플로우 등의 문제가 발생함. 만약 둘다 구조체인데 서로 그 구조가 다르다면 문제가 더 커짐. static을 붙여 local로 사용하거나, external global variable을 참조해야 하는 경우 extern을 붙여 외부 파일을 참조함을 명시하는 것이 좋음.

2.2.3. Relocation

1. Relocation
Reloation은 여러 object file들이 가지는 각 section들을 합쳐 하나의 section으로 만들고, 각 object file에 대해 상대 주소를 가지는 symbol들의 주소를 실제 memory 위치에 맞게 재조정하는 과정임.



Relocation entry는 오브젝트 파일 내에서 특정 symbol의 reference가 실제 memory 주소로 변환되는 방법 등을 나타내는 정보임. 이는 rel. text., rel. data.에 저장됨.

linker도 relocation을 수행하지만, loader 또한 relocation을 수행함. linker가 수행하는 작업은 위치를 잘 조정해 합치는 것이고, loader가 수행하는 작업은 이후 메모리에 올릴 때 random 배치를 위한 reloaction table을 잘 구성하는 것임.

2.3. Library Linking

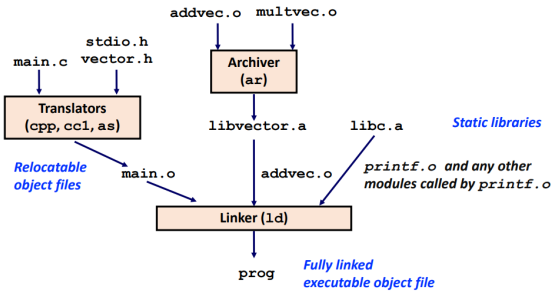
2.3.1. Library Linking

library를 linking하는 방법에는 아래와 같이 2가지 방식이 있음.

1. Static Library
Static Library는 관련 object file들을 하나의 file로 합치는 방식임. 이렇게 동작하는 library를 Archive(.a file)라고 함. archive는 index 정보를 포함하고 있어서 빠른 탐색이 가능하다고 함. 즉, 관련 library의 모든 object file을 linking하는 static linking을 적용함.

linker는 지정한 obejct file과 archive들로부터 unsolved symbol에 대한 reference를 찾고 해당되는 archive를 link함. 이에 따라 당연하게도 library에 변화가 발생하면 다시 compile해야 하고, 전체 archive를 link하여 저장 및 실행하므로 그 비용이 높음. 또한 기본적으로 linker가 작성된 file을 순서대로 확인하는데, library를 맨 뒤에 작성하거나 하지 않으면 성능이 더 떨어짐.

c에서는 libc.a(c standard library)와 libm.a(c math library) 등이 있음.

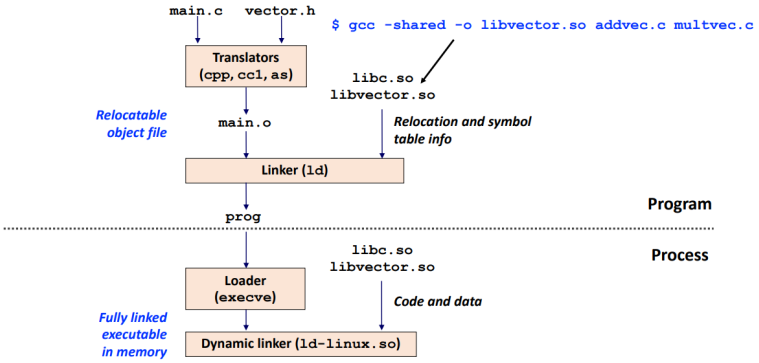


2. Shared Library
Shared Libraray는 관련 object file들을 load time 또는 runtime에서도 동적으로 load하고 link하는 방식임. 이렇게 동작하는 library를 Dynamic Link Library(DLL. .so file)이라고 함. 즉, dynamic linking을 적용함.

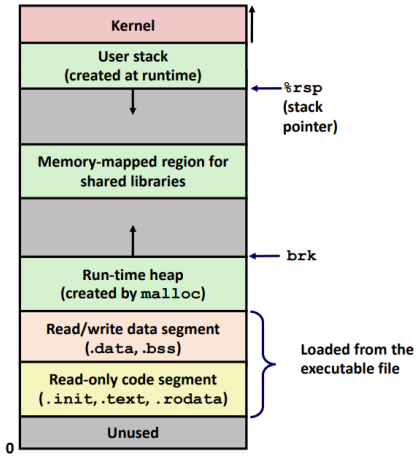
1) link time에서의 dynamic linking
link time에는 object file에 실제 코드를 추가하는 대신 symbol table만 수정함.

2) load time에서의 dynamic linking
shared library를 memroy에 load함. linux에서는 ld-linux.so라는 linker에 의해 수행되고, standard c library(libc.so) 등이 dynamic하게 link됨.

3) runtime에서의 dynamic linking
shared library를 runtime에 memroy에 load하여 활용함. linux에서는 dlopen()이라는 함수로 runtime에 dynamic하게 link할 수 있음.



memory에서 shared library는 stack과 heap 사이에 저장됨.

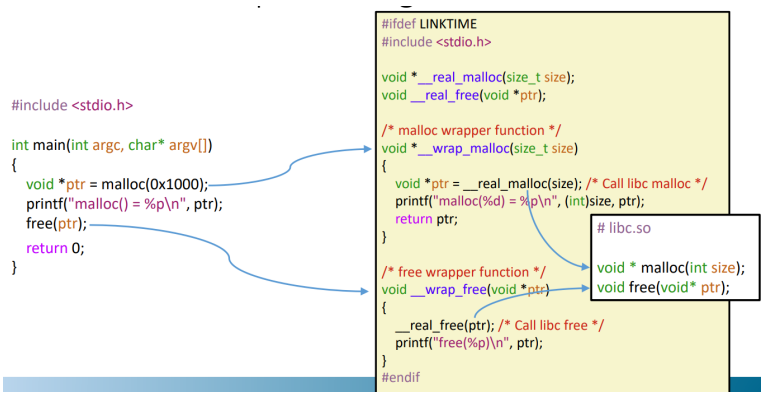


2.3.2. Library Interpositioning

Library Interpositioning은 library 함수 호출을 사용자 정의 함수로 redirect하는 기법으로, compile time, link time, load/runtime에서 구현될 수 있음. memory profiling, debugging, security 등을 위해 사용함.

1) compile time : 전처리기 등을 활용해 소스 코드를 조작함.

2) link time : linker의 symbol table을 조작함. 예를 들어, 아래와 같이 ___wrap__malloc이 malloc을 대체하게 되고, malloc은 ___real__malloc으로 호출하게 됨.



3) load/runtime : dynamic linking을 활용해 조작함. 예를 들어, LD_PRELOAD environment variable에 호출에 대한 우선순위를 지정할 수 있음.

3. Process

3.1. Executable Format

3.1.1. Executable Format

각 Executable(실행 파일)은 Executable Format으로 저장됨. executable format은 아래와 같이 실행에 필요한 정보들을 포함함.

1) program에 필요한 environment
2) program code와 데이터
3) 기타 metadata들

executable format으로는 UNIX의 ELF, windows의 PE, macOS의 Mach-O 등이 있음.

초창기의 executable format은 단순히 raw memory로 구성되어 있었고, 이를 단순히 memory에 복사한 뒤 실행했음.

3.2. Loading

3.2.1. Loading

Loading은 실행을 위해 executable 또는 library를 memory에 올리는 과정임. linux에서는 ELF executable과 library에 대해 대략적으로 아래와 같은 순서로 loading이 수행됨. 이때 마지막 단계에서 program의 시작 지점을 Entry Point라고 함.

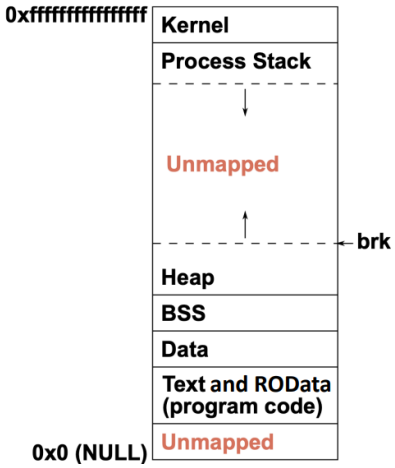
1) kernel(os loader)이 ELF executable의 여러 부분을 memory에 올림.
2) kernel이 loader(ld-linux.so)를 memory에 올림.
3) kernel이 loader를 호출하고, loader는 memory에 올라온 부분에 대해 여러 작업을 수행함.
4) loader가 program의 시작 지점으로 jump하도록 함.

최초로 executable을 memory에 올리는 것은 커널의 os loader인데, os loader가 loading을 전부 처리하기에는 overhead가 커서 이후 ld-linux.so에게 이 역할을 위임한다고 함.

3.2.2. Memory Layout

1. Memory Layout
executable은 process로서 실제로 실행되기 위해 memory에 load되는데, 이때 load된 process의 memory layout에 대해 알아보자. 대부분의 POSIX 시스템에서 process의 memory layout은 아래와 같이 ELF section과 유사한 형태를 가짐.

이때 현대 시스템에서는 ELF file에는 나타나지 않는 shared library를 memory에 포함시키고, 각 section의 위치를 randomize하는 등의 처리를 적용하기는 함. 하지만 이 경우에도 각 section의 순서는 동일함.



1) Unmapped : 가장 낮은 주소 부분은 비어있음. 이에 따라 null이 invalid pointer로 활용될 수 있음.

2) Text : ELF file의 .text가 그대로 활용됨. 물론 이때 loader에 의해 위치가 조정되는 등의 처리가 적용될 수 있음.

3) ROData : ELF file의 .rodata가 그대로 활용됨.

4) Data : ELF file의 .data가 그대로 활용됨.

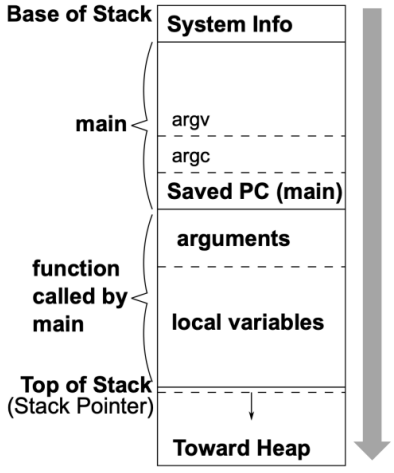
5) BSS : ELF file의 .bss에 대응되는 부분. 앞에서 다룬 것처럼 file에서 BSS section은 물리적으로 존재하지 않고, loading 시에 memory에 공간이 생성됨.

2. Stack/Heap

stack/heap은 모두 memory 공간이 grow될(늘어날) 수 있음. 이에 따라 그 사이에 unmapped 영역이 존재함.

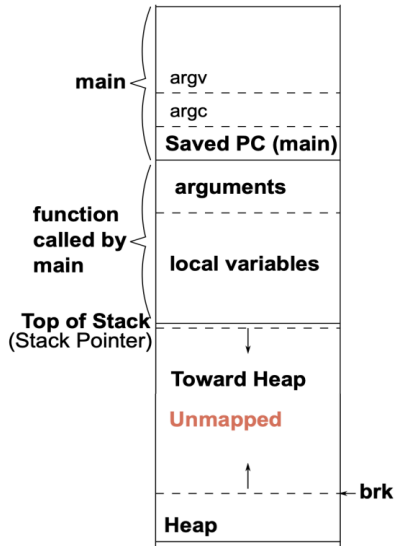
1) Stack : 함수 호출에 따른 local variable을 저장하는 stack 공간으로, loader에 의해 구성됨.

stack은 함수 호출에 따라 downward로(거꾸로) 자라고 함수가 종료되면 줄어드는데, 이 과정은 kernel에 의해 automatic하게 관리됨. 각 호출된 함수에 대해 존재하는 memory 공간은 Stack Frame이라고 함. stack frame은 pc, argument, local variable 등을 포함함.



2) Heap : explicit하게 할당된 dynamic memory를 저장하는 공간으로, loader에 의해 구성됨.

Program Break은 unmapped와 heap 사이에 존재하여 heap의 끝 지점을 나타내는 가상의 memory 주소임. kernel은 이를 활용하여 heap에서 사용 가능한 공간을 나타냄. dynamic memory를 explicit하게 할당받으면 program break이 stack 쪽으로 움직이며 heap 공간이 늘어남.



3.3. Process

3.3.1. Process

1. Process

Process는 실행 중인 program의 instance임. program이 단순히 instruction의 집합이라면, process는 해당 instruction과, memory, system 리소스 등을 포함함.

Process Environment는 process가 실행되는 동안 접근할 수 있는 데이터와 설정 등의 환경으로, process environment로는 system call, file system, signal 등이 있음.

2. Kernel Service

개별 UNIX process는 cpu와 memory 전체를 독점해 사용하는 것처럼 동작하고, 실제로 독립된 memory 공간을 가지고 있어(memory isolation) 다른 process로부터 보호됨. 이런 구성에는 하드웨어적인 지원이 필요하고, 하드웨어에 대한 조작은 kernel에 의해서만 수행될 수 있음. 개별 process에서는 system call로 kernel의 service를 활용함.

즉, kernel은 Supervisor mode(Kernel mode)로 동작하며 memory, file, 하드웨어 장치에 대한 직접적인 처리를 수행하고, 개별 process는 User mode로 동작하며 kernel service를 이용함. 이렇게 kernel에 의해 생성되는 process environment를 Userspace라고도 함.

supervisor mode와 user mode를 통틀어 Protection Domain이라고 함.

