CSAPP

EUnS

January 4, 2020

Contents

Contents **1**

1 A Tour of Computer Systems **3**

I Program Structure and Execution **11**

2 Representing and Manipulating Information **13**

3 Machine-Level Representation of Programs **15**
3.1 x86 assembly 15

4 Processor Architecture **17**

5 Optimizing Program Performance **19**
5.1 Capabilities and Limitations of Optimizing Compilers 19
5.2 Eliminating Loop Inefficiencies 20
5.3 Reducing Procedure Calls 22
5.4 Eliminating Unneeded Memory References 22
5.5 Understanding Modern Processors 23
5.6 Loop Unrolling 23
5.7 Enhancing Parallelism 24
5.8 Summary of Results for Optimizing Combining Code 27
5.9 Some Limiting Factors 27
5.10 Understanding Memory Performance 27
5.11 Life in the Real World: Performance Improvement Techniques . 27

6 **29**

II Running Programs on a System **31**

7 Linker **33**
7.1 Compiler Drivers 33
7.2 Static Linking 33
7.3 Object Files 34
7.4 Relocatable Object Files 34
7.5 Symbols and Symbol Tables 36
7.6 Symbol Resolution 36
7.7 Relocation 37
7.8 Executable Object Files 37
7.9 Loading Executable Object Files 37
7.10 Dynamic Linking with Shared Libraries 37
7.11 Loading and Linking Shared Libraries from Applications 38
7.12 Position-Independent Code (PIC) 38
7.13 Library Interpositioning 38

8 Exceptional Control Flow **41**
8.1 Exceptions 41
8.2 Processes 42
8.3 Signals 43
8.4 Nonlocal Jumps 43
8.5 Tools for Manipulating Processes 43

III Interaction and Communication between Programs **45**

Chapter 1

A Tour of Computer Systems

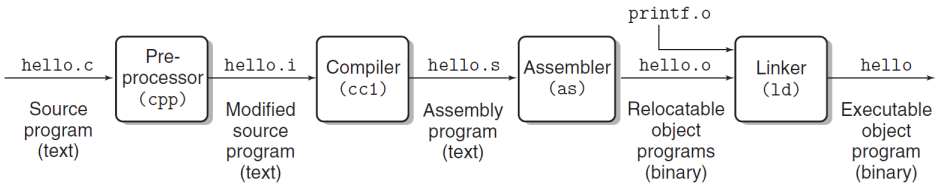


Figure 1.1: The compilation system.

linux¿ gcc -o hello hello.c
컴파일 시스템을 이해해야하는 이유

1. 성능 최적화
2. 링커 에러 이해하기
3. 보안 약점

- Buses

  하드웨어에 돌아다니는 데이터 통로(a collection of electrical conduits) word라고 하는 고정크기의 바이트 단위로 데이터가 전송된다. 최근에는 4byte 또는 8byte 크기를 가진다.

- I/O Devices 시스템과 외부로부터의 연결을 가능케하는 장치.

- Main Memory

  프로세서가 프로그램을 실행하는 동안 데이터와 프로그램을 저장하는 임시 저장장치이다. 물리적으로 DRAM(Dynamic Random Access Memorry)로 구성되어 있다.

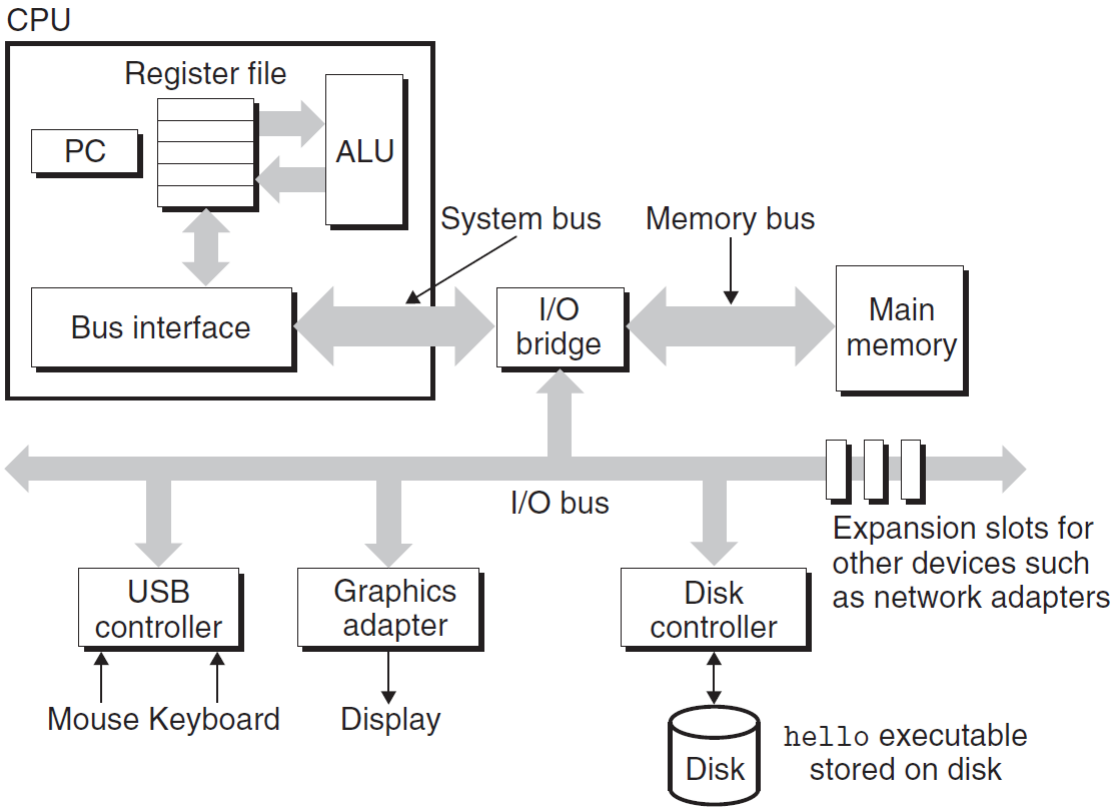


Figure 1.2: Hardware organization of a typical system. CPU: central processing unit, ALU: arithmetic/logic unit, PC: program counter, USB: Universal Serial Bus.

- Processor(CPU)

  프로그램 카운터(PC)는 메인메모리의 기계어 인스트럭션을 가리키는데 이 인스트럭션 값을 읽어서 특정한 동작을 수행하고 PC값을 갱신한다. 이 동작은 메인 메모리, 레지스터 파일, ALU주위를 돈다. 레지스터 파일은 고유의 이름을 갖는 워드 크기의 레지스터 집합이다.

추가적인것 .
캐시
메모리 계층
쓰레드 동시성
프로세스 contex switching
가상메모리
파일

네트워크

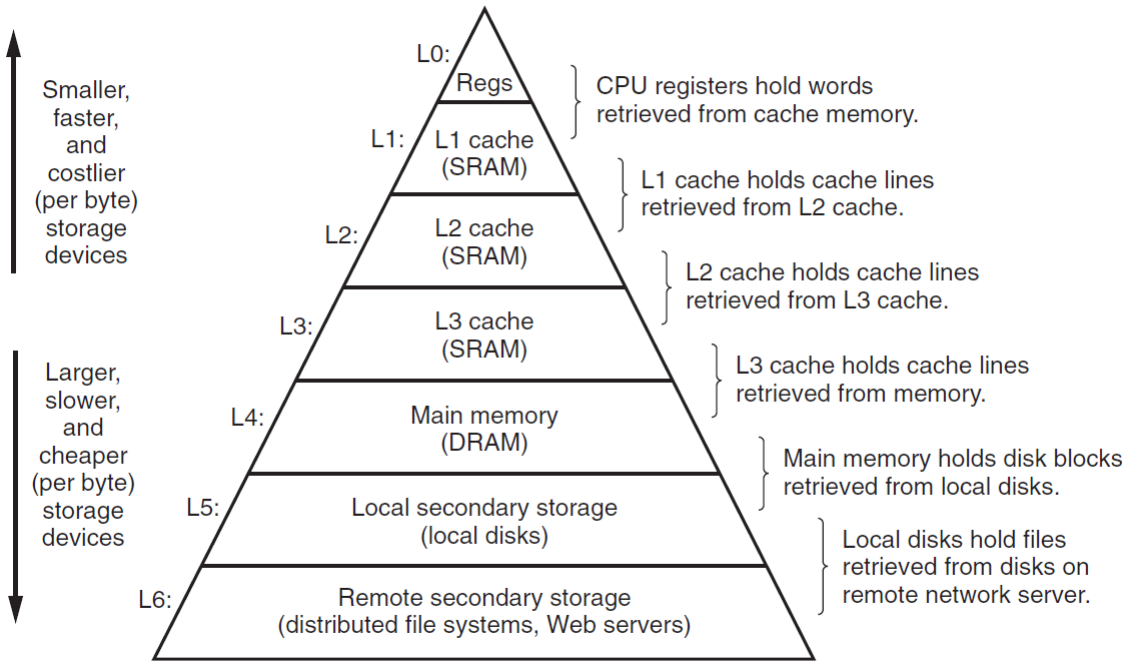


Figure 1.3: 메모리 계층

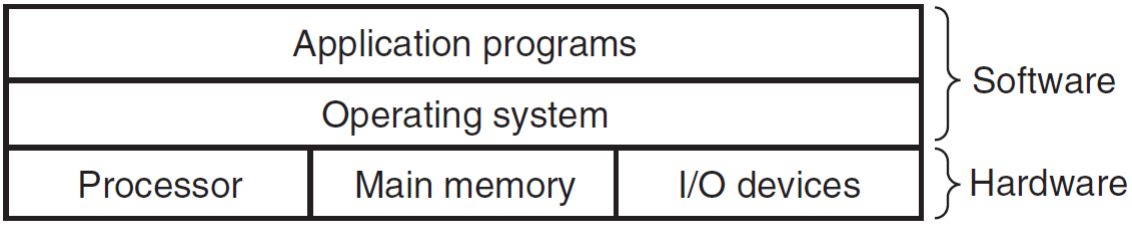


Figure 1.4: Layered view of a computer system.

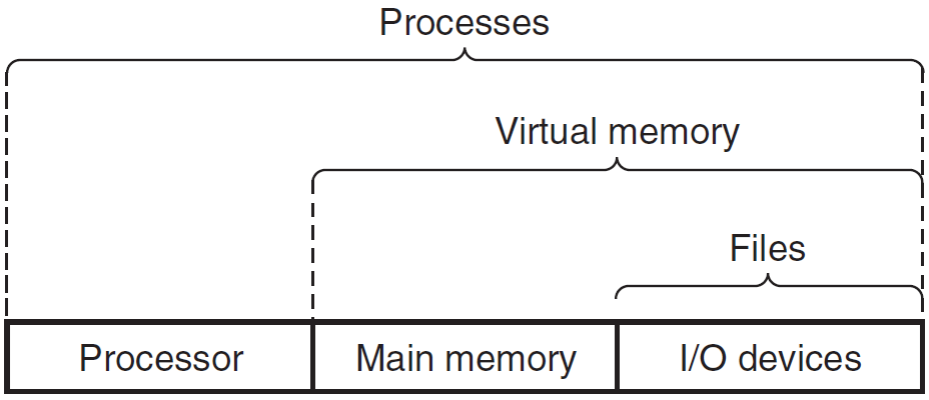


Figure 1.5: Abstractions provided by an operating system.

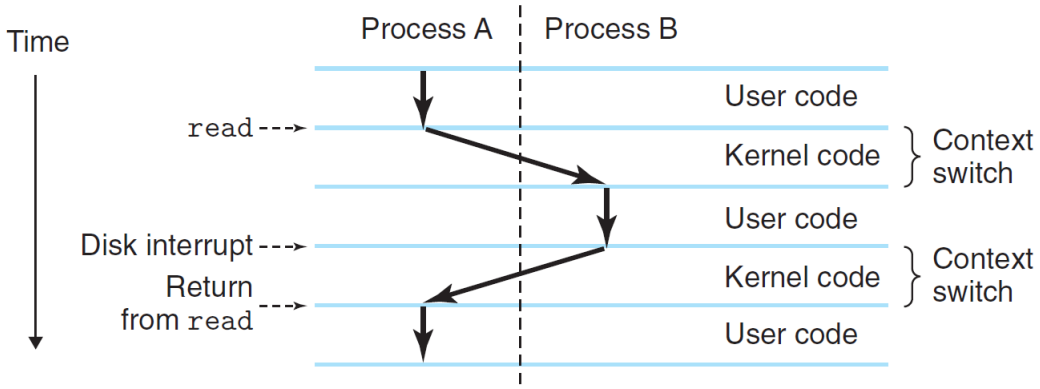


Figure 1.6: context switching

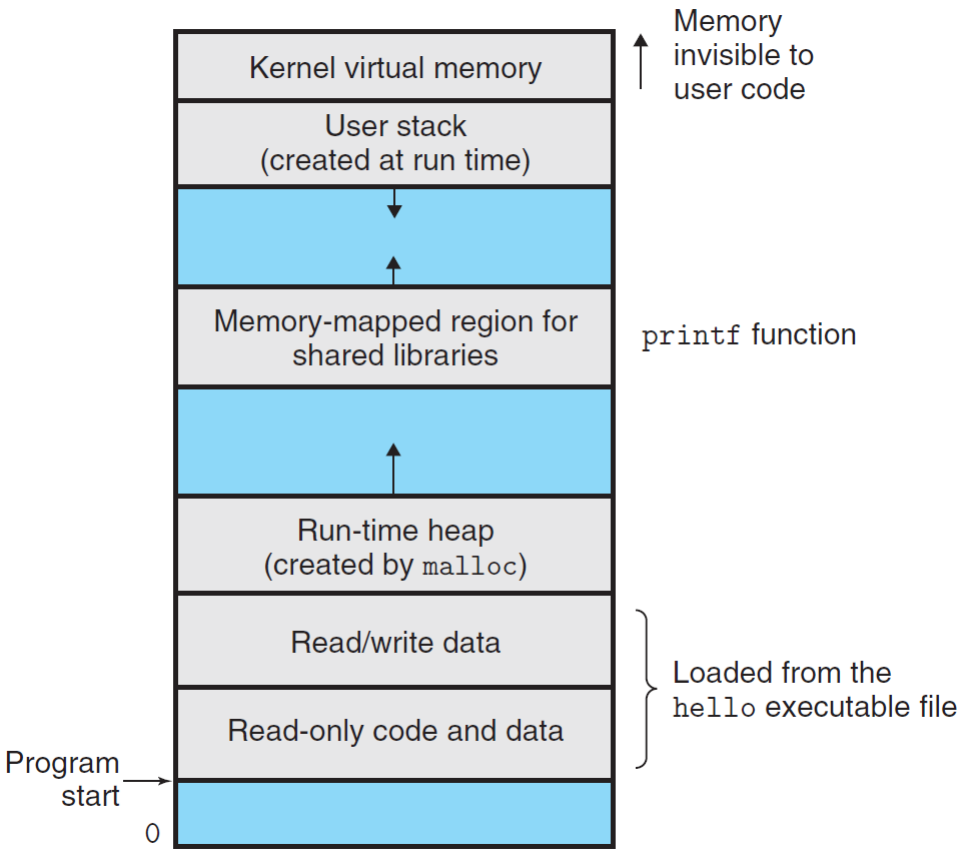


Figure 1.7: Process virtual address space.

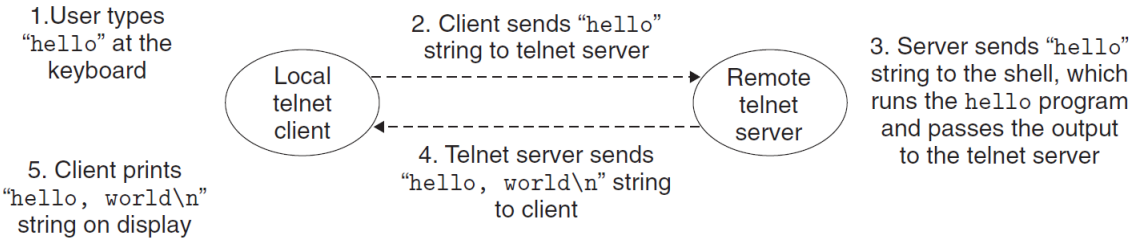


Figure 1.8: Using telnet to run hello remotely over a network.