3.3.2. System Call

1. System Call

System Call은 user mode에서 실행되는 process가 kernel service를 활용할 수 있도록 kernel이 제공하는 interface임.

system call은 protection domain 사이의 이동이 수행되는데, 이는 하드웨어적인 지원이 필요하므로 system call은 복잡하고 비용이 많이 듦.

x86_64 linux에서는 interrupt 또는 processor instuction을 활용해 system call을 호출하게 됨.

2. Invoking System Call

일반적인 함수는 호출 시에 argument를 register나 stack에 넣고, 현재 pc를 저장한 뒤 해당 함수의 시작 지점으로 pc 값을 수정함. 이후 함수가 종료되면 반환값을 register에 넣고, 저장해둔 pc를 불러와 해당 위치로 돌아감. 이와 달리 system call은 아래와 같이 동작함.

1) 호출

system call은 호출(inovke) 시에 system call number와 argument를 register에 넣고, int 0x80, sysenter,

syscall 등의 instruction을 호출함. 이후 cpu는 protection domain을 변경하고 system call에 따른 특정 위치로 jump함.

2) 종료
system call이 종료되면 반환값을 register에 넣고, iret, sysleave, sysret 등의 instruction을 호출함. 이후 cpu는 protection domain을 변경하고 system call을 호출한 원래 위치로 jump함. 이때 kernel이 user 보다 권한이 많으므로 단순 반환할 수도 있지만, 이 방법은 보안 상 취약하다고 함.

이때 system call이 사용하는 instruction은 processor instruction으로, 이는 processor에 종속적인 instuction이므로 환경에 따라 달라질 수 있음.

이렇게 system call에 대한 호출과 종료는 복잡한 과정을 가지므로, C library는 system call에 대한 wrapper function들을 제공함. 이 function들은 위에 정리한 과정을 단순히 자동화한 것임.

3.3.3. More Environment

process enviroment로는 또 어떤 것들이 있는지 알아보자. 이 또한 kernel에 의해 처리됨.

1. Current Working Directory
모든 process는 current working directory(cwd)가 존재함. 아래와 같은 system call 또는 함수를 호출해 관련 정보를 얻거나 지정할 수 있음.

1) chdir() : cwd 지정.
2) getcwd() : cwd 반환.
3) getwd() : cwd 반환. 하지만 보안 상 위험성이 있어 사용해선 안 됨.

2. Environment Variable
모든 process는 environment variable을 가짐. 이는 environ이라는 global array에 저장되어 있음. 아래와 같은 함수를 호출해 관련 정보를 얻거나 지정할 수 있음.

1) getenv() : environment variable 반환.
2) setenv() : environment variable 지정.

environ은 fork()에 의해 복사됨.

```
char *a = getenv("HOME");
```

3. File
kernel은 모든 process에 대해 stdin, stdout, stderr를 file로 제공함.

뒤에서 설명하겠지만 각 file은 file descriptor라고 하는 integer 값으로 구분되는데, stdin, stdout, stderr는 각각 0, 1, 2임. 또한 각 file descriptor에 대해 최근 read/write를 수행한 위치를 저장해둠.

file descriptor는 fork()에 의해 복사되고, read/write 위치 정보도 복사됨. exec()에서 close-on-exec flag를 지정하여 exec() 시에 복사하는 대신 새로운 process에서는 닫도록 지정할 수 있음. 이때 stdin/stdout/stderr는 닫히지 않음.

3.4. Process Lifecycle

system call을 활용한 proess의 lifecycle을 살펴보자.

3.4.1. Creation and Execution

1. Creation
기본적으로 UNIX에서는 process를 만드는 방법으로 fork() system call만을 제공함.

fork()는 해당 system call을 사용한 process를 아래와 같은 과정을 통해 복제하며, 부모(원본) process에서는 child proces의 PID를 반환하고 자식(새로운) process에서는 0을 반환함.

1) 새로운 PID와 kernel structure를 생성함.
2) 새로운 process에 대한 memory 영역을 생성함.

3) 생성한 memory 공간에 기존 process의 모든 정보를 COW(Copy-on-Write)로 복사함.
4) 두 process 각각에 대해 return함.

이때 결과로 도출된 두 process 중에 뭐가 먼저 실행되는지는 예측할 수 없음.

2. Process 계층 구조
fork()를 사용해서 process를 생성했을 때, 기존에 실행 중이던 process를 부모 프로세스(Parent Process), 새로 생성된 process를 자식 프로세스(Child Process)라고 함. 모든 process는 parent process를 가지고, fork()나 posix_spawn()을 사용했다면 child proces도 가짐.

root process 또는 init은 PID가 1인 process로, OS에서 최초로 실행되는 process임. process들은 계층적 구조를 가지는데, os 내부의 모든 process들은 intit의 child process임.

3. Execution
fork()로 새로운 process를 만들면 parent process와 완전히 동일한 작업을 수행하게 되므로, exec() system call을 사용해 특정 program을 실행하도록 할 수 있음. 또는 POSIX에서는 posix_spawn()도 사용 가능함.

exec() 계열 system call은 인자 전달 방식 등에 따라서 execl(), execv(), execle(), execve(), execlp() 등 여러 가지가 존재함. 이를 호출하면 기존 process의 image를 새로운 program의 것으로 대체함.

3.4.2. Termination

1. Termination
process가 terminate되는 상황은 아래와 같이 3가지가 존재함.

1) exit() system call을 호출하는 경우. 이때 반환값은 exit()의 integer argument임.
2) main()이 return하는 경우. 이때 반환값은 main()의 integer 반환값임.
3) 특정 signal을 수신받았는데 이를 catch하지 못한 경우. 이때 반환값은 signal에 따른 특수한 값임.

2. Child Process의 종료
child process가 종료되면 parent process에서는 reaping이 수행됨. 수확(reaping)은 child process가 종료된 이후 parent process가 해당 child process의 종료 상태를 전달받는 것을 의미함. 종료 상태를 전달받은 parent process는 해당 child process의 system resource를 반납함.

wait() system call을 통해 reaping을 수행할 수 있음. wait()을 호출하면 child process가 종료될 때까지 대기하고, 이에 따라 실행 순서가 deterministic이 됨.

Orphan Process는 parent process가 먼저 종료된 child process임. parent process가 먼저 종료되면 orphan process는 kernel에 의해 intit의 child process로 입양되고, 이후 reap됨. init의 주요 작업이 orphan process를 handling하는 것이라고 함.

Zombie Process는 종료되었지만 parent process에 의한 reaping이 수행되지 않은 process임. child proess는 종료 직후 zombie process가 됨. zombie process는 system resource를 잡아먹음.

4. I/O

4.1. I/O

I/O에 대해 더 자세히 알아보자.

4.1.1. File

1. File
Unix에서 file은 바이트의 나열인데, 실제로 바이트의 나열인 것 외에도 device, service 등 모든 것은 file로서 처리됨.

file은 그 역할에 따라 regular type, directory, symbolic link, device 등 여러 type으로 구분됨.

1) Regular Type : 데이터를 포함하는 기본적인 형태의 file. application 수준에서 regular file은 text

file(ASCII, unicode 등)과 binary file로 구분될 수 있는데, 당연하게도 kernel은 이 둘은 구분하지 않음.

file에서 read/write 중인 특정 위치를 Current File Position이라고 함.

참고로 text file에서 EOL(End-of-Line)을 나타내는 것으로 unix 계열 os에서는 LF(\n)를, windows에서는 CRLF(\r \n)를 사용함.

2) Directory : 다른 file에 대한 link들로 구성된 file. 각 directory는 .과 ..에 대한 link를 기본적으로 가지고 있음. directory에 대해서는 /로부터 시작하는 계층 구조가 존재하고, kernel은 각 process에 대해 cwd(current working directory)를 관리함.

2. File Descriptor

File Descriptor는 특정 process에서 open 상태의 file을 나타내는 integer임. 표준 file descriptor로는 stdin(0), stdout(1), stderr(2)가 있고, 이는 기본적으로 모든 process에 대해 존재함.

3. File Mode

각 file은 owner, group, other(기타)에 대해 File Mode라는 권한 정보를 가짐. 이는 rwxrwxrwx와 같이 read, write, execute에 대해 총 9개의 mode bit로 표현됨. 또한 아래와 같이 3비트씩 묶어서 8진수로 나타낼 수 있음.

750 (111101000b): rwxr-x---
-> User can read, write, execute; group can read and execute
others have no access.
664 (110110100b): rw-rw-r--
-> User and group can read and write, others can read.

4.1.2. Unix I/O

POSIX에 의해 정의되는 Unix I/O에 대해 알아보자.

1. File 조작

아래와 같은 함수들로 file을 열고 닫고, 읽고 쓸 수 있음.

1) Opening

open() 또는 creat()으로 file을 열 수 있음. 이 함수들은 연 파일의 file descriptor를 반환함.

```
#include <fcntl.h>
int open(const char *path, int flags, mode_t mode);
int creat(const char *path, mode_t mode);
```

flag로는 아래와 같은 것들을 사용할 수 있음. O_CREAT | O_EXCL과 같이 비트 연산자 or로 여러 flag를 지정함. 이때 O_CREAT | O_EXCL와 같이 지정하면 호출하는 system call에 의해 file open이 atomic하게 처리되며, exclusive access를 구현할 수 있음.

The flags parameter controls how open() behaves:
- O_RDONLY: Open read-only
- O_WRONLY: Open write-only
- O_RDWR: Open for reading and writing
- O_CREAT: When writing, create the file if it doesn't exist
- O_EXCL: When creating a file, fail if it already exists (Exclusive)
- O_APPEND: When writing, start at the end of the file
- O_TRUC: When writing, truncate the file to 0 bytes
- O_CLOEXEC: Close this file on exec()

2) Reading/Writing

read()로 file을 읽고, write로 file에 쓸 수 있음.

read()는 단순히 raw byte로 읽어 저장함. 읽을 byte 수를 반환하고, EOF이면 0을 반환함.

```
#include <unistd.h>
int read(int fd, void *buffer, size_t bytes);
```

write() 또한 단순히 raw byte로 씀. 작성한 byte 수를 반환함.

```
#include <unistd.h>
int write(int fd, const void *buffer, size_t
bytes);
```

3) Closing
close() file을 닫을 수 있음. 닫힌 descriptor를 활용하려고 하면 error가 발생함. 닫힌 file descriptor는 이후에 다른 file에 대해 다시 활용될 수 있음.

```
#include <unistd.h>
int close(int fd);
```

4) Current file position 지정
lseek()로 current file position을 지정할 수 있음.

2. Error 처리
unix I/O 함수에서 error가 발생하면, 해당 함수는 negative integer를 반환하고, 해당 error의 번호를 errno라고 하는 global variable에 저장함. 이는 errno.h를 include하여 활용할 수 있음.

또한 perror(), strerror()를 사용하여 errno에 대응되는 식별 가능한 설명을 stderr에 출력하도록 할 수 있음.

```
if ((bytes = read(fd, buf, sizeof(buf))) < 0)
{
     perror("read"); ← read: Input/output error
     exit(1);
}

if ((bytes = write(fd, buf, sizeof(buf)) < 0)
{
   perror("write"); ← write: Input/output error
   exit(1);
}
```

대부분의 system call에서도 error가 발생하면 negative integer를 반환함.

4.1.3. C Standard I/O

C standard I/O에 대해 알아보자.

c standard I/O에서는 file descriptor에 대한 wrapper인 Stream을 활용함. 이는 저수준인 file descriptor를 추상화하여 오류 처리, buffering 등의 안정성과 편의성을 제공함. stream은 FILE형을 사용함.

1. File 조작
아래와 같은 함수들로 file을 열고 닫고, 읽고 쓸 수 있음.

1) Opening
fopen(), fdopen()으로 file을 열고 stream을 생성할 수 있음. fopen()은 단순히 stream을 생성하고, fdopen()은 인자로 지정된 file descriptor를 wrapping하여 stream을 생성함.

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
```

각 함수에서는 아래와 같이 mode를 지정하여 file을 열 수 있음. 이때 truncation은 기존 내용을 지우는 것을 말함. c standard I/O에서는 기본적으로 file을 text file로 취급하여 문자열로 읽고 쓰는데, mode에 b를 붙여 binary file로 취급하여 읽고 쓸 수 있음. (unix I/O에서는 전부 binary file로 취급함.)

write을 위한 mode에서는 기본적으로 file이 존재하지 않으면 생성함.

- "r": reading
- "w": writing, with truncation
- "a": writing, without truncation (append)
- "r+": reading and writing, without truncation
- "w+": reading and writing, with truncation

2) Reading/Writing
fread()와 fwrite()를 사용해 binary로 읽고 쓸 수 있음. item의 size와 개수를 지정함. 이후 읽거나 쓴 item의 개수를 반환하고, EOF의 경우 0을 반환함.

```
#include <stdio.h>
size_t fread(void *dest, size_t size, size_t nmemb, FILE *fp);
size_t fwrite(const void *buf, size_t size, size_t nmemb, FILE *fp);
```

3) Closing
fclose()를 사용해 stream을 닫고, 포함하고 있던 file descriptor를 닫고 buffer를 해제함.

2. Error 처리
c standard I/O에서는 error가 발생하면 0을 반환함. 즉, error는 EOF인 경우와 동일한 반환값을 가짐. 이에 따라 feof()를 사용하여 지정한 stream에서 EOF가 발생했는지를 확인하거나, ferror()를 사용하여 지정한 stream에서 error가 발생했는지를 확인할 수 있음. 각각 EOF/error가 발생했으면 true(non-zero)를 반환하고, 발생하지 않았으면 false(0)을 반환함.

- `int feof(FILE *fp);`
- `int ferror(FILE *fp);`

feof()/ferror()는 EOF/error 발생 시에 지정된 stream의 status를 확인하는 함수인데, clearerr()는 ELF/error에 대한 status를 초기화함.

- `void clearerr(FILE *fp);`

3. Buffering
system call은 비싼 작업이므로, 모든 입출력에 대해 매번 system call을 활용하는 것은 비효율적임. 이에 따라 c standard I/O에서는 read, write 작업에 대해 buffering을 적용함.

1) read 시의 buffering
fread() 등으로 파일에서 read할 때, 한 번에 여러 바이트(full disk block)를 읽어 buffer에 저장함. 이후 함수 호출마다 buffer에서 하나씩 꺼내 반환함.

2) write 시의 buffering
fwrite() 등으로 write할 때 즉시 file에 쓰는 대신, 내부 buffer에 해당 내용을 저장해뒀다가 buffer가 가득 차거나 fflush()/fclose()를 호출하면 실제로 file에 씀. 이에 따라 여러 번의 read/write에 대해 성능이 크게 향상될 수 있음. fflush()는 buffer의 내용을 즉시 쓰기 위해 사용하는 함수임.

```
int fflush(FILE *fp);
```

POSIX I/O은 buffering을 활용하지 않음.

4.2. Pipe and Redirection

4.2.1. Pipe

1. Pipe
Pipe는 file-like abstraction으로, file descriptor로 관리되며 unix I/O 함수를 사용해 한쪽 끝에 데이터를 write하면 반대쪽 끝에서 해당 데이터를 read할 수 있음.

pipe는 IPC 중 하나임. IPC(Interprocess Communication)로는 pipe, socket, shared memory, signal, environment variable등이 존재함.

pipe는 redirection을 통해 임의의 file descriptor에 연결될 수 있음. 특히 stdin과 stdout에 연결하여

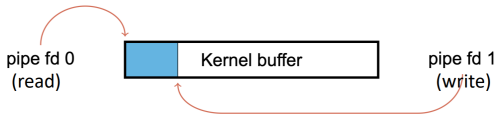
한쪽 process의 출력이 pipe로 연결된 다른 process의 입력으로 들어가도록 할 수 있음. unix 계열 os의 shell에서 |로 제공하는 pipe가 이렇게 구현됨.

2. 동작 원리

pipe는 아래와 같이 pipe() system call을 사용해 생성할 수 있음. pipe()는 두 개의 file descriptor를 생성해 pipe에 대한 read/write를 수행할 수 있도록 함. 첫 번째 file descriptor는 read-only이고, 두 번째 file descriptor는 write-only임. (stdin과 stdout의 순서와 동일함.)

```
int pipefd[2], rval, wval = 42;

pipe(pipefd);  <-- Create a pipe on the fd array.

write(pipefd[1], &wval, sizeof(wval));
read(pipefd[0], &rval, sizeof(rval));

printf("%d\n", rval);
Output:

42
```

pipe는 해당 file descriptor로 접근되는 kernel buffer임. 즉, read file descriptor는 read pointer를 가지고, write file descriptor는 write pointer를 가짐. read는 buffer에서 해당 pointer 위치에 데이터를 채우고, write는 buffer에서 해당 pointer 위치의 데이터를 꺼냄. 이는 아래 그림과 같이 동작하지만, 실제로는 단순 circular queue라고 함.



write에 대한 open file table이 닫히면(가리키는 file descriptor가 모두 닫힘.), pipe에 대한 read 시에 EOF가 반환됨.

3. Deadlock in Pipe

pipe에 대한 read/write는 기본적으로 block operation임. 즉, pipe의 buffer가 꽉 차있으면 write 시에 block되고, buffer가 비어 있으면 read 시에 block됨. 이에 따라 반대편의 read/write이 수행될 때까지 계속 block되는 deadlock이 걸릴 수 있음. (물론 엄밀하게는 deadlock이 아니고 그냥 block임.)

아래와 같이 fcntl() 함수(file control)를 활용해 pipe를 nonblock으로 지정하면 deadlock을 방지할 수 있음. 이렇게 nonblock으로 지정하면 read/write 시에 block되는 대신 negative integer를 반환하고 errno에 EAGAIN가 지정됨.

```
#include <fcntl.h>

fcntl(pipe[0], F_SETFL, O_NONBLOCK);
fcntl(pipe[1], F_SETFL, O_NONBLOCK);
```

4. Safe Pipe Usage

fork() 시에는 아래에서 설명할 descriptor table이 복사되므로, parent와 child가 모두 해당 pipe에 대한 file descriptor를 가지고 있음.

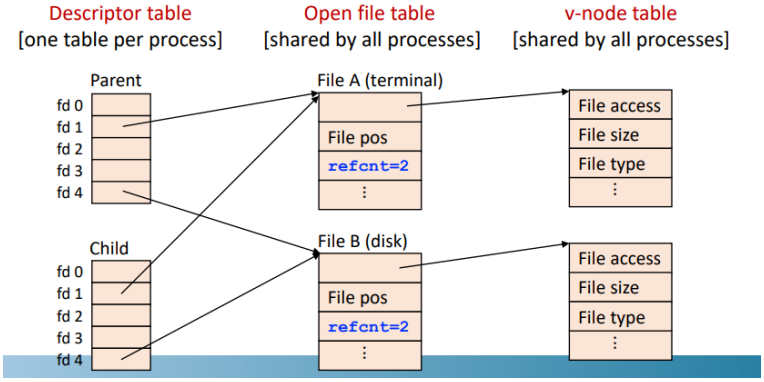
이에 따라 아래와 같이 pipe를 안전하게 사용하기 위해서는 write하는 쪽에서는 read를 닫고, read하는 쪽에서는 write를 닫는 것이 안전함. fork()에 의해 해당 pipe의 양쪽 끝에 대한 open file entry는 reference count가 각각 1씩 늘어나게 되므로, pipe가 닫히지 않음.

```c
int pipefd[2], pid;
char buf[6];

pipe(pipefd);
if ((pid = fork()) == 0) {
    close(pipefd[0]);
    write(pipefd[1], "Hello", 6);
} else {
    close(pipefd[1]);
    read(pipefd[0], &buf, 6);
}
```

The child process closes the pipe output, and
the parent process closes the pipe input.

4.2.2. Unix에서의 Open File 관리

unix에서는 아래와 같이 descriptor table, open file table, v-node table을 활용해 open file을 관리함.



1. Descriptor Table
Descriptor Table은 각 process가 가지는 file descriptor를 저장하는 table로, 각 process는 개별적인 descriptor table을 가짐. 위의 그림에서와 같이 file descriptor는 open file table에 대한 index로, open file에 대한 간접적인 접근을 지원함.

stdin, stdout, stderr는 기본적으로 terminal을 가리킴.

2. Open File Table
Open File Table은 open file에 대한 metadata를 저장하는 table로, process들에 대한 global table임. current file position(current pointer. cp.), filesystem, disk location, file mode, reference count 등에 대한 정보를 포함함.

open file table은 각 open file에 대한 Open File Entry를 가짐. 사용자가 file 앞에서 다룬 것처럼 여기에서 file로는 file(disk), terminal, pipe 등이 존재함. 또한 동일한 file(disk)에 대해 서로 다른 open file entry가 존재할 수 있음.

reference count는 해당 open file entry를 몇 개의 file descriptor가 참조하고 있는지를 나타내는 값임. reference count가 없으면 하나의 file에서만 file을 닫아도 연결이 닫혀버리는 문제가 발생함.

fork() 시에는 descriptor table이 복제되므로, 동일한 open file table의 entry를 가리키게 됨. 즉, parent와 child는 current file position 등 open file table이 가지는 file 관련 정보를 공유하게 됨. 이에 따라 parent와 child의 read/write가 서로 영향을 주게 됨.

3. v-node Table
v-node는 inode 등에 대한 abstraction으로, 모든 종류의 file을 공통적으로 처리하기 위한 것임.

v-node가 추상화하는 inode는 실제 disk에 대한 metadata를 포함함.

4.2.3. Redirection

dup()과 dup2()를 활용해 file descriptor를 복사할 수 있고, 이를 활용해 redirection을 구현할 수 있음.

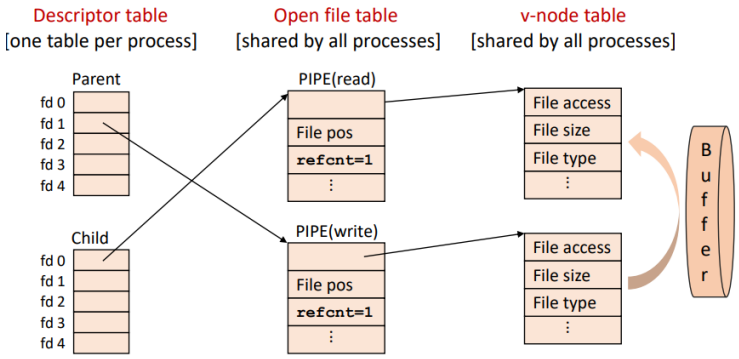
```
#include <unistd.h>
int dup(int fd);
int dup2(int oldfd, int newfd);
```

dup()는 인자로 넣은 file descriptor를 복사해 새로운(다른 번호의) file descriptor를 반환함. 이때 file descriptor만 복사한 것이므로 해당 file descriptor들은 동일한 open file entry를 가리킴.

dup2()는 첫 번째로 작성한 file descriptor의 open file entry를 두 번째로 작성한 file descriptor에 복사함. 두 번째로 작성한 file descriptor가 어떤 open file entry를 가리키고 있었다면, 기존의 연결이 끊어지고 새로운 open file entyr를 가리키게 됨. 아래는 그 예시임.

```
int fd;

fd = open("output.txt", O_WRONLY|O_CREAT, 0666);

dup2(fd, 1);
close(fd);

puts("Redirected output!");
```

shell command에서 사용하는 pipe(|)는 아래와 같이 구현됨. 즉, stdin과 stdout을 pipe에 대한 open file table로 redirection함.



5. Shell

5.1. Unix Shell

5.1.1. Unix Shell

1. Shell

Unix Shell은 unix 계열 os에 대한 primary user interface임. shell에 대해서는 아래와 같이 두 관점으로 생각할 수 있음.

1) Interactive Shell : shell은 interactive하게 prompt를 입력하고 즉시 실행함. 또한 command 실행에 대한 효율성을 확보하기 위해 alias, history 등의 기능을 제공함.

2) Programming Environment : shell은 variable, loop, procedure, exception 등을 지원하며 prgramming environment로도 기능함.

2. Word와 Statement

shell은 입력 string을 whitespace를 단위로 나누는데, 나뉘어진 각 string을 Word라고 함. command에서의 첫 번째 word는 해당 command가 어떤 것인지를 나타냄. whitespace는 따옴표나 역슬래시로 escape할 수 있음.

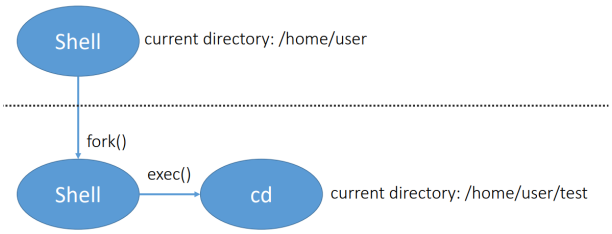
shell script 내에서 Statement는 이전 command 바로 다음부터 newline/;/& 까지의 부분임. shell은 shell script를 statement 단위로 parsing한 뒤, 각 statement가 아래와 같은 4가지 종류 중에 어떤 것에 속하는지 판단함.

1) variable assignment
2) builtin command
3) control statement(ex. if, while)
4) external program

5.1.2. Builtin Command

Builtin Command는 shell 내부에 내장된 command임. 아래와 같은 이유로 shell에서는 builtin command를 활용함.

1) efficiency. 매번 fork()하고 exec()하는 것은 비용이 많이 듦.
2) shell의 internal state를 수정해야 함. 아래와 같이 fork()한 뒤 작업을 수행하면 그 결과가 shell에 반영되기 어려움(cd는 실제로 builtin command임.).



builtin command가 아닌 External Command는 fork()와 exec()을 통해 실행함.

5.1.3. Variable

1. Variable
shell에서 variable은 string임. 또한 variable은 global이며 별도로 정의되는 대신 바로 VAR=value 형태로 값을 할당해 활용함. 이때 = 앞뒤로는 공백 문자가 존재하지 않아야 함.

shell은 아래와 같은 special variable을 지원함.

The shell recognizes quite a few special variables, including:
- $0: the name of the current executable
- $1-$9: the first 9 arguments to the shell (or a function)
- $#: The number of arguments $1-$9 that are valid
- $* and $@: All of the arguments to the shell (or a function)
- $?: The return value of the previous command
- $!: The process ID of the previous command
- $PS1: The prompt given in interactive use
- $IFS: The input field separator used to determine if an expansion creates new words

이때 IFS(input field seperator)는 string을 word로 분리하거나 필드를 구분할 때 사용하는 구분 문자를 지정하는 변수임. IFS의 default 값은 newline, tab, 공백 문자임. 예를 들어, argument가 아래와 같이 지정되는데, IFS를 지정하여 그 구분 문자열을 다른 것으로 할 수 있음.