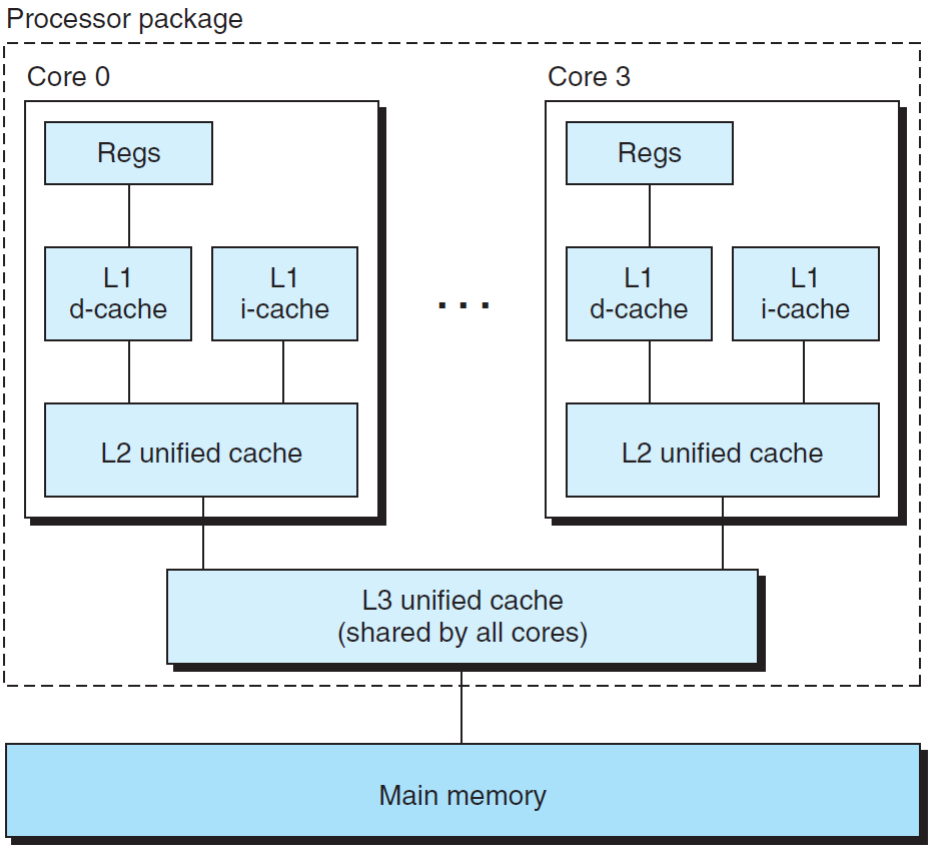


Figure 1.9: Multi-core processor organization. Four processor cores are integrated onto a single chip.

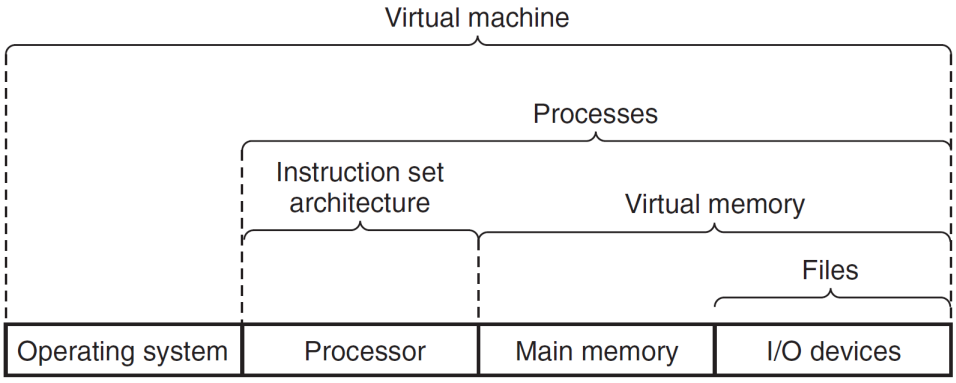


Figure 1.10: Some abstractions provided by a computer system

Part I

Program Structure and Execution

Chapter 2

Representing and Manipulating Information

메모리에 객체가 저장되는 방식
객체의 수조는 사용된 바이트의 최소 주소로 정함. 4byte크기인 int type 변수 x가 주소 0x100로 설정된다면 $int\&x$의 값은 0x100이고 주소 0x100,1,2,0x103에 x가 저장된다.

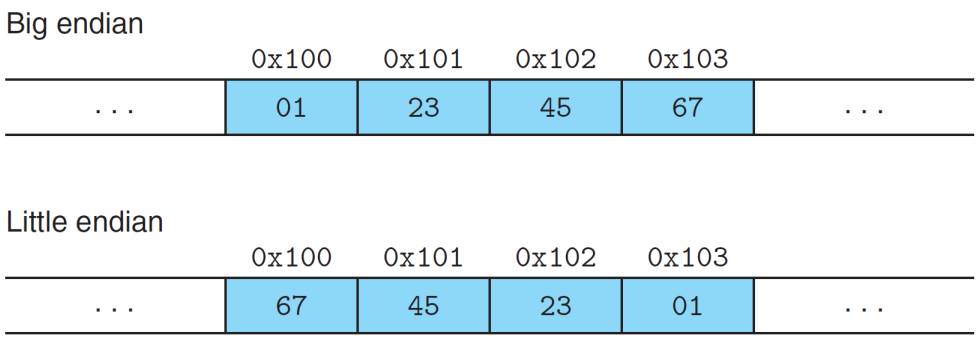


Figure 2.1: ex endian

- little endian : 최하위 바이트를 시작주소에 차례로 저장하는 방식 (intel)
- big endian : 최상위 바이트를 시작주소에 차례로 저장하는 방식 (IBM Oracle)

```
1 #include <stdio.h>
2 typedef unsigned char *byte_pointer;
3
4 void show_bytes(byte_pointer start, size_t len) {
5 int i;
```

```
6  for (i = 0; i < len; i++)
7  printf(" %.2x", start[i]);
8  printf("\n");
9  }
10
11 void show_int(int x) {
12 show_bytes((byte_pointer) &x, sizeof(int));
13 }
14
15 void show_float(float x) {
16 show_bytes((byte_pointer) &x, sizeof(float));
17 }
18
19 void show_pointer(void *x) {
20 show_bytes((byte_pointer) &x, sizeof(void *));
21 }
22
23
24 void test_show_bytes(int val) {
25     int ival = val;
26     float fval = (float) ival;
27     int *pval = &ival;
28     show_int(ival);
29     show_float(fval);
30     show_pointer(pval);
31     }
```

서로 다른 컴퓨터 타입은 서로 다르고, 호환성이 없는 인스트럭션과 인코딩을 사용한다. 다른 운영체제를 실행하는 동일한 프로세서들도 각자의 코딩 관습에 차이가 잇으며, 따라서 이들은 바이너리 호환성을 갖지 못한다.

```
1     void inplace_swap(int *x, int *y) {
2      *y = *x ^ *y; /* Step 1 */
3      *x = *x ^ *y; /* Step 2 */
4      *y = *x ^ *y; /* Step 3 */
5      }
6
```

Shift Operations in C

- 논리(logical) 우측 쉬프트 : 좌측 끝을 k개의 0으로 채운다.
- 산출(arithmetic) 우측 쉬프트 : 좌측 끝을 k개의 1로 채운다.