```
$ VAR="arg1 arg2"
$ ./writeargs $VAR
arg1
arg2
```

2. Interpolation
variable은 $VAR 또는 ${VAR}으로 그 값을 command에서 활용할 수 있음. ${VAR}은 문자열 등과 붙여서 사용하는 경우 variable 이름을 명확히 구분하기 위한 것임.

command도 $(command)로 그 값을 다른 command에서 활용할 수 있음.

3. Environment Variable
Environment 또는 Environment variable은 key-value 쌍임. 모든 POSIX process는 environment를 가지고, parent의 environment를 상속받음.

variable은 기본적으로 shell에 대해 private한데, export builtin command를 사용해 environment로 등록하면 global하게 사용이 가능함. export VAR과 같이 작성함

env command를 사용해 environment 목록을 출력할 수 있음.

5.1.4. Globbing

Globbing은 shell에서 file이나 directory 이름을 pattern matching을 통해 일괄적으로 지정하는 기법임. shell에서는 아래와 같은 globbing을 지원함.

*1) * : 임의의 문자/길이를 가지는 문자열과 matching. 문자열의 길이가 0이어도 matching됨.*
2) ? : 임의의 문자 1개와 matching.
3) [] : 지정한 범위 내의 문자 1개와 matching. (ex. [a-z], [ch])

5.1.5. Pipe and Redirection

아래와 같은 기호로 pipe(pipe())와 redirection(dup2())을 활용할 수 있음.

1) | : pipe.
2) < : stdin을 지정한 file에 연결함.
3) > : stdout을 지정한 file에 연결함. 기존 file 내용을 삭제함.
4) 2> : stderr를 지정한 file에 연결함. 임의의 정수를 지정해 해당 file descriptor와 연결할 수 있음.
5) » : stdout을 지정한 file에 연결함. 기존 file 내용 뒤에 작성함.
6) N>&M : dup2(N, M)을 호출하여 file descriptor를 복사함. 이는 특정 command에 대해 작성하는데, 예를 들어 ls /invalid_dir 2>&1와 같이 작성할 수 있음.

pipe의 경우, cmd1 | cmd2와 같이 작성함. 이때 내부적으로는 pipe()로 pipe를 생성하고, cmd1, cmd2에 대한 fork()가 수행된 뒤, dup2()로 file descriptor를 연결함. 이후 exec()으로 작업을 수행하고 출력이 입력에 전달됨.

6. Memory Management

6.1. Virtual Address

6.1.1. Virtual Address

1. Address Space
Address Space는 process에서 참조할 수 있는 주소의 범위 또는 집합임. 이에 대한 범위는 시스템에 따라 결정됨.

컴퓨터 시스템에서 사용하는 address space는 linear address space임. 즉, 0부터 시작하는 non-negative integer set임. 당연하게도 n개의 비트를 사용하는 경우 $\{0, 1, \cdots, 2^n - 1\}$임.

2. Physical Address vs. Virtual Address
Physical Address(PA)는 memory에서 물리적으로 사용되는 실제 주소이고, Virtual Address(VA) 또는 Logical Address는 process 관점에서 사용하는 가상의 주소임.

초창기 시스템에서는 각 program은 compile time에 PA가 결정되었음. 이 방식으로는 여러 program을 동시에 memory에 load하는데에 한계가 있었는데, 초기에 사용되었던 MS-DOS 같은 os는 CLI 기반이어서 큰 문제가 없었음. windows 등 GUI가 등장하며 multiprogramming의 필요성이 생기자 VA를 사용하기 시작했음. 현재 대부분의 상용 시스템에서는 VA를 사용하고, 일부 embedded microcontroller와 같은 단순한 시스템에서는 PA를 사용하기도 함.

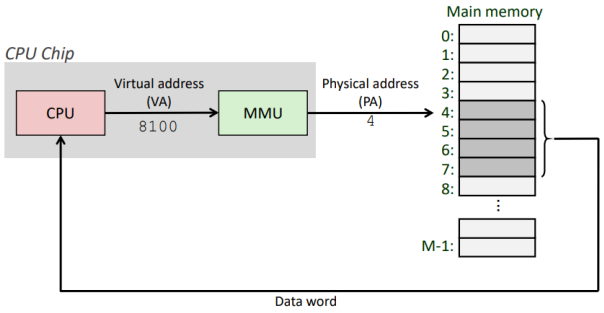
PA와 VA의 변환은 MMU가 수행함.

6.1.2. MMU

MMU(Memory Mangement Unit)는 VA과 PA 간의 address translation을 수행하는 하드웨어 장치로, cpu 안에 위치함.

cpu가 VA를 활용해 특정 부분에 접근하려 할 때마다 MMU는 해당 주소를 중간에 가로채 PA로 변환하고, cpu는 PA를 활용해 접근하게 됨. memory 접근은 빈번히 발생하는 작업이므로 MMU는 성능에 미치는 영향이 큼.

MMU는 뒤에서 설명할 page table을 통해 translation을 수행함.



6.2. Virtual Memory

6.2.1. Virtual Memory

1. Virtual Memory

Virtual Memory(VM) 또는 Logical Memory는 VA로 구성된 가상의 memory로, 각 process가 활용하는 memory임. 반면 Physical Memory(PM)는 PA로 구성된 memory로, 실제 하드웨어적으로 존재하는 memory임.

VM의 사용은 아래와 같은 이점이 존재함.

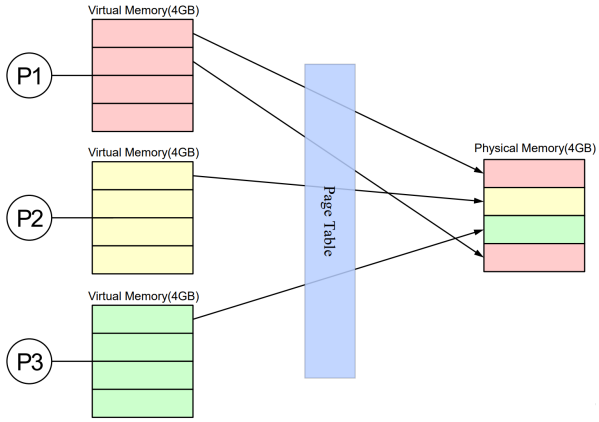
1) VM은 PM보다 더 큰 크기를 가질 수 있으므로, main memory(PM)의 물리적인 한계를 극복함. 이는 lazy allocation(demand paging)으로 구현됨.

2) 각 process는 각각 동일한 형태의 memroy view를 활용할 수 있음.

3) process 별로 독립적인 memory 공간을 가지도록 할 수 있음.

process가 가지는 memory layout의 측면에서, 각 process는 전체 memory space에 접근 가능해야 하고, 다른 process의 memory에 접근할 수 없어야 하고, dynamic library는 여러 process들에이 활용할 수 있어야 함. 이는 모두 VM에 의해 구현됨.

RAM이 가지는 physical layout의 측면에서, 일부 영역은 접근이 불가능해야 하고, memory의 특정 부분에 대한 역할이 정해져 있을 수 있고, 각 시스템 별로 다른 크기의 RAM을 가질 수 있음. VM은 각 program이 이런 physical layout에서의 세부 사항을 고려하지 않도록 함. 물론 kernel에서는 이를 여전히 고려해야 함.



2. VM 접근 과정
VM 접근 과정은 다음과 같음. 각 process는 memory 접근 시에 VM에 접근하는 것으로 가정하고 동작함. VA를 사용해 VM에 접근하려고 하면, MMU는 page table을 사용해 해당 VA와 매핑된 부분이 PM에 올라와 있는지를 확인함. PM에 올라와 있으면(page hit) translation을 수행 후 접근하고, 올라와 있지 않으면(page fault) 해당 부분을 PM에 올린 뒤 translation하여 접근함.

6.2.2. VM 활용의 여러 관점

VM의 활용은 아래와 같이 여러 관점에서의 이점이 존재함.

1. Caching
PM은 제한되어 있으므로, VM을 통한 caching을 활용하면 PM을 효율적으로 사용할 수 있음. 즉, VM은 damand paging으로 필요한 page만을 PM에 올리고, 필요 없는 page는 disk로 내리는 처리를 통해 효율적인 caching을 구현함.

Working Set은 특정 시간 간격 동안 참조한 page number를 모은 집합임. (집합이므로 중복 원소를 가지지 않음.) working set은 locality를 나타냄. working set의 크기가이 PM의 크기보다 작다면 locality가 잘 활용되어 page fault가 덜 발생한 것이고, PM보다 크다면 page fault가 많이 발생한 것으로 이해할 수 있음. 후자의 경우 thrashing이 발생했을 수 있음. Thrashing은 process의 전체 실행 시간에서 작업 시간보다 page fault 처리 시간이 더 긴 상황을 말함.

2. Memory Management
각 process별로 page table을 활용해 독립적인 memory 공간을 활용할 수 있음. 또한 PM을 잘게 쪼개어 활용할 수 있으므로 fragmentation을 줄일 수 있음. 각 process가 가진 page들을 동일한 physical page에 mapping하여 process 간의 code/data 공유를 간단하게 구현할 수 있음.

3. Memory Protection
process 별로 독립적인 memory를 활용할 수 있기도 하지만, page table의 flag bit로 permission bit들을 추가하여 더 효과적으로 memory를 보호할 수 있음.

6.3. Paging

6.3.1. Page

1. Page
Page 또는 Virtual Page는 VM에서 address space를 관리하는 단위임. 반면에 Frame 또는 Physical Page은 PM에서 address space를 관리하는 단위임. 즉, VM과 PM은 각각 page와 physical page이라는 고정된 단위로 memory를 나누어 관리함.

일반적인 상용 시스템에서 page와 physical page의 크기는 4KB임. 이때 page와 physical page, 그리고 disk 관리 단위의 크기는 동일함. (동일하지 않다면 구현이 복잡해짐.)

page와 physical page는 fully associative로 mapping됨. 즉, 임의의 위치가 page가 임의의 위치의 physical page와 mapping될 수 있음.

당연하게도 page에 대응되는 실제 데이터는 PM에 올라와 있을 수도 있고(page에 physical page가 할당된 상태라고 함.), disk에 있을 수도 있음.

2. Page Number/Address
Page Number는 process가 가진 page 각각에 부여된 번호로, 각 page를 구분하기 위한 번호임. Page Address 또는 Page Offset는 특정 page 내부에서의 구분을 위해 사용하는 주소임.

cpu가 관리하는 모든 주소는 page number와 page address로 나뉨. MMU는 VA를 page number와 page offset으로 분리하여 PA로 변환함.

page의 크기가 4KB이고 주소 하나가 1바이트 크기이면 총 2^{12}개의 주소가 있으므로, page 내부의 구분을 위해서 page offset으로는 12bit가 필요함. 하위 12bit를 제외한 나머지 bit는 page number로 사용됨. 예를 들어, page의 크기가 4KB이고 주소 하나가 1byte인 32bit system에서는 page number에 20bit가 사용되고, page offset에 12bit가 사용됨.

6.3.2. Page Table

1. Page Table
Page Table은 각 process 별 page 정보를 저장하는 table임. process들은 각각 개별적인 page table을 가짐. page table의 인덱스는 page number이고, 그 값은 PTE임.

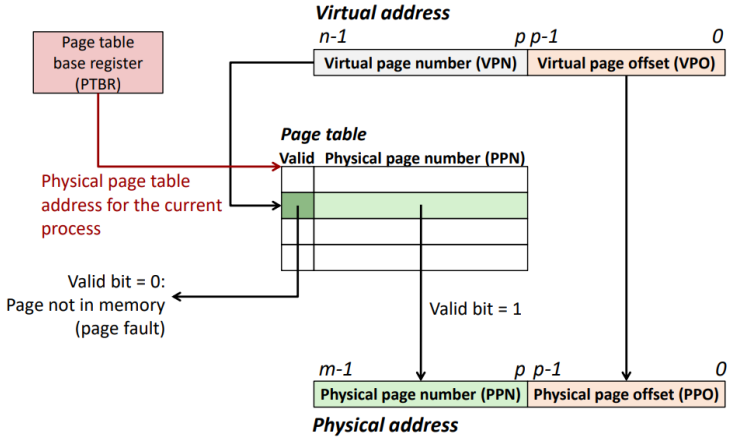
2. Page Table Entry
Page Table Entry(PTE)는 page table의 레코드(열)임. PTE는 page base address와 flag bit 등의 항목들을 포함함.

Page Base Address 또는 Physical Page Number는 해당 page에 할당된 physical page의 시작 주소임. physical page number의 bit 수는 PA의 전체 길이에서 page address만큼을 뺀 것임. 이때 당연하게도 PA의 전체 길이는 PM의 크기에 의해 정해지고, VA의 길이(32bit 등)와는 다를 수 있음.

Flag Bit는 해당 page의 상태를 나타내는 bit로, 다양한 것들이 있지만 기본적으로 사용되는 valid bit는 해당 page가 physical memory에 존재하는지를 나타냄. 1이면 존재하는 것, 0이면 존재하지 않는 것임.

3. PTBR
PTBR(Page Table Base Register)은 해당 process에 대한 page table에 대한 PA를 저장하는 register임. page table의 주소에 VM을 사용할 수는 없으므로, page table에 대한 PA를 저장하는 register를 사용하는 것임.

4. Address Translation
address translation은 다음과 같은 과정을 통해 수행됨. MMU는 VA를 page number와 page address로 분리하고, PTBR에 저장된 주소로 page table에 접근한 뒤 page number를 인덱스로 PTE를 찾음. 이후 PTE의 flag bit 등을 확인하고 해당 physical page number를 page offset과 결합하여 PA를 반환함.

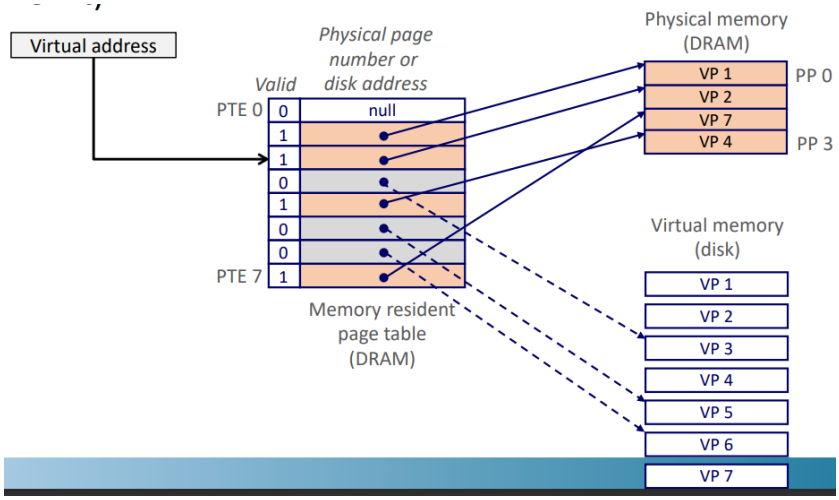


6.3.3. Page Hit/Fault

page는 Demand Paging 또는 Lazy Allocation으로 처리됨. 즉, 모든 page를 PM에 올리는 대신, 필요한 page만을 PM에 올림. 이에 따라 어떤 page에 접근했을 때 해당 page에 대응되는 부분이 PM에 존재하지 않을 수 있음. 이때 page hit 또는 page fault가 발생할 수 있음.

1. Page Hit

Page Hit(DRAM cache hit)은 page number를 index로 page table에 접근했는데 PTE가 PM에 존재하는 상황(valid)임. 즉, page에 해당하는 부분이 PM에 올라와 있는 것임. 이 경우 단순히 translation을 수행하고 그 결과를 cpu에게 넘겨주면 됨.

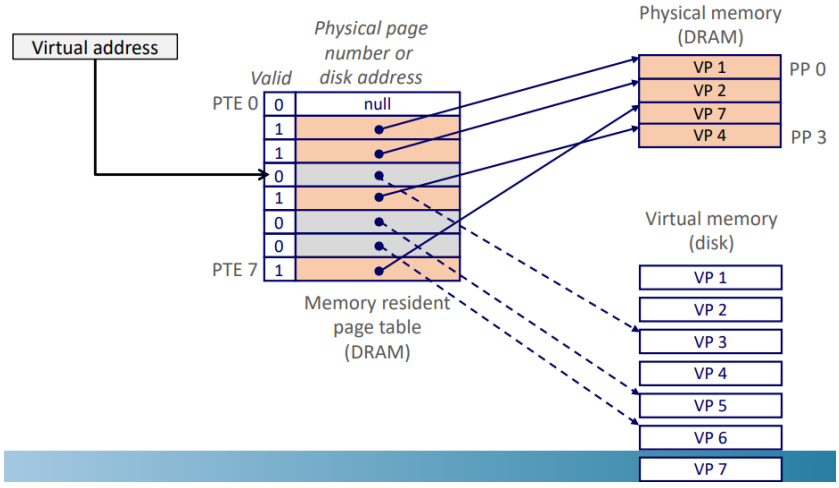


2. Page Fault

Page Fault(DRAM cache miss)는 page number를 index로 page table에 접근했는데 PTE가 PM에 존재하지 않는 상황(invalid)임. 즉, page에 해당하는 부분이 PM에 올라와 있지 않은 것임.

page fault가 발생하면 아래의 동작이 수행됨.

1) disk에서 page에 해당되는 내용을 찾음.
2) 해당 page에 free physical page를 할당함.
2-1) free physical page가 있다면 해당 physical page를 사용함.
2-2) free physical page가 없다면 page replacement를 수행함. 즉, victim physical page를 PM에서 disk로 내리고(swap out), page table에서 해당 physical page를 할당받았던 page를 invalid 처리함.
3) 요구 page에 해당되는 내용을 disk에서 empty physical page에 올림(swap in).
4) page table에서 요구 page를 valid 처리함.
5) process의 작업(instruction)을 재시작함.
6) page hit이 발생함.



이때 page replacement 알고리즘은 매우 복잡해서 하드웨어적으로 구현되기는 어렵다고 함.

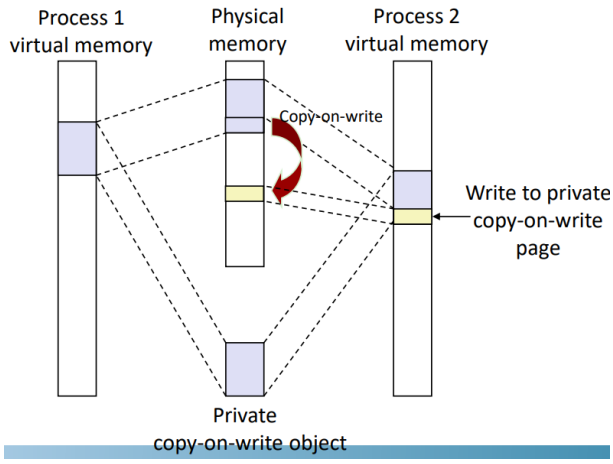
6.4. 기타 Memory 관련 기법들

6.4.1. COW

COW(Copy-on-Write)는 여러 process에게 참조되는 memory가 존재하면 각 process들은 memory 공간을 read-only로 접근하고, 이후 특정 process에서 해당 memory 공간에 대해 write을 시도하면 해당 memory 공간의 내용을 복사하여 새로운 공간을 할당하는 기법을 말함. 이를 통해 fork() 등에서도 memory를 절약할 수 있음.

즉, 아래와 같은 과정을 거쳐 수행됨.

1) 어떤 process가 특정 PM을 참조하고 있음.
2) 다른 process가 해당 PM을 참조하면 둘 다 read-only로 지정됨.
3) 해당 PM에 write하려고 하면 protection에 의해 page fault가 발생하고 handler가 호출됨.
4) handler는 page fault를 발생시킨 process에 대해, 해당 PM 공간의 내용을 복사하여 새로운 PM을 할당함.
5) page fault를 발생시킨 instruction을 다시 수행함.



fork() 시에 page table의 내용도 복사되고 동일한 PM을 가리키게 되는데, 이는 COW로 동작함.

6.4.2. Stack과 Memory

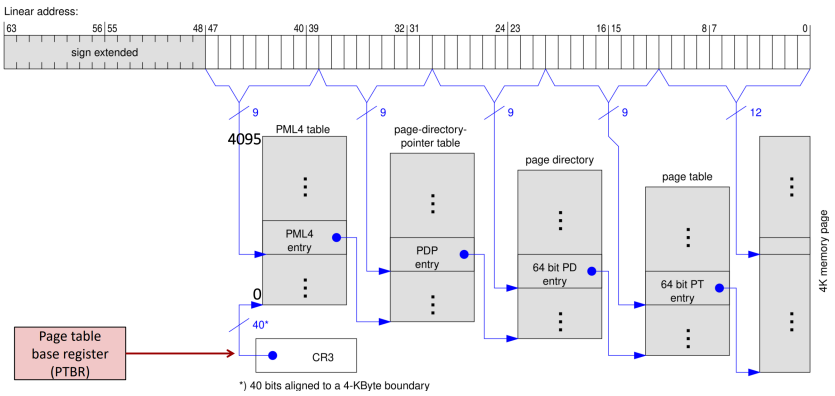
process의 stack 또한 VM으로 처리됨. 즉, process 생성 시에 kernel은 예상되는 stack 크기만큼 page table에 PM을 할당해 read/write가 가능하도록 지정해 놓음. 이후 stack이 자라고 page fault가 발생하면 새로운 physical page를 할당함.

6.4.3. Multilevel Page Table

1. Multilevel Page Table
Multilevel Page Table은 page table도 paging된 공간에 저장하는 기법임. 즉, page table을 여러 개의 작은 page table로 나누고, level을 부여하여 계층 구조로 구성함.

multilevel page table에서는 각 page table에 대한 PM을 필요할 때마다 할당하여 memory 사용을 최적화함. multilevel이 아닌 page table에서는 page table 전체를 PM에 올림. 반면 multilevel page table에서는 level 1 page table만을 PM에 올려 두고, 필요할 때마다 page table을 만듦. 구체적인 내용은 아래에 있는 2-level page table에서 정리함.

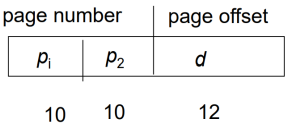
level이 더 많은 page table도 2-level page table과 동일한 원리로 동작함. 즉, VM을 각 level에 대해 나눠서 활용하고, page table의 크기를 최적화함.



2. 2-Level Page Table

2-Level Page Table은 2개의 level을 가지는 계층 구조의 page table임. 2-level page table에는 level 1 page table(page directory, outer page table)과 level 2 page table이 있음. 이 방식은 주로 32bit 시스템에서 사용함. 아래의 설명은 32bit 시스템과 4KB의 page 크기를 가정한 것임.

2-level page table에서는 VA의 page number(20bit)를 10bit씩 나누어 각각 level 1과 level 2에 대한 인덱스로 사용함. 각각을 page directory index, page table index라고도 함.



level 1 page table은 전체가 PM에 올라와 있음. 이 table은 VA의 첫 번째 10bit 값을 인덱스로 하고, level 2 page table의 frame number를 값(구현 방식에 따라 다를 수 있음.)으로 함. 각 크기가 4B(PA)인 2^{10}개의 레코드를 가지므로 총 4KB의 크기를 가짐. 이 크기는 frame 하나의 크기와 일치하므로 효율적임.

level 2 page table은 필요할 때마다 PM에 생성됨. 이 table은 VA의 두 번째 10bit 값을 인덱스로 하고, 구하려는 PA의 frame number를 값으로 함. 이 table도 마찬가지로 각 크기가 4B인 2^{10}개의 레코드를 가지므로 총 4KB의 크기를 가짐.

MMU는 VA를 받으면 TLB를 검사하고, 매핑이 없으면 PTBR에 저장된 PA를 사용해 level 1 page table에 접근함. 첫 번째 10bit를 인덱스로 level 1 page table에서 level 2 page table의 PA를 얻음. 이후 두 번째 10bit를 인덱스로 level 2 page table에서 frame number를 얻음. 이 frame number와 page address를 합쳐서 PA를 도출함.

7. Function

7.1. Function

7.1.1. Stack Manipulation

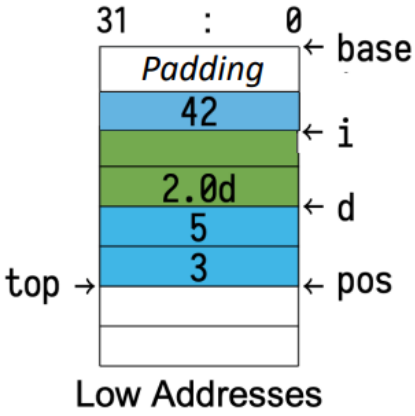
1. ABI

ABI(Application Binary Interface)는 binary level에서 program이 어떻게 상호작용하는지에 대한 specification임.

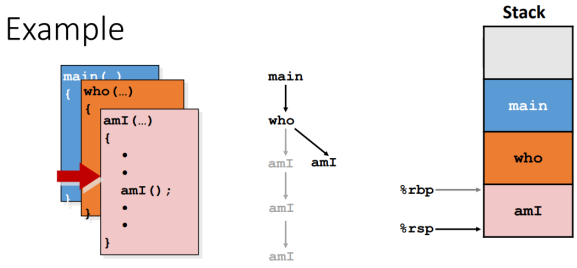
ABI는 cpu instruction set, calling convention, system call, object file format 등을 정의함. calling convention에서는 stack의 활용에 대한 내용도 포함함.

2. Stack

Stack은 memory에서 downward로(높은 address에서 낮은 address로) 자라고, push/pop 연산을 활용한 LIFO로 동작하는 자료구조임. 이때 stack(해당 stack frame)의 가장 아랫쪽(높은 주소)은 base pointer(rbp)로 가리키고, 가장 위쪽(낮은 주소)은 top pointer(rsp)로 가리킴.



rbp와 rsp는 function call에 따라 아래 그림과 같이 동적으로 변함.



stack은 implicit하게 할당됨. function call과 automatic variable은 stack을 활용함.

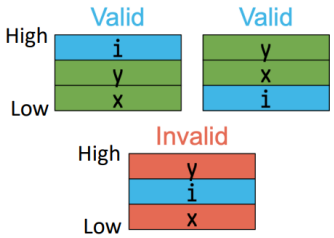
3. Automatic Variable
Automatic Vaiable 또는 Local Variable은 function 내부에서 정의된 variable임.

variable의 선언은 memory location에 이름을 붙이는 것이기도 하지만, compiler에게 stack을 reserve 하도록 하는 것이기도 함.

automatic variable은 적어도 처음 참조되기 전에 할당되는데, 대체로 해당 function 또는 block이 시작 되면 할당됨. 이후 적어도 해당 function 또는 block이 종료되기 전까지는 valid함.

automatic variable에 대한 할당 위치와 순서는 compiler의 알고리즘에 의해 결정되므로 프로그래머는 알 수 없음. 또한 memory가 아니라 register에만 저장될 수도 있음. 물론 구조체 등의 경우 해당 구조가 보존됨.

```
int i;
struct {
  int x; int y;
} pos;
```

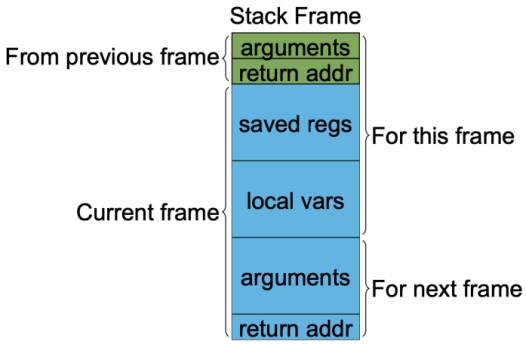


참고로 stack에 대한 pop시에 데이터를 지우지는 않고, 단순히 접근만 막음.

7.1.2. Stack Frame

1. Stack Frame
Stack Frame은 하나의 function invokation을 위한 정보를 포함하는 memory block으로, 아래와 같은 정보를 포함함.

1) saved processor register.
2) local variable.
3) arguments.
4) PC.



Function Call은 기본적으로 새로운 location으로의 jump, function code execution, calling location 으로의 jump로 이해할 수 있음. 물론 이 과정을 포함하여 variable 할당, 또 다른 function의 호출, register 등 여러 작업을 더 수행하는데, 이런 작업에는 stack frame이 활용됨. stack frame의 활용에 따라, function은 각 function들을 순차적으로 호출하고, 한 번에 하나의 function만 호출할 수 있음.

stack frame을 활용한 function call의 과정은 아래와 같음. 이때 push하는 과정은 prologue, pop 및 memory를 반납하는 과정을 epilogue라고 함.

1) caller의 PC를 stack에 push함.
2) callee가 활용할 argument를 stack에 push함.
3) callee를 호출(jump)함.
4) callee가 local variable 등을 위한 memory를 할당함.
5) callee의 작업이 수행됨.
6) callee는 memory를 반납하고, PC 값을 pop해 원래의 위치로 jump함.

stack frame의 각 요소에 대해 더 살펴보자.

2. Local Variable
대체로 function에 대한 모든 local variable들은 한 번에 할당됨. function이 호출되면 stack의 top부터 local storage로 필요한 만큼을 할당받고, 이후 이 크기는 고정됨.

이때 funcation의 local variable들에 대해 할당된 memory 크기는 저장되지 않지만, 각 local variable에 대해 stack top으로부터의 거리를 저장해두고 이를 활용해 참조됨.

3. Argument
x86-64 시스템에서는 function 호출 시에 앞쪽 6개의 argument를 register에 저장하고, 그 이후의 argument들은 stack에 저장함. 이에 따라 대부분의 function이 가지는 argument들은 register에 저장됨. 이때 앞쪽 argument가 가장 위로 가도록 argument들은 역순으로 stack에 push됨.

추가로, 이후 function의 반환값도 register(rax)에 저장됨.

caller는 argument를 stack에 넣고, callee를 호출함. 이후 callee가 종료되면 caller는 argument passing 에 활용되었던 부분을 clean up함.

4. PC
PC(Program Counter)는 processor가 현재 실행 중인 instruction의 address를 나타내는 값임.

function call 시에 caller의 PC 값이 stack에 push되고, 이후 callee가 호출됨. return 시에 이 PC 값을 pop해 원래의 위치로 jump함.

8. Signal

8.0.1. Signal

1. Signal
POSIX에서 Signal은 asynchronous message로, IPC 중 하나임. 각 signal은 번호를 가짐.

signal은 두 processer가 가지는 instruction 위치에서 주고받을 수 있고, 전달된 signal은 proecss를 terminate하거나, catch되거나, block되거나 ignore됨.

signal로는 아래와 같은 것들이 있음.

1) SIGHUP(1) : terminal이 disconnect되면 전송됨. (signal hang up)
2) SIGINT(2) : ctrl+c를 누르면 전송됨. (signal interrupt)
3) SIGKILL(9) : process 종료 시에 전송됨. catch할 수 없음.
4) SIGSEGV(11) : invalid memory access 시에 전송됨.
5) SIGCHLD(17) : child process가 exit한 경우 전송됨.

2. Sending Signal
kill()로 특정 process에 signal을 전송할 수 있음.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

3. Blocking Signal
signal은 process에서 block될 수 있음. signal은 해당 signal에 대한 handler가 이미 실행 중인 경우 implicit하게 block될 수 있고, 아래와 같이 sigprocmask()에 의해 explicit하게 block/unblock될 수 있음.