Operation	Value 1	Value 2
Argument x	[01100011]	[10010101]
x << 4	[0011*0000*]	[0101*0000*]
x >> 4 (logical)	[*0000*0110]	[*0000*1001]
x >> 4 (arithmetic)	[*0000*0110]	[*1111*1001]

Chapter 3

Machine-Level Representation of Programs

linux¿ gcc -Og -o p p1.c p2.c
-0g 옵션 : 최적화 x
ISA (instruction set architectune)
linux¿ gcc -Og -S mstore.c
-S : 어셈블리 코드만 만든다 .s
linux¿ gcc -Og -c mstore.c
바이너리 형식 목적파일 .o 생성
linux¿ objdump -d mstore.o
disassembly

3.1 x86 assembly

3.1 A Historical Perspective 202 3.2 Program Encodings 205 3.3 Data Formats 213 3.4 Accessing Information 215 3.5 Arithmetic and Logical Operations 227 3.6 Control 236 3.7 Procedures 274 3.8 Array Allocation and Access 291 3.9 Heterogeneous Data Structures 301 3.10 Combining Control and Data in Machine-Level Programs 312 3.11 Floating-Point Code 329 3.12 Summary 345

Chapter 4

Processor Architecture

387 4.1 The Y86-64 Instruction Set Architecture 391 4.2 Logic Design and the Hardware Control Language HCL 408 4.3 Sequential Y86-64 Implementations 420 4.4 General Principles of Pipelining 448 4.5 Pipelined Y86-64 Implementations 457 4.6 Summary 506 4.6.1 Y86-64 Simulators 508 Bibliographic Notes 509 Homework Problems 509 Solutions to Practice Problems 516

Chapter 5

Optimizing Program Performance

1. select an appropriate set of algorithms and data structures
2. write source code that the compiler can effectively optimize to turn into efficient executable code
3. divide a task into portions that can be computed in parallel, on some combination of multiple cores and multiple processors.

명심할것. 두번째를 이해하기위해 컴파일러의 능력과 한계를 알아야한다. 최적화를 위하면서도 코드 가독성은 유지해야한다.

5.1 Capabilities and Limitations of Optimizing Compilers

컴파일러 최적화 명령어는 -0g -1g -2g ... 이 있다.
다음 두개의 코드가 있다고 생각해보자

```
1  void twiddle1(long *xp, long *yp)
2  {
3  *xp += *yp;
4  *xp += *yp;
5  }
6
7  void twiddle2(long *xp, long *yp)
8  {
9  *xp += 2* *yp;
10 }
```

twiddle1은 twiddle2로 최적화 될수있는가? 답은 아니 다. xp와 yp가 동일하다고 생각해보자. 그러면 명확할 것이다.

```
1 long f();
2 long func1() {
3 return f() + f() + f() + f();
```

```
}

long func2() {
return 4*f();
}
```

func1 이 func2로 최적화 될 수 있다고 생각한다. 하지만 f에서 전역변수를 건든다고 생각해보자 4번 바뀔게 한번만 바뀌는 일이 될것이다.

컴파일러는 위험요소가 있을경우 최적화를 하지않는다.

5.2 Eliminating Loop Inefficiencies

```
void combine1(vec_ptr v, data_t *dest)
 {
 long i;

 *dest = IDENT;
 for (i = 0; i < vec_length(v); i++) {
 data_t val;
 get_vec_element(v, i, &val);
 *dest = *dest OP val;
 }
 }
```

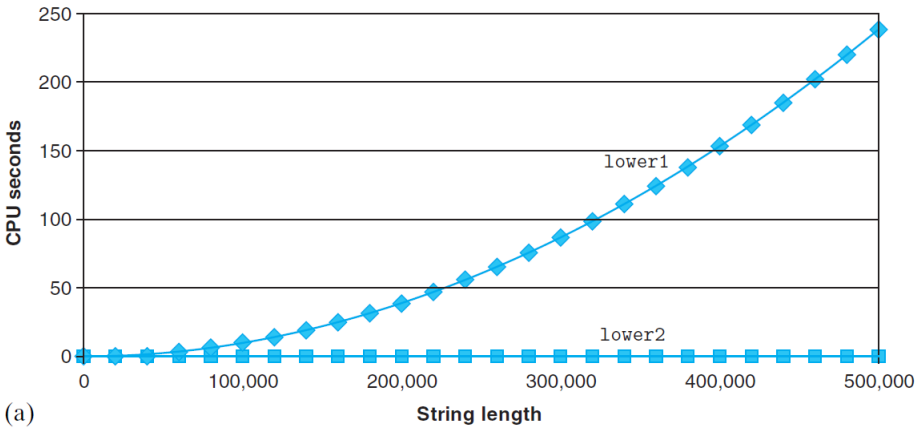
이게 어떻게 성능 개선이되는지 보자.

```
void combine2(vec_ptr v, data_t *dest)
 {
 long i;
 long length = vec_length(v);

 *dest = IDENT;
 for (i = 0; i < length; i++) {
 data_t val;
 get_vec_element(v, i, &val);
 *dest = *dest OP val;
 }
 }
```

			Integer		Floating point	
Function	Page	Method	+	*	+	*
combine1	543	Abstract -O1	10.12	10.12	10.17	11.14
combine2	545	Move vec_length	7.02	9.03	9.02	11.03

```
/* Convert string to lowercase: slow */
void lower1(char *s)
{
    long i;

    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
```

```
}

/* Convert string to lowercase: faster */
void lower2(char *s)
{
    long i;
    long len = strlen(s);

    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

/* Sample implementation of library function strlen */
/* Compute length of string */
size_t strlen(const char *s)
{
    long length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```



(a)

	String length						
Function	16,384	32,768	65,536	131,072	262,144	524,288	1,048,576
`lower1`	0.26	1.03	4.10	16.41	65.62	262.48	1,049.89
`lower2`	0.0000	0.0001	0.0001	0.0003	0.0005	0.0010	0.0020

(b)

Figure 5.1: lower 성능비교

시간복잡도 계산으로도 충분히 알 수 있다. 문자열 길이가 변하는게 아니라면

strlen을 반복문안에 넣는 짓은 하지말자.

5.3 Reducing Procedure Calls

```
void combine3(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = IDENT;
    for (i = 0; i < length; i++) {
    *dest = *dest OP data[i];
    }
}
```

대충 함수안에서 계속 함수를 쳐부르는 짓은 오버헤드와 불리는 함수안에서 처리하는 불필요한 작업으로 느려진다는 뜻. 근데 뒤에 더다룬다함

			Integer		Floating point	
Function	Page	Method	+	*	+	*
combine2	545	Move vec_length	7.02	9.03	9.02	11.03
combine3	549	Direct data access	7.17	9.02	9.02	11.03

5.4 Eliminating Unneeded Memory References

combine3에서 내부 루프는 포인터 dest가 메모리 참조를 계속하는 식이다. 다음 방식이 조금더 효율적이다.

```
void combine4(vec_ptr v, data_t *dest)
 {
 long i;
 long length = vec_length(v);
 data_t *data = get_vec_start(v);
 data_t acc = IDENT;

for (i = 0; i < length; i++) {
 acc = acc OP data[i];
 }
 *dest = acc;
 }
```

			Integer		Floating point	
Function	Page	Method	+	*	+	*
combine3	549	Direct data access	7.17	9.02	9.02	11.03
combine4	551	Accumulate in temporary	1.27	3.01	3.01	5.01

5.5 Understanding Modern Processors

아몰랑

5.6 Loop Unrolling

while의 작동방식을 어셈블리로 한번보면 if와 goto를 합쳐놓은 방식이다. if는 단일 연산에비해느림 따라서 if검사를 적게 하게하면(=반복되는 횟수를 줄이면) 성능개선이 이루어진다.

```
1  /* 2 x 1 loop unrolling */
2  void combine5(vec_ptr v, data_t *dest)
3      {
4      long i;
5      long length = vec_length(v);
6      long limit = length-1;
7      data_t *data = get_vec_start(v);
8      data_t acc = IDENT;
9
10     /* Combine 2 elements at a time */
11     for (i = 0; i < limit; i+=2) {
12     acc = (acc OP data[i]) OP data[i+1];
13     }
14
15     /* Finish any remaining elements */
16     for (; i < length; i++) {
17     acc = acc OP data[i];
18     }
19     *dest = acc;
20 }
```

			Integer		Floating point	
Function	Page	Method	+	*	+	*
combine4	551	No unrolling	1.27	3.01	3.01	5.01
combine5	568	2 × 1 unrolling	1.01	3.01	3.01	5.01
		3 × 1 unrolling	1.01	3.01	3.01	5.01
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

이 생각을 할수있다. 루프풀기를 최대로하면 제일 좋은게아닌가?
여러단점이있다.
http://z3moon.com/e/loop_unrolling
https://en.wikipedia.org/wiki/Loop_unrolling

1. 코드 크기 증가

2. 가독성 저해

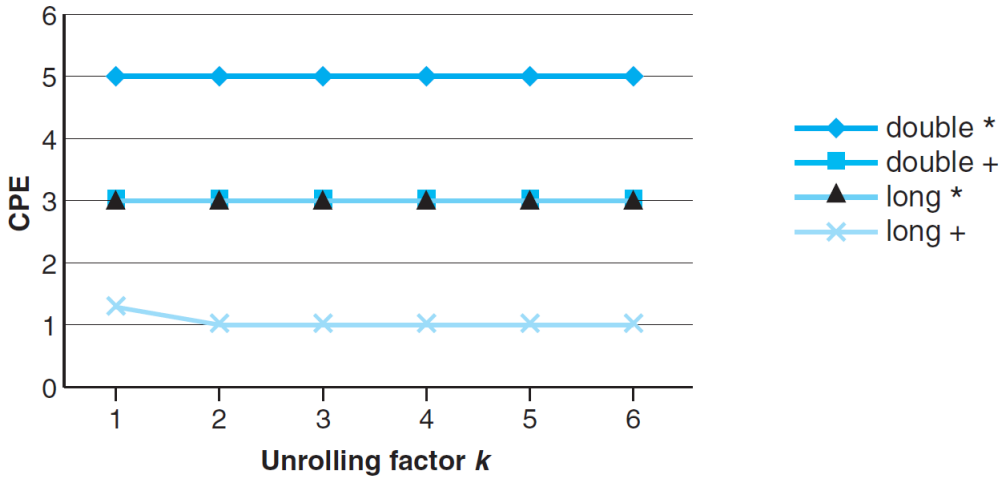


Figure 5.2: kx1 에따른 성능비교

3. 함수호출이 있을 경우 캐시 미스율 향상

연산이 복잡해질수록 인덱스의 계산과 if조건이 수행시간에 영향을 주지않는다. for문 내부가 간단한 코드일때 가장 효과가 좋다.

5.7 Enhancing Parallelism

Multiple Accumulators

연산을 나눠서 계산하고 마지막에 합치는 방식

```
/* 2 x 2 loop unrolling */
void combine6(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc0 = IDENT;
    data_t acc1 = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
    acc0 = acc0 OP data[i];
    acc1 = acc1 OP data[i+1];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
    acc0 = acc0 OP data[i];
    }
```

```
    *dest = acc0 OP acc1;
}
```

Function	Page	Method	Integer +	Integer *	Floating point +	Floating point *
`combine4`	551	Accumulate in temporary	1.27	3.01	3.01	5.01
`combine5`	568	2 × 1 unrolling	1.01	3.01	3.01	5.01
`combine6`	573	2 × 2 unrolling	0.81	1.51	1.51	2.51
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

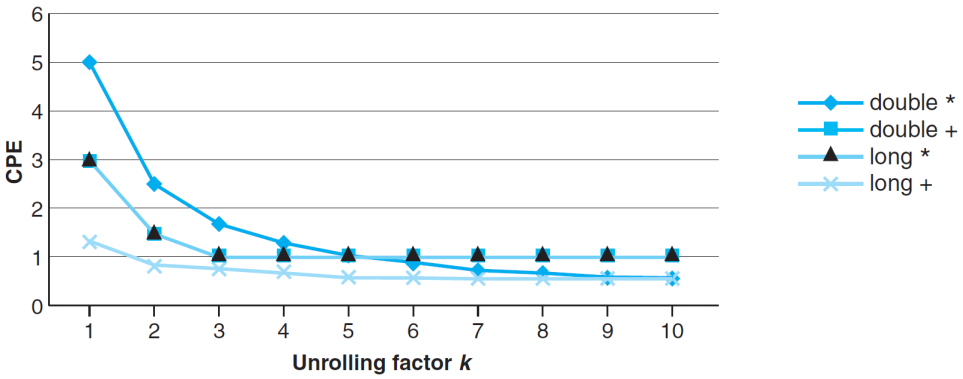


Figure 5.3: kxk

Reassociation Transformation

```
/* 2 x 1a loop unrolling */
void combine7(vec_ptr v, data_t *dest)
{
long i;
long length = vec_length(v);
long limit = length-1;
data_t *data = get_vec_start(v);
data_t acc = IDENT;

/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
acc = acc OP (data[i] OP data[i+1]);
}

/* Finish any remaining elements */
```

```
16 for (; i < length; i++) {
17 acc = acc OP data[i];
18 }
19 *dest = acc;
20 }
```

combine5의 다음코드를

```
1     acc = (acc OP data[i]) OP data[i+1];
```

```
1     acc = acc OP (data[i] OP data[i+1]);
```

로 바꾼다.

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine4`	551	Accumulate in temporary	1.27	3.01	3.01	5.01
`combine5`	568	2 × 1 unrolling	1.01	3.01	3.01	5.01
`combine6`	573	2 × 2 unrolling	0.81	1.51	1.51	2.51
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

Figure 5.4:

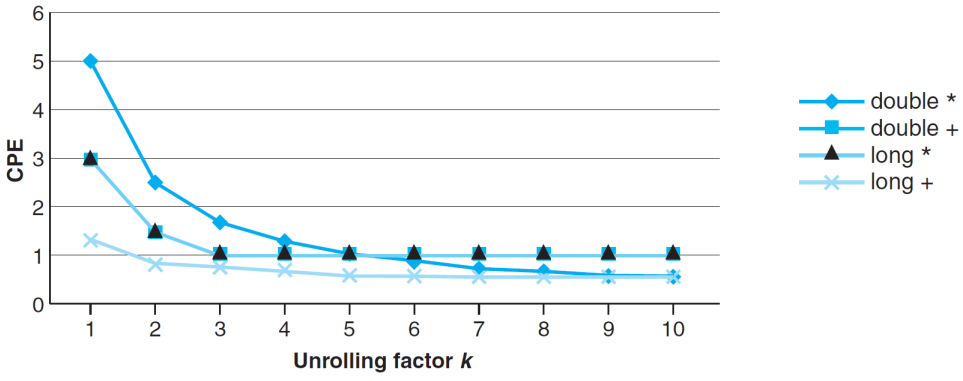


Figure 5.5:

5.8 Summary of Results for Optimizing Combining Code

5.9 Some Limiting Factors

Register Spilling

사용가능한 레지스터 수를 넘긴 병렬성을 갖는다면 값들을 메모리로 넘기는 식을 이용해서 런타임 스택에 공간을 할당한다.

Branch Prediction and Misprediction Penalties

신경쓰지말것

5.10 Understanding Memory Performance

Load Performance

메모리 읽기에대해서 매 클럭사이클마다 유닛갯수만큼 읽을수있기 때문에 하한선이 존재한다.

Store Performance

store 유닛과 load 유닛의 데이터 의존성이 발생할 수 있다. 같은 주소를 바로 저장하고 읽는경우에 성능저하가 나올수있다. 이때 생성되는 연산이 7개의 클럭사이클이 필요.

5.11 Life in the Real World: Performance Improvement Techniques

High-level design. Choose appropriate algorithms and data structures for the problem at hand. Be especially vigilant to avoid algorithms or coding techniques that yield asymptotically poor performance.

Basic coding principles. Avoid optimization blockers so that a compiler can generate efficient code.

- Eliminate excessive function calls. Move computations out of loops when possible. Consider selective compromises of program modularity to gain greater efficiency.
- Eliminate unnecessary memory references. Introduce temporary variables to hold intermediate results. Store a result in an array or global variable only when the final value has been computed.

Low-level optimizations. Structure code to take advantage of the hardware capabilities.

- Unroll loops to reduce overhead and to enable further optimizations.
- Find ways to increase instruction-level parallelism by techniques such as multiple accumulators and reassociation.
- Rewrite conditional operations in a functional style to enable compilation via conditional data transfers.

5.14 Identifying and Eliminating Performance Bottlenecks 598
gprof