```
sigset_t mask, oldmask;

sigemptyset(mask);
sigaddmask(mask, SIGCHLD);
sigprocmask(SIG_BLOCK, &mask, &oldmask);

// SIGCHLD is blocked here

sigprocmask(SIG_SETMASK, &oldmask, NULL);

// SIGCHLD restored to its state before the block
// If it is unblocked and pending, the handler will run now
```

signal은 아래와 같은 종류가 존재함.

1) Reliable Signal
2) Real-time Signal : 데이터를 포함할 수 있는 signal.

signal은 scheduling에서도 활용될 수 있음. scheduling은 process에 대한 직접적인 함수 호출을 통해 수행되는 Explicit Scheduling과, IPC 등을 활용하는 Implicit Scheduling으로 나뉠 수 있음.

8.0.2. Signal Handler

1. Signal Handler
전달된 signal은 signal에 대해 정의된 Signal Handler에 의해 처리됨. 각 signal을 default handler를 가지는데, signal을 무시하거나, process에 대한 stop/continue/terminate 등을 수행함.

signal handler는 독립된 control flow로서 동작함. 이에 따라 signal에 의한 shared data 접근도 가능은

하지만, signal은 실행 시점이 asynchronous하므로 적절하지 않음.

2. Signal Handler 등록
*signal handler는 function으로, 아래와 같이 signal에 대해 등록할 수 있음. 이때 SIGKILL과 SIGSTOP에 대해서는 등록이 불가능함. 이때 sighandler_t는 함수 포인터 타입으로, typedef void (*sighandler_t)(int);와 같음. signal()의 반환값은 기존에 존재하던 signal handler임.*

```
sighandler_t signal(int signum,
                    sighandler_t handler);
```

아래와 같은 특별한 signal hander가 정의되어 있는데, 이를 signal()의 handler에 지정할 수 있음.

1) SIG_IGN : signal을 ignore함.
2) SIG_DFL : default signal handler를 복원함.

signal()은 시스템에 따라 default handler로 초기화되거나 handler 동작 도중에 reentrance가 수행되는 등의 portability 문제가 존재함. 이에 따라 sigaction() 등을 대신 사용하기도 함.

3. Signal Reception
signal을 전달받으면 아래와 같은 작업이 수행됨.

1) PC를 stack에 push함.
2) signal handler로 jump하여 실행함.
3) PC를 pop하여 return함.

실제로는 아래와 같은 상황에 kernel은 signal을 전달함.

1) kernel mode에서 user mode로 넘어가는 경우.
2) process가 idle/wait state에서 run state가 되는 경우.
3) blocked signal의 block이 풀린 경우.

shell에 대해서 foreground job은 쉽게 처리가 가능한데, background job의 경우 종료되는 시점 등을 알 수 없으므로 user가 이를 처리할 수 없음. 이에 따라 kernel은 background job 종료 시에 signal을 전송함.

8.0.3. Nonlocal Jump

Nonlocal Jump는 function call이 너무 깊어지는 경우 등에서 예외 상황이 발생하면 전부 반환하는 과정을 거치는 대신 한 번에 jump하는 것임. signal은 예외 처리에도 꽤 사용되는데, nonlocal jump도 그런 용도로 사용할 수 있음. 즉, context를 user level에서 저장해 활용하도록 함.

이는 setjmp와 longjmp로 구현됨.

1) int setjmp(jmp_buf j)는 longjump 이전에 한 번 호출되어, return site을 식별하도록 함. 해당 시점의 stack frame 정보를 jmp_buf에 저장해 인자로 넣음.

2) void longjmp(jmp_buf j, int i)는 setjump 이후에 호출되어, i에 해당되는 값을 반환값으로 하고 jmp_buf에 저장된 시점으로 jump함.

이는 아래와 같이 setjmp()의 반환값으로 조건을 나눠 처리할 수 있음. 이때 setjmp()를 호출한 함수가 종료되서 stack frame에서 없어진 경우에는 해당 지점으로 jump할 수 없다는 한계가 존재함.

```c
jmp_buf buf;
int error1 = 0;
int error2 = 1;
void foo(void), bar(void);

int main()
{
   switch(setjmp(buf)) {
   case 0:
      foo();
      break;
   case 1:
      printf("Detected an error1 condition in foo\n");
      break;
   case 2:
      printf("Detected an error2 condition in foo\n");
      break;
   default:
      printf("Unknown error condition in foo\n");
   }
   exit(0);
}
```

```c
/* Deeply nested function foo */
void foo(void)
{
   if (error1)
      longjmp(buf, 1);
   bar();
}

void bar(void)
{
   if (error2)
      longjmp(buf, 2);
}
```

error recovery 등에 사용될 수 있지만, 이는 함수 호출 과정을 무시하므로 쓰는 것이 권장되지는 않는다고 함.

9. Synchronization

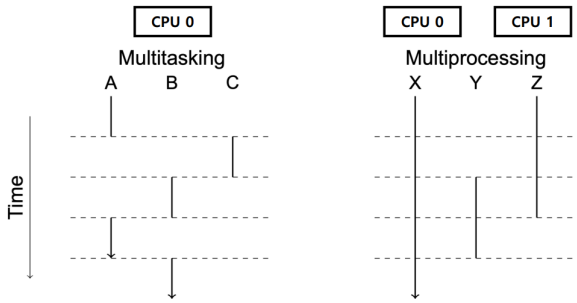
9.1. Concurrency

9.1.1. Concurrency

1. Concurrency

Concurrency는 하나 이상의 logical control flow들이 동시에 존재할 수 있도록 하는 것으로, 이때의 control flow를 Concurrent Flow라고 함. concurrency를 제공하는 system은 각 process가 dedicated computer에 의한 logical control flow를 가지고 동작하는 것처럼 처리함.

concurrency는 여러 개의 processor를 활용하는 Multiprocessing으로도 구현될 수 있지만, 시간을 여러 개의 time slot으로 나누고 context switching을 통해 하나(또는 다수)의 processor를 활용하는 Multi-tasking으로도 구현될 수 있음.



2. Concurrency and Separation

concurrency에 의한 dedicated computer model은 concurrent flow가 서로 unrelated이면 동작이 단순하지만, related인 경우 복잡해짐. 각 경우에 대해 아래와 같이 나누어 생각할 수 있음.

1) Independent, unrelated task : 각 concurrent flow가 독립적이고 관련이 없는 경우. 단순히 처리하면 됨.

2) Independent, related task : 각 concurrent flow가 함께 작업하도록 설계되진 않았지만 서로 유사한 작업을 수행하는 경우. synchronization이 필요할 수 있음.

3) Cooperating task : 각 concurrent flow가 함께 작업하도록 설계된 경우. synchronization이 필요할 수 있음.

9.2. Shared Memory

9.2.1. Shared Memory

1. Shared Memory

Shared Memroy는 여러 concurrent flow에 대한 공유 memory를 사용하고, 해당 공유 memory에 대해 읽고 씀으로써 통신하는 방식임. shared memory를 활용하는 방식(type)으로는 아래와 같이 3가지가 있음.

1) shared memory를 하나의 process안에서 여러 개의 thread가 서로 다른 시간 동안 활용하는 경우. 대체로 문제가 없음.

2) shared memory를 하나의 process안에서 여러 개의 thread가 같은 시간 동안 활용하는 경우. memory를 기본적으로 공유하므로 별도의 처리가 필요하지 않음.

3) shared memory를 서로 다른 process들이 같은 시간 동안 활용하는 경우. thread 간 shared memory와 달리, process 간의 shared memory를 위해서는 아래와 같은 방식을 활용해야 함.

3-1) fork 이전에 shared mapping을 생성함.
3-2) shm_open()으로 named mapping을 생성함.
3-3) memory-mapped file을 활용함.

2. Write Propagation Problem

shared memory에 따른 concurrency에서는 consistency가 고려되어야 하고, 이에 대한 synchronization을 수행함. 다만 이때는 timing 등의 temporal synchronization만이 아니라 caching 등의 spatial synchronization도 처리되어야 함.

cache는 계층 구조를 가짐. 또한 process들에 대해 cache는 공유되기도 하고, 서로 다른 core를 활용한다면 core에 대해 독립된 공간으로 존재하기도 함(local cache). 이에 따라 어떤 memory 부분에 write한 결과는 즉시 모든 cache에 반영되는 대신 점진적으로 propagate됨. 이때 local cache를 활용한다면 이는 다른 process에게 보이지 않는데, 이에 따라 적절한 순서로 instruction을 사용하여 shared memory를 활용하면서도 그 내용이 반영되지 않을 수 있음. 이를 Write Propagation Problem이라고 함.

3. Barrier

temoral synchronization은 register의 결과가 memory에 작성됨을 보증하고, 그 결과가 cache가 아닌 memory에 반영되었음을 보장하는 것이 Barrier임. barrier는 아래와 같은 작업 중 하나 이상을 수행함. barrier는 하드웨어에서 지원하는 function으로, 대부분의 processor는 이를 지원함.

1) write한 내용이 모든 core에게 visible할 때까지 현재 core를 block함.
2) write한 내용이 모든 core에게 visible할 때까지 모든 core를 block함.
3) write한 내용이 모든 core에게 visible할 때까지 특정 location에 대해 모든 core를 block함.
4) 해당 instruction에 대한 cpu instruction reordering을 막음.

즉, 아래와 같이 동작함.

Consider:

1. Core C0 executes a write for memory location m
2. The write is stored to C0's L1 cache
3. Core C1 issues a barrier for all writes to m
4. C0's L1 propagates m to C0's L2
5. C0's L2 propagates m to the shared L3
6. Core C1 blocks because C0 is writing m
7. Core C1 executes a read for memory location m
8. C1 reads m from shared L3

fork(), pthread_mutex_lock() 등 대부분의(POSIX의 모든) synchronization 함수들은 barrier를 포함함.

9.2.2. Explicitly Shared Memory

1. Explicitly Shared Memory

process들은 Implicitly Shared Memory로 shared library, executable image, kernel memory 등을 read-only 또는 숨김 상태로 가짐.

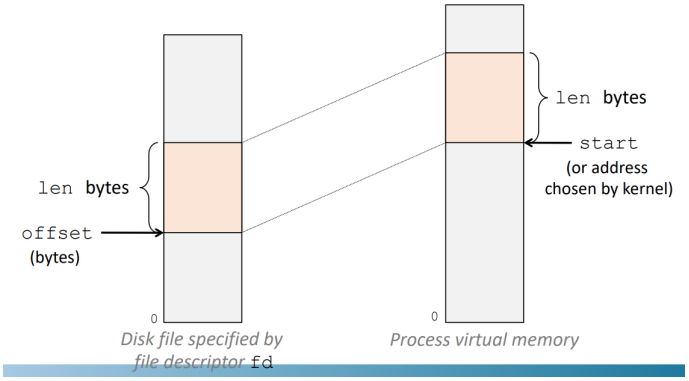
Explicitly Shared Memory를 활용하기 위해서는 kernel의 도움을 받아야 함. POSIX에서는 mmap(), shm_open() system call을 제공함. 이를 활용하여 shared memory를 explicit하게 생성하여 활용할 수 있음.

2. mmap()

mmap()는 각 process에서 memory map을 조정할 수 있도록 하는 system call임. 이는 기본적으로 disk에 존재하는 file을 memory에 mapping하여 활용할 수 있도록 함. mmap()으로 shared memory를 활용하는 방법에는 아래와 같이 2가지가 있음.

1) file에 대한 memory map을 생성하고, 각 process가 file을 읽고 씀.
2) memory map을 생성한 뒤 fork하여 parent와 child가 해당 공간에 읽고 씀.

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```