```
linux> gcc -Og -pg prog.c -o prog
linux> ./prog file.txt
linux> gprof prog
```

Chapter 6

Part II

Running Programs on a System

Chapter 7

Linker

7.1 Compiler Drivers

```
linux> gcc -Og -o prog main.c sum.c

cc1 /tmp/main.i -Og [other arguments] -o /tmp/main.s
as [other arguments] -o /tmp/main.o /tmp/main.s
ld -o prog [system object files and args] /tmp/main.o /tmp/sum.o
linux> ./prog
```

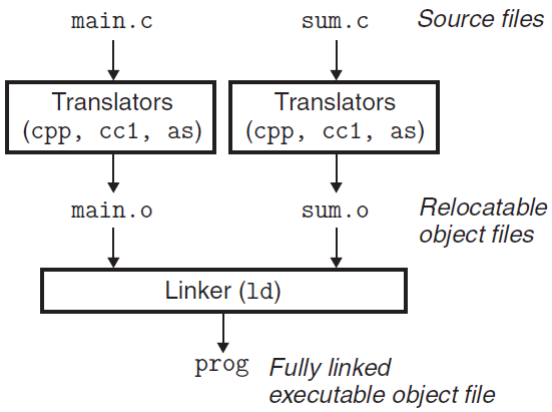


Figure 7.1: Static linking. The linker combines relocatable object files to form an executable object file prog

7.2 Static Linking

about symbol(7.5)

1. symbol resolution(7.6)
2. Relocation(7.7)

7.3 Object Files

Object files are merely collections of blocks of bytes. Some of these blocks contain program code, others contain program data, and others contain data structures that guide the linker and loader. A linker concatenates blocks together, decides on run-time locations for the concatenated blocks, and modifies various locations within the code and data blocks. Linkers have minimal understanding of the target machine. The compilers and assemblers that generate the object files have already done most of the work.

- Relocatable ob (7.4) : compiler,assembler output
- Executable ob (7.8,7.9) linker output
- shared ob (7.10)

7.4 Relocatable Object Files

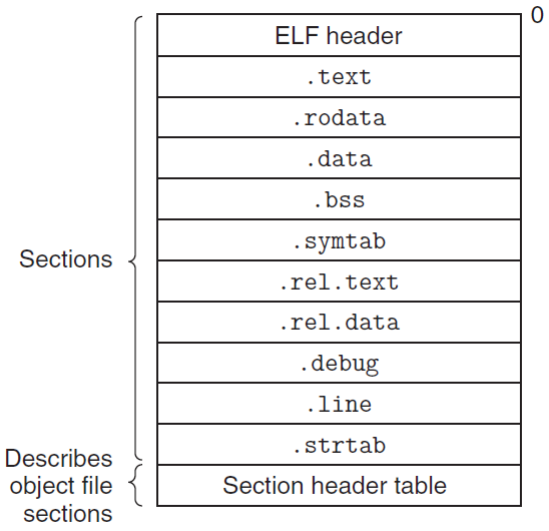


Figure 7.2: Typical ELF relocatable object file.

- **.text** The machine code of the compiled program.

- **.rodata** Read-only data such as the format strings in printf statements, and jump tables for switch statements. .data Initialized global and static C variables. Local C variables are maintained at run time on the stack and do not appear in either the .data or .bss sections.

- **.bss** Uninitialized global and static C variables, along with any global or static variables that are initialized to zero. This section occupies no actual space in the object file; it is merely a laceholder. Object file formats distinguish between initialized and uninitialized variables for space efficiency: uninitialized variables do not have to occupy any actual disk space in the object file. At run time, these variables are allocated in memory with an initial value of zero.

- **.symtab** A symbol table with information about functions and global variables that are defined and referenced in the program. Some programmers mistakenly believe that a program must be compiled with the -g option to get symbol table information. In fact, every relocatable object file has a symbol table in .symtab (unless the programmer has specifically removed it with the strip command). However, unlike the symbol table inside a compiler, the .symtab symbol table does not contain entries for local variables.

- **.rel.text** A list of locations in the .text section that will need to be modified when the linker combines this object file with others. In general, any instruction that calls an external function or references a global variable will need to be modified. On the other hand, instructions that call local functions do not need to be modified. Note that relocation information is not needed in executable object files, and is usually omitted unless the user explicitly instructs the linker to include it.

- **.rel.data** Relocation information for any global variables that are referenced or defined by the module. In general, any initialized global variable whose initial value is the address of a global variable or externally defined function will need to be modified.

- **.debug** A debugging symbol table with entries for local variables and typedefs defined in the program, global variables defined and referenced in the program, and the original C source file. It is only present if the compiler driver is invoked with the -g option.

- **.line** A mapping between line numbers in the original C source program and machine code instructions in the .text section. It is only present if the compiler driver is invoked with the -g option.

- **.strtab** A string table for the symbol tables in the .symtab and .debug sections and for the section names in the section headers. A string table is a sequence of null-terminated character strings.

7.5 Symbols and Symbol Tables

- Global symbols that are defined by module m and that can be referenced by other modules :nonstatic C functions and global variables
- Global symbols that are referenced by module m but defined by some other module : nonstatic C functions and global variables that are defined in other modules.
- Local symbols that are defined and referenced exclusively by module m : nonstatic C functions and global variables that are defined in other modules.

7.6 Symbol Resolution

How Linkers Resolve Duplicate Symbol Name

- strong symbol : 초기화된 전역변수
- weak symbol : 초기화x 전역변수

Rule

1. Multiple strong symbols with the same name are not allowed.
2. Given a strong symbol and multiple weak symbols with the same name, choose the strong symbol.
3. Given multiple weak symbols with the same name, choose any of the weak symbols.

Linking with Static Libraries

라이브러리 생성

```
linux> gcc -c addvec.c multvec.c
linux> ar rcs libvector.a addvec.o multvec.o
```

라이브러리 링킹