```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flags, int fd,
                                                    off_t offset);
```

mmap()은 위와 같은 원형을 가짐. 반환값은 생성된 memory 공간의 시작 주소이고, 각 parameter의 의미는 아래와 같음. 이때 mapping이 생성된 이후에는 file이 삭제되어도 memory 공간은 존재함.

1) addr : mapping을 생성할 memory 주소를 지정함. NULL로 지정하면 kernel이 알아서 결정함.
2) len : mapping 시에 생성할 memory 공간의 길이를 바이트 단위로 지정함.
3) prot
4) flag : mapping의 종류를 or 비트 연산자(|)로 지정함. MAP_PRIVATE 또는 MAP_SHARED는 중 하나는 반드시 지정해야 하고, MAP_READ, MAP_WRITE, MAP_ANONYMOUS(file을 map하지 않음.) 등을 지정할 수 있음.
5) fd : mapping을 생성할 file을 지정함. file을 map하지 않는 경우 -1을 지정함.
6) offset : file에서 mapping을 시작할 시작 위치를 바이트 단위로 지정함.

MAP_SHARED가 지정된 경우 fork 이후 child가 해당 memory를 shared memory로 활용할 수 있음. 즉, COW로 page가 복사되지 않음. 또한 MAP_ANONYMOUS을 지정하여 file 없이도 shared memory를 생성할 수 있음. 아래는 그 예시임.

```
int fd = open("somefile", O_RDWR);
void *mapping = mmap(NULL, 4096,
                     PROT_READ | PROT_WRITE,
                     MAP_SHARED, fd, 0);
close(fd);
```

3. shm_open()

shm_open()은 kernel memory buffer를 참조하는 file descriptor를 반환하는 system call로, file 없이 shared memory를 활용할 수 있도록 함.

file을 활용하여 shared memory를 생성하면 사용하는 process가 없거나 computer가 재부팅되어도 그 내용이 존재하고, 이를 쉽게 뜯어서 분석할 수 있음. 하지만 단순한 정보 교환에서는 file이 필요하지 않을 수 있음.

```c
#include <sys/mman.h>
#include <fcntl.h>

int shm_open(const char *name, int flags, int mode);
```

shm_open()은 위와 같은 원형을 가짐. name에는 해당 shared memory의 이름으로 문자열을 지정하고, 임의의 process가 해당 이름으로 shared memory에 접근할 수 있음. 이때 flag와 mode는 open()과 동일함. 이때 shm_open()은 buffer의 생성만 수행하므로, ftruncate()를 호출해 실제 memory 공간을 할당해야 함.

memory 공간 할당 이후 mmap()으로 관리할 수도 있음.

이렇게 할당된 memory는 computer가 재부팅되거나, shm_unlink()로 이를 제거하고 모든 process에서 mapping을 없애야 사라짐.

```c
int fd = shm_open("/shm_example", O_CREAT | O_RDWR, 0600);
ftruncate(fd, 4096);
void *mapping = mmap(NULL, 4096,
                     PROT_READ | PROT_WRITE,
                     MAP_SHARED, fd, 0);
```

executable은 disk에 저장되어 있다가, 실행할 때 memory로 load되는데, 이때 mmap()이 활용됨. mmap()으로 각 section의 내용을 추출할 수 있음.

9.3. Race

9.3.1. Synchronization

1. Race Condition
Race Condition 또는 Race는 하나 이상의 dependent한 event들이 존재하고, 제어되지 않은 순서에 따라 error state가 도출되는 상황임.

race condition을 잘 처리하지 못하면 program의 결과가 처리 타이밍에 따라 다르게 결정될 수 있고, 이는 데이터에 대한 consistency를 해칠 수 있음.

Data Race는 shared data(shared state)에 대해 하나 이상의 concurrent flow들이 read/write을 시도하는 race condition임. 이에 대한 synchronization은 주로 각 flow들에 대한 구체적인 순서를 지정하는 것은 아니고, 단순히 각 flow가 memory에 one-by-one으로 접근하도록 함.

2. Synchronization
Synchronization는 event의 발생 순서를 결정해 race condition을 해소하는 작업임. 예를 들어, 공유 데이터에 여러 process/thread가 접근할 때의 그 접근 순서를 결정함.

synchronization은 synchronization object를 critical section의 경계에 두는 것으로 구현됨.

9.3.2. Critical Section

1. Critical Section
Critical Section은 race condition이 발생할 수 있는 코드 영역으로, 한 번에 하나의 control flow만 접근해야 하는 부분임.

critical section은 shared data에 접근하는 코드를 포함하는 경우가 많음. shared data로의 write은 대부

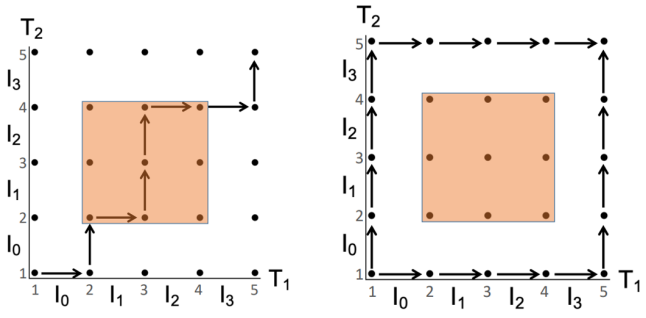
분의 경우 critical section이고, shared data로의 read는 critical section이 아닌 경우가 많음.

2. Progress Graph

Progress Graph는 concurrent flow와 critical section을 modeling하는 graph임. n개의 concurrent flow가 존재하는 경우 progress graph는 n차원이고, critical section 또한 n차원 영역으로 나타냄. 이때 점은 code line, 화살표는 실행임.

아래와 같이 critical section 영역을 나타냈을 때 안에 화살표가 있으면 error가 발생했을 수 있는 상황임. 이때 화살표가 지나가는 점만 영역 안에 있는 경우는 문제되지 않음. 즉, progress graph로 critical section에 한 번에 하나의 concurrent flow만 들어갔는지를 확인할 수 있음. 해당 영역을 피해가도록 synchronization을 적용해야 함.

```
1 void setstring(char *str) {
2   int index = nstrings;
3   strings[index] = str;
4   nstrings++;
5 }
```



9.4. Synchronization의 구현

9.4.1. Atomic Operation

1. Atomic Operation

Atomic Operation은 아래와 같은 특징을 가지는 operation으로, 하드웨어적인 지원을 통해 구현됨. 이는 가장 단순한 형태의 synchronization임.

1) interrupt되지 않음.
2) 실행 중에 다른 operation이 동시에 실행되지 않는 것처럼 실행됨.
3) 실행이 fully success하거나, 별다른 영향 없이 fail함.

synchronization은 synchronization object를 critical section의 경계에 두는 것으로 구현되는데, synchronization object 자체에 대한 synchronization 또한 처리되어야 함. 즉, synchronization object는 atomic operation이어야 함.

lock을 만들기 위해서는 atomic operation이 지원되어야 한다. 하드웨어적인 지원을 받아야 한다. 동기화 object 내부에 critical section이 존재하면.. 의미가 없으므로..

2. Atomic Operation 사용

atomic operation은 c에서 표준으로 지원하지 않음. atomic operation은 assembly를 사용하거나, library 함수를 사용하거나, kernel의 지원을 받아야 함.

XCHG와 CMPXCHG는 x86 아키텍처에서 제공하는 assembly instruction임. XCHG는 두 operand를 교환하는 연산이고, CMPXCHG는 Compare and Swap(CAS) 연산으로 dst와 target이 같으면 dst와 src를 교환하는 연산임. 이를 c언어로 나타내면 아래와 같음.

```
void XCHG(int *dst, int *src)
{
   int temp = *dst;
   *dst = *src;
   * src = temp;
}
```

```
void CMPXCHG(int *dst, int *src, int target)
{
   int temp = *dst;

   if(*dst == target)
   {
      return XCHG(dst, src);
   }
}
```

이를 활용해 lock만을 단순 구현하는 예시를 c언어로 나타내면 아래와 같음. g_lock은 현재 lock이 걸려 있는지를 저장하는 변수임(1이면 걸려있는 것, 0이면 걸려있지 않은 것.).

```
lock = 1;
XCHG(&g_lock, &lock);
if(lock == 0)
{
 ... // critical section
}
```

```
lock = 1;
MPXCHG(&g_lock, &lock, 0)
if(lock == 0)
{
 ... // critical section
}
```

LOCK은 atomic 연산을 수행하도록 지정하는 prefix로, LOCK XCHG나 LOCK CMPXCHG와 같이 작성할 수 있음. 근데 이를 지정하지 않아도 atomic하게 수행된다고 함.

모든 machine instruction이 atomic한 것은 아님.

9.4.2. Mutex

1. Mutex

Mutex는 mutual exclusion을 구현하는 소프트웨어 도구로, lock과 unlock을 제공함. lock과 unlock은 mutex state에 따라 다르게 동작함.

Mutex State	Operation	Action
Unlocked	Lock	Lock mutex immediately
Locked	Lock	Block until unlocked, then lock
Locked	Unlock	Unlock mutex immediately
Unlocked	Unlock	Implementation dependent

프로그래머의 실수로 unlocked를 unlock을 하는 상황이 발생할 수도 있는데, 구현에 따라 다르지만 그냥 무시하고 넘어가거나, 예외 또는 오류를 발생시키도록 함.

mutex에서는 lock과 unlock을 critical section 앞뒤로 걸어서 한 번의 하나의 flow만 접근하도록 함.

2. Condition Variable

Condition Variable은 mutex와 함께 사용하는 variable로, block 적용에 대한 condition 값을 저장함.

아래의 코드와 같이 사용될 수 있음. 해당 값에 따라 block하고, 이후 이 값을 수정한 뒤 조건을 다시 검사해 block이 풀림. 이때 condition variable에 대한 수정도 lock/unlock 사이에 있어야 함. 또한 signal을 보내 wait 중인 flow를 깨운 뒤 다시 조건을 검사하는 것을 Wake and Check라고 하는데, 이는 spurious wakeup(명시적인 signal 등 없이 깨어나는 동작) 등의 상황을 방지함.

```
Mutex m
ConditionVariable cv
Data d

waiter() {
    lock m
    while condition on d {
        wait on cv
    }
    take action
    unlock m
}
```

```
signaler() {
    lock m
    modify d
    signal cv
    unlock m
}
```

mutex는 초기화 값이 1인 semaphore와 동일함.

9.4.3. Semaphore

Semaphore는 두 개의 atomic operation P와 V를 가지는 integer 변수임. semaphore는 nonnegative integer로 초기화되고, 초기화 값은 critical section에 현재 접근 가능한 process의 개수를 의미함.

semaphore(s)에 대한 P와 V의 동작은 아래와 같음. P와 V는 critical section 앞뒤로 사용함.

P() : s가 0보다 크면 s 값을 1 줄이고, s가 0이면 0보다 커질 때까지 block함.
V() : block된 flow가 있다면 하나를 release하고, 없다면 semaphore 값을 1 늘림.