```
linux> gcc -c main2.c
linux> gcc -static -o prog2c main2.o ./libvector.a

linux> gcc -c main2.c
linux> gcc -static -o prog2c main2.o -L. -lvector
```

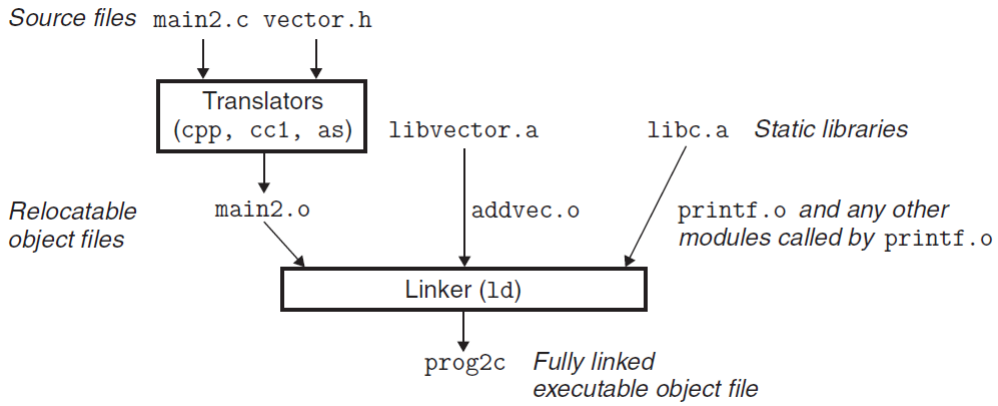


Figure 7.3: Linking with static libraries.

7.7 Relocation

1. Relocating sections and symbol definitionss : 여러개의 .data section 합친다. 이후 인스트럭션과 전역변수들이 런타임 메모리 주소를 가진다.
2. Relocating symbol references within sections. :모든 심볼 참조를 수정한다. 이후 모든 심볼들이 런타임 메모리 주소를 가진다.

Relocation Entries

위치를 모르는 심볼을 어떻게 처리할지에대해 어셈블러가만듬 .rel.data section에 존재

재배치방식

- R_X86_64_PC32 : 상대주소
- R_X86_64_32. : 절대주소

Relocating Symbol References

7.8 Executable Object Files

7.9 Loading Executable Object Files

loading : 실행가능한 목적파일 내의 코드와 데이터를 메모리로 복사하고 첫번째 인스트럭션(엔트리 포인트)으로 점프해서실행하는 과정

7.10 Dynamic Linking with Shared Libraries

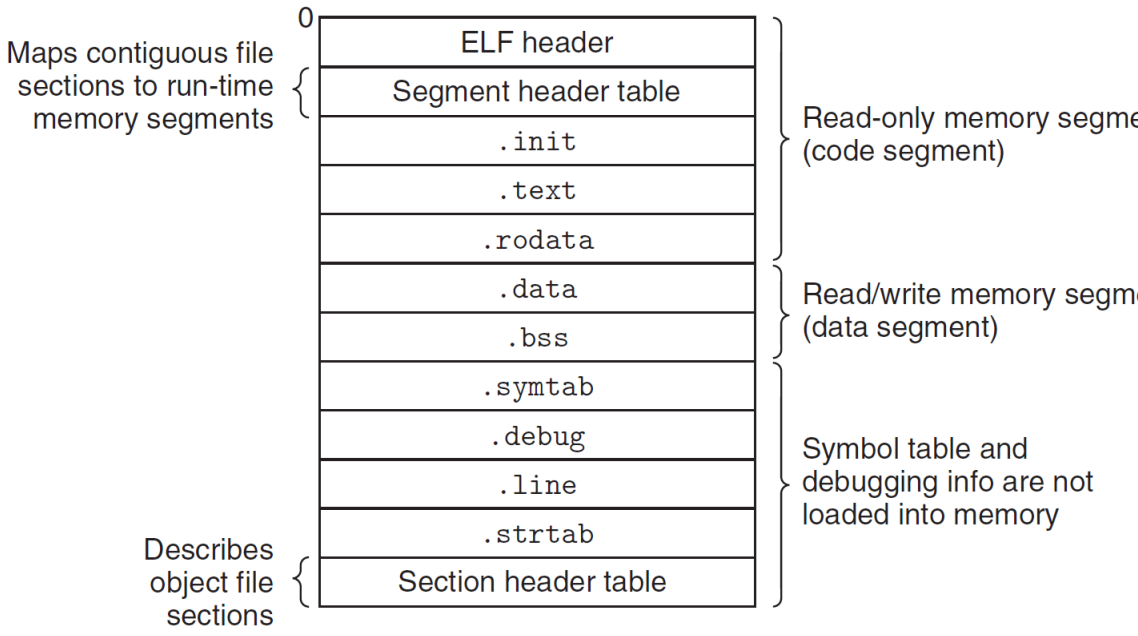


Figure 7.4: Typical ELF executable object file.

```
linux> gcc -shared -fpic -o libvector.so addvec.c multvec.c

    linux> gcc -o prog2l main2.c ./libvector.so
```

7.11 Loading and Linking Shared Libraries from Applications

7.12 Position-Independent Code (PIC)

7.13 Library Interpositioning

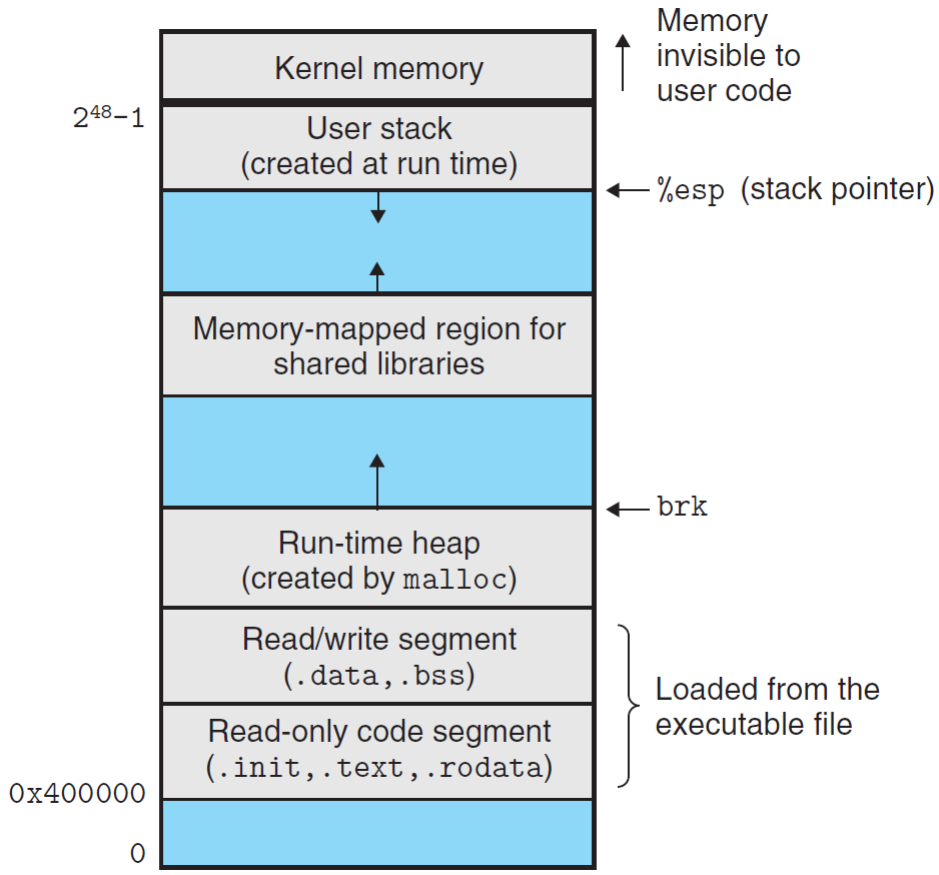


Figure 7.5: **Linux x86-64 run-time memory image.** Gaps due to segment alignment requirements and addressspace layout randomization (ASLR) are not shown. Not to scale

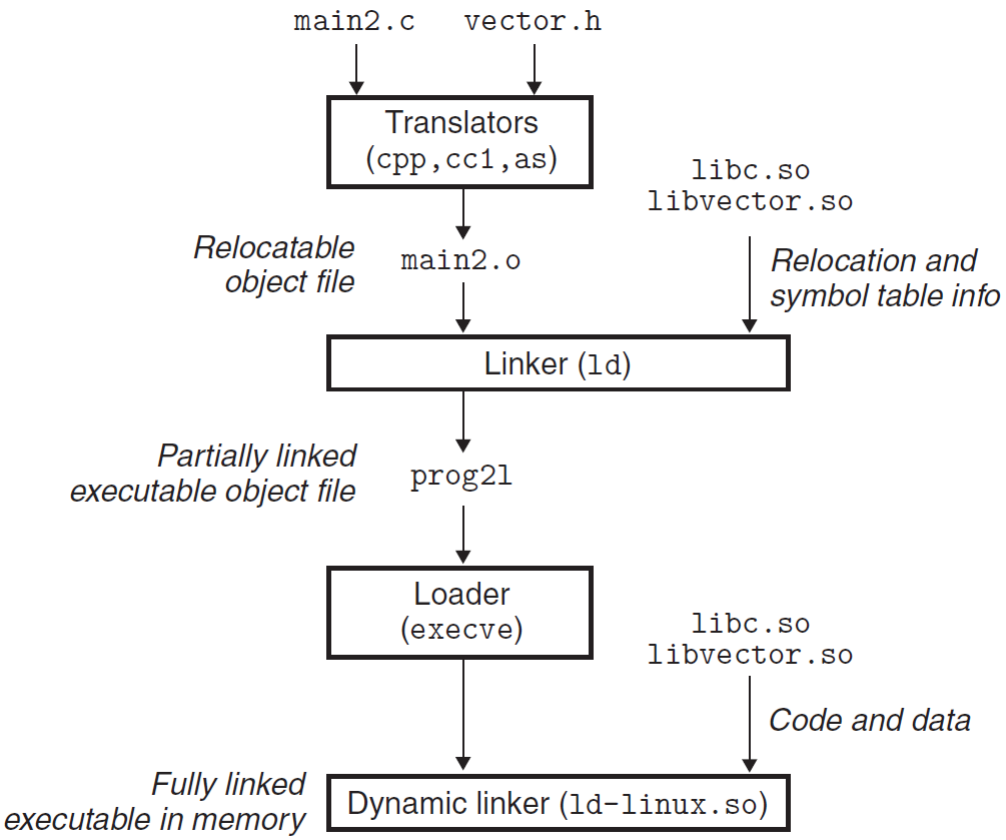


Figure 7.6: Dynamic linking with shared libraries.

Chapter 8

Exceptional Control Flow

8.1 Exceptions

이벤트 발생시에 예외 테이블이라고하는 점프 테이블을 통해서 운영체제에서 핸들러로 프로시저 콜을 하게한다. 그후에 제어를 다음 세가지중 하나로 처리한다.

1. 현재 인스트럭션에 돌려준다
2. 다음 인스트럭션으로 돌려준다
3. 프로그램을 종료한다.

예외에는 하드웨어,소프트웨어 사이에서 각각 일어날 수 있으며 각각의 프로세서,OS 설계자가 예외를 미리 정해놓고 시스템 부팅시에 점프테이블에 할당하여 쓰는식.
예외 종류

- inturrupt(비동기) : I/O에서 시그널 다음 인스트럭션으로 돌려준다.
- trap(동기) : instruction 실행결과. syscall로 OS단에서 처리,다음 인스트럭션으로 돌려준다.
- fault(동기) : ex) 가상 메모리 페이지 오류, 현재 인스트럭션으로 돌려준다.
- abort(동기) : 하드웨어 단의 복구불가능 에러, 프로그램 종료

syscall :
리눅스에서 제공하는 syscall

- read
- write
- open
- close

- stat
- mmap
- brk
- dup2
- pause
- alarm
- getpid
- fork
- execve
- _exit
- sait4
- kill

8.2 Processes

프로세스 생성에는 두가지 종류가있다.

리눅스 프로세스 제어는 unistd.h에 함수가 정의되어 있다.

- fork : 새로운 메모리에 매핑하여 자식프로세스로서 호출
- exevce : 현재 실행되는 프로그램의 메모리에 덮어쓰워 프로세스를 호출

exevce는 실행할 프로그램에 argv(인자리스트)와 envp(환경변수)를 보낼수있다

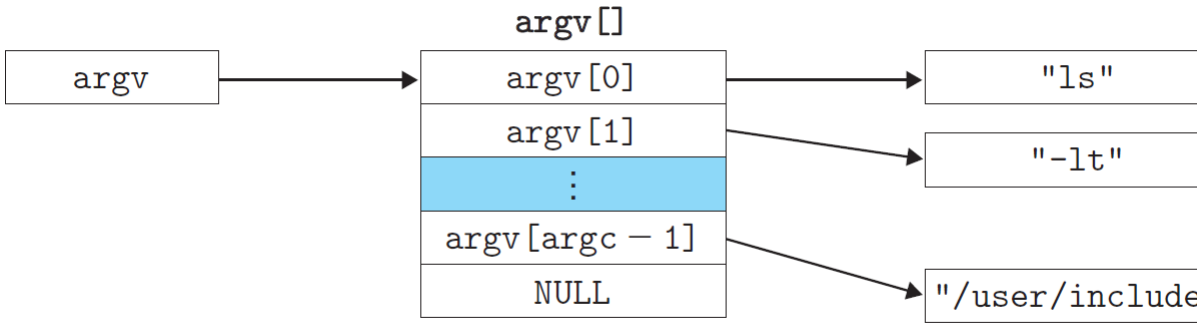


Figure 8.1: Organization of an argument list.

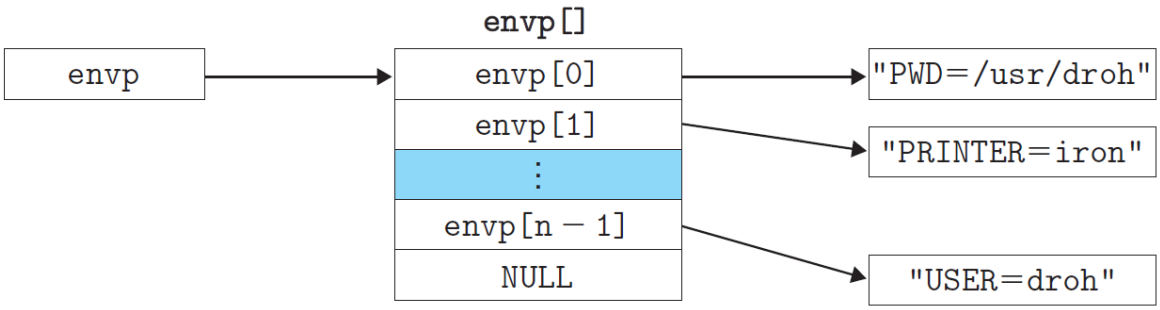


Figure 8.2: Organization of an environment variable list.

8.3 Signals

8.4 Nonlocal Jumps

setjmp longjmp

8.5 Tools for Manipulating Processes

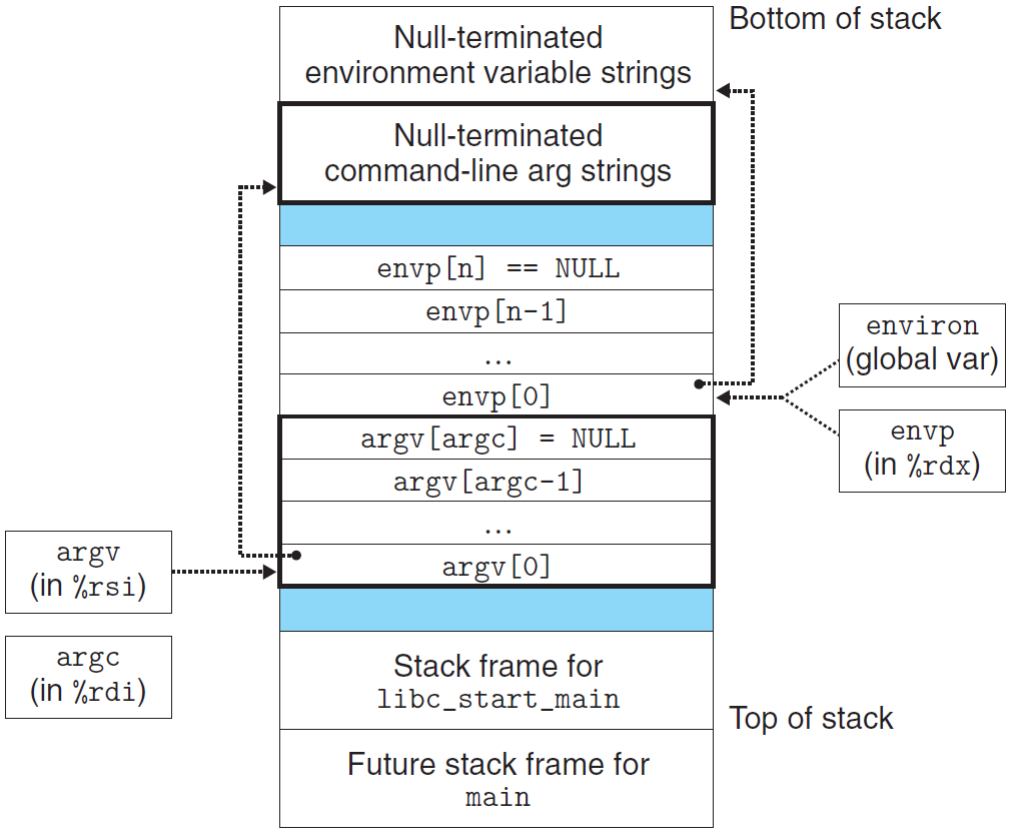


Figure 8.3: Typical organization of the user stack when a new program starts.

Part III

Interaction and Communication between Programs