```
P();
... // critical section
V();
```

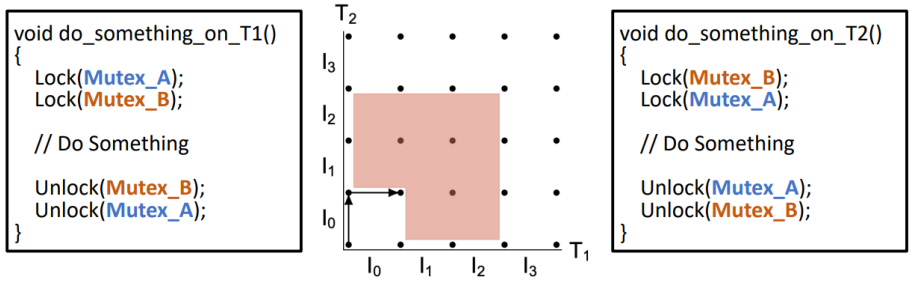
9.4.4. Deadlock

Deadlock은 두 process가 서로의 작업이 완료되기를 기다리고 있으면 어느 한 쪽도 작업이 완료되지 못하는 상황임. 예를 들어, 아래와 같은 상황이 존재할 수 있음.

deadlock이 발생하려면 아래와 같은 조건들이 만족해야 함.

1) 적어도 하나의 resource는 mutually exclusive해야 함. 2) flow가 lock이 풀리길 기다리면서 lock을 들고 있어야 함. 3) lock은 해당 flow가 놓기 전까지는 빼앗을 수 없음. 4) flow들이 circular하게 lock을 들고 있어야 함.

deadlock은 실제로는 timer 등을 사용해서 일부 해결할 수도 있지만, 여기의 구현에서는 해결이 불가능함. 또한 deadlock은 푸는 게 아니라 피해야 함. lock에 순서를 부여해 deadlock을 피할 수 있음. lock은 in-order로 걸고, unlock은 reverse order로 걸면 되는데, 모든 flow에서 이 순서가 통일되어 있어야 함. 아래와 같이 순서가 다르면 deadlock이 발생함.



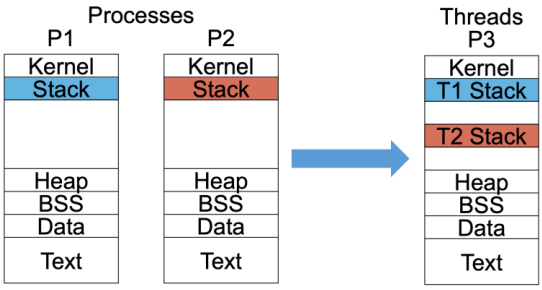
9.5. Thread

9.5.1. Thread

1. Thread
Thread는 process 내의 logical control flow로, process보다 작은 단위의 abstraction임.

병렬화를 위해서는 매번 새로운 process를 생성(cooperative process)해야 함. 이 경우 각 process끼리의 IPC(system call. 비용이 큼.)가 필요하고, 매번 process 간의 context switiching이 수행되므로 오버헤드가 크기 때문에 thread라는 더 작은 단위로 병렬화를 구현함. thread는 특정 process 내부 작업의 병렬화에 기여함.

하나의 process에 속하는 thread들끼리는 memeory map을 공유함. context에는 stack이 저장되는데, 여러 개의 stack을 저장하면 여러 개의 control flow를 구현할 수 있음.



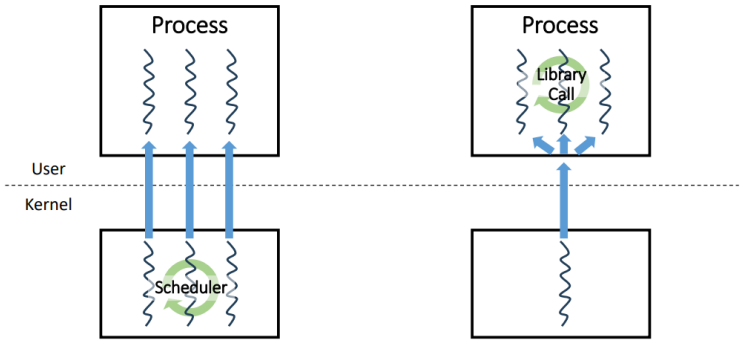
process 내부의 thread들은 memory map 등 resource를 공유하므로 더 낮은 비용으로 작업을 수행할 수 있음. 하지만 shared resource에 대한 처리가 까다로울 수 있고, 많은 API가 lock을 활용하여 구현되지 않았으므로 thread-unsafe함.

thread는 여러 task 사이의 control 전환이 빈번한 경우, shared data에 많이 접근해야 하는 경우, 하나의 cpu에서 빠르게 수행될 수 있는 연산을 주로 활용하는 경우 등에 사용됨.

2. Kernel-level Thread vs. User-level Thread
thread model에는 아래와 같이 두 가지가 있음.

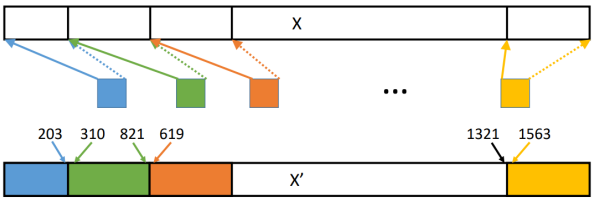
1) Kernel-level Thread : kernel 내부에 존재하며 kernel에 의해 scheduling되는 thread. kernel은 나름의 context block을 가지고 있고, 이에 대한 context switching 비용은 user-level thread보다 높음. kerenl은 core가 여러 개인 경우 병렬 실행을 수행함.

2) User-level Thread : library 코드에 의해 구현 및 scheduling되는 thread. kernel은 해당 thread를 인지하지 못하므로, library에서 자체적으로 thread를 처리함. 이에 따라 context switching이 더 가볍고, 사용자가 flexible하게 scheduling을 관리할 수 있음. 이는 실제로 하드웨어적인 병렬성은 이용하지 못하므로 하나의 thread가 block되면 전체가 block될 수 있음.



3. Inter-thread Communication
단순히 shared memory를 활용하는 것 외에도, pipe, message queue, signal 등의 Inter-thread Communication이 존재함. 이는 IPC와 동일한 방식이지만 별도의 API를 활용해야 함.

아래와 같이 정렬에 thread를 활용할 수도 있음.



9.5.2. pthread

pthread는 POSIX에서 제공하는 threading API로, thread에 대한 생성 및 종료와 thread synchronization을 지원함. 여기에서는 그 사용법을 알아보자.

시스템에 따라 다르지만, pthread를 compile/linking할 때는 -pthread를 옵션으로 지정해 줘야 함.

1. Thread Creation
pthread_create()으로 thread를 생성할 수 있음.

```
#include <pthread.h>
int pthread_create( pthread_t *thread, const pthread_attr_t *attr,
                    void *(*start_function)(void *), void *arg )
```

pthread_create()의 인자는 아래와 같음. 이를 호출하면 stack을 포함한 execution context와 concurrent flow가 생성됨. 이는 user space에 생성되거나, (linux의 경우) kernel에게 생성을 요청함. 이후 start_function에 argument를 전달하며 호출함.

1) thread : 생성된 thread를 저장할 phtread_t 포인터를 지정함.

2) attr : scheduler, detach size 등을 포함하는 attribute를 지정함. NULL을 지정하면 default attribute가 지정됨.

3) start_function : 생성한 thread의 flow가 호출할 함수를 지정함. 예를 들어, 아래와 같이 정의한 함수 이름을 지정할 수 있음.

```
void *thread_main(void *arg) { return NULL; }
```

4) arg : start_function에 전달할 argument를 지정함.

예를 들어, 아래와 같이 사용됨.

```
pthread_t thread;
pthread_create(&thread, NULL, thread_function, NULL);
```

2. Thread Termination
thread는 아래와 같은 방법으로 종료될 수 있음.

1) process가 종료됨.
2) pthread_exit()이 호출됨.
3) start_function이 종료됨.
4) 다른 thread가 pthread_cancel()을 호출함.

각 방법 모두에서 thread가 detach되지 않았다면, 종료 시에 join되기 전까지 zombie process와 같은 상태가 됨.

3. Thread Joining and Detaching
thread에 대한 joining과 detaching에 대해 알아보자.

1) Joining
Joining은 특정 thread가 종료될 때까지 기다리도록 하는 것으로, process의 wait()과 유사함. 아래와 같이 pthread_join()에 인자로 지정한 thread가 종료될 때까지 caller를 block한 뒤, 해당 thread가 종료되면 exit status를 retrieve함.

즉, thread가 synchronous하게 처리됨.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

2) Detaching
Detaching은 thread가 종료되었을 때 자원을 자동으로 해제하도록 설정하는 것임. 이에 따라 종료 시에 zombie 상태가 되지 않음. thread 생성 시에 attribute로 지정하거나, pthread_detach()를 호출하여 detach하도록 지정함.

즉, thread가 asynchronous하게 처리됨.

9.5.3. pthread Synchronization

pthread에서 제공하는 thread synchronization을 알아보자.

1. Mutex

mutex는 pthread_mutex_t 형으로 처리함. pthread에서 다른 thread에서 lock한 mutex를 unlock하는 것은 error임.

1) mutex initialization

mutex는 아래와 같이 PTHREAD_MUTEX_INITIALIZER를 사용하는 static initializer와, pthread_mutex_init()을 사용하여 mutex에 대한 attribute를 지정할 수 있는 dynamic initializer가 존재함.

```
#include <pthread.h>
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

2) mutex operation

아래와 같은 함수들로 lock/unlock을 할 수 있음. 이때 trylock 연산은 항상 즉시 반환되는데, 이미 lock이 걸려 있으면 EBUSY를 반환하고, lock이 걸려 있지 않으면 lock하고 0을 반환함.

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

3) destroying mutex

아래와 같이 mutex를 제거할 수 있음. 이때 lock이 걸려 있는 mutex를 제거하는 것은 error임.

linux에서는 mutex에 대한 명시적 삭제를 수행하지 않지만(아무 일도 일어나지 않음.), 다른 시스템들에서는 mutex에 대한 명시적 삭제를 통해 관련 resource를 반환함.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

4) Recursive Mutex

pthread mutex에서는 Recursive Mutex를 지원함. 기본적으로 하나의 mutex에 대해 단순히 lock만 두 번 거는 경우 unlock이 없으므로 deadlock이 걸리지만, pthread에서는 lock이 두 번 걸린 것으로 처리되고 이후 unlock을 두 번 하도록 함.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&mutex);
pthread_mutex_lock(&mutex); ← The same lock!
```

2. Condition Variable

pthread에서 condition variable은 mutex와 함께 사용함. condition variable은 pthread_cond_t 형으로 처리함.

1) condition variable initialization

condition variable도 mutex처럼 두 가지 방법으로 initialization할 수 있음.

```
#include <pthread.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
```

2) waiting

아래와 같이 condition variable과 mutex를 지정해 wait하도록 할 수 있음.

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

3) signaling
아래와 같이 wait 중인 한 thread 또는 wait 중인 전체 thread에 대해 signal을 전송할 수 있음. 이때 지정하는 wait 중인 thread가 없으면 아무 일도 일어나지 않음. mutex는 shared state에 대해 적용되는 것을 지정해야 함.

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

아래는 waiting과 signaling을 포함한 예시임.

Signaling Example

```
extern pthread_mutex_t lock;
extern pthread_cond_t cond;
extern bool done;

void signal_done() {
     pthread_mutex_lock(&lock);
     done = true;
     pthread_mutex_unlock(&lock);
     pthread_cond_signal(&cond);
}
```

```
void *block_until_done(void *ignored) {
      pthread_mutex_lock(&lock);
      while (!done) {
            pthread_cond_wait(&cond, &lock);
      }
      pthread_mutex_unlock(&lock);
}
```

아래는 pthread 관련 함수들을 모두 활용한 예시임.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
bool done;

int main(int argc, char *argv[]) {
      pthread_t t;
      pthread_create(&t, NULL, block_until_done, NULL);
      usleep(100000);
      signal_done();
      pthread_join(t, NULL);
      return 0;
}
```

4) destroying
아래와 같이 condition variable을 삭제할 수 있음. waiting thread에서 condition variable을 삭제하는 것은 error임.

mutex에서와 같이, linux에서는 condition variable에 대한 명시적 삭제를 수행하지 않지만(아무 일도 일어나지 않음.), 다른 시스템들에서는 condition variable에 대한 명시적 삭제를 통해 관련 resource를 반환함.

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

3. Semaphore
posix semaphore는 thread와 preocess 모두에 대해 적용이 가능함.

1) semaphore initialization
아래와 같이 semaphore를 생성할 수 있음. semaphore에 대해서는 static initalizer가 없음. pshared가 true로 지정되면 해당 sempahore는 process 간에도 사용이 가능함. value로는 semaphore의 초깃값을 지정함.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

2) semaphore manipulation

wait()은 P()에, post()는 V()에 해당됨. trywait()은 mutex의 trylock과 유사하게, 해당 연산은 항상 즉시 반환되는데, 접근에 실패하면 EAGAIN을 반환하고, 접근에 성공하면 0을 반환함.

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
```