EUnS's book

EUnS

2019년 11월 22일

차 례

제 1 장

A Tour of Computer Systems

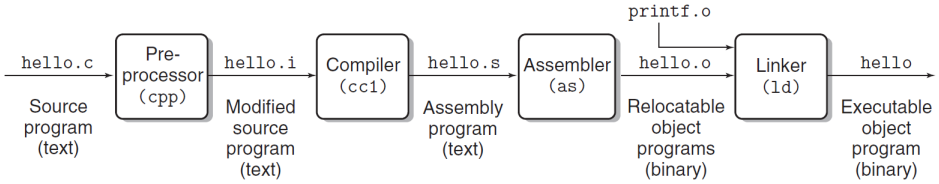


그림 1.1: The compilation system.

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.

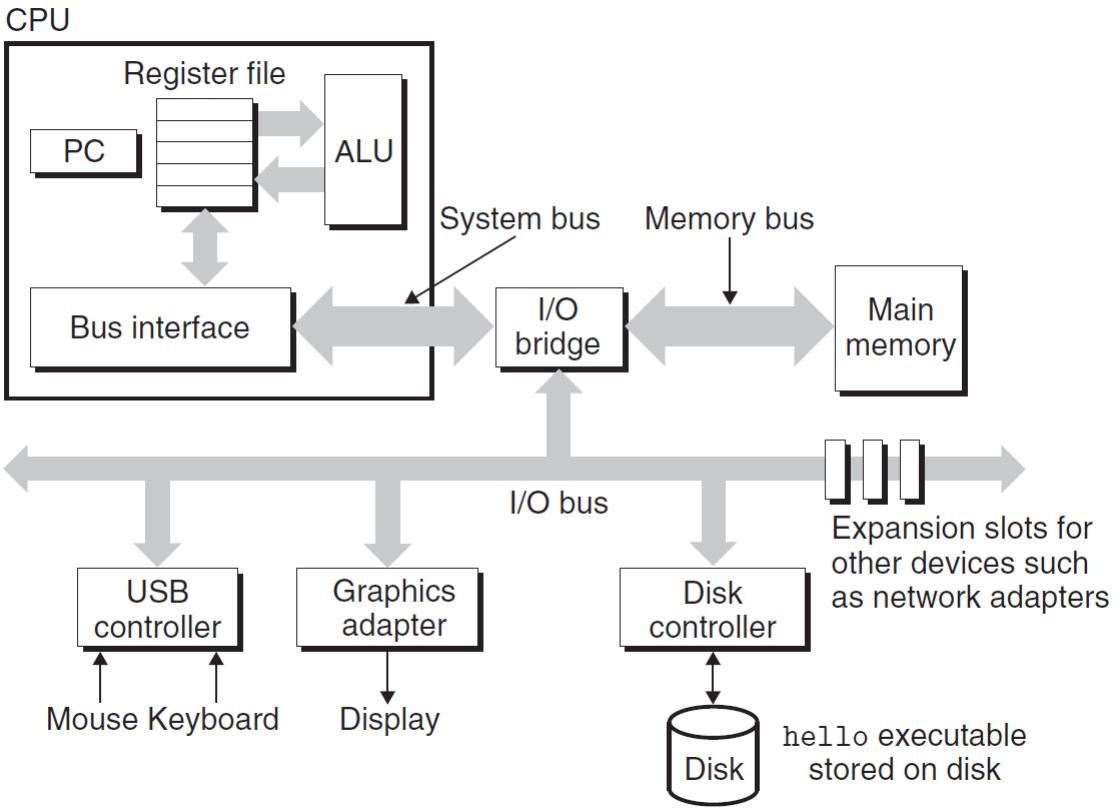


그림 1.2: Hardware organization of a typical system. CPU: central processing unit, ALU: arithmetic/logic unit, PC: program counter, USB: Universal Serial Bus.

- Main Memory

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

- 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

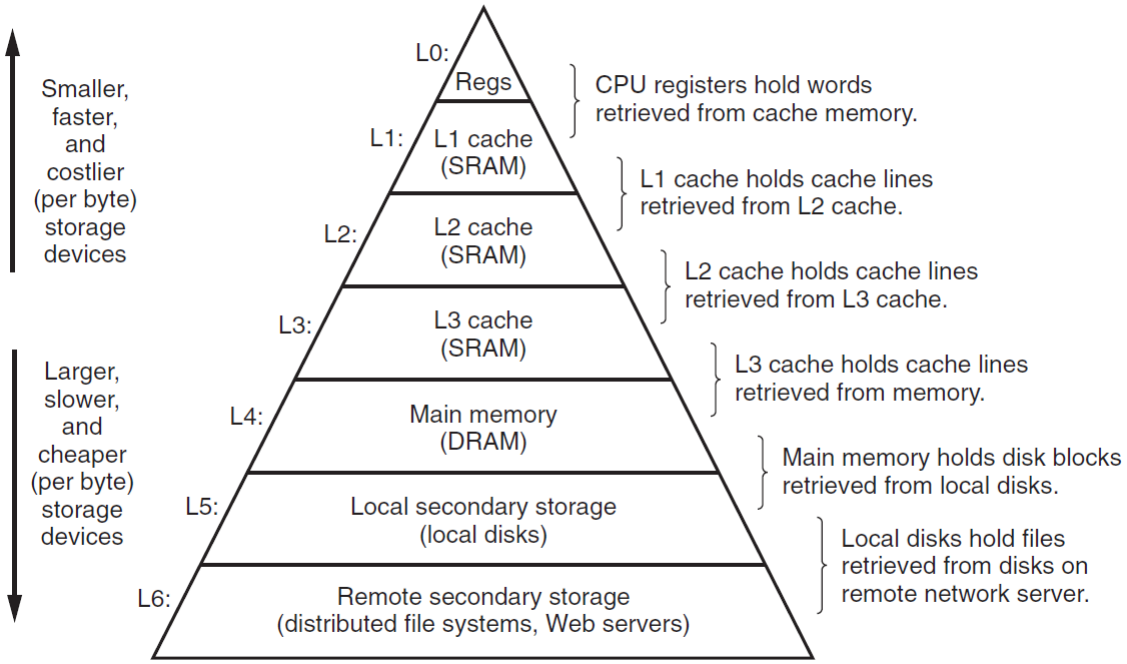


그림 1.3: 메모리 계층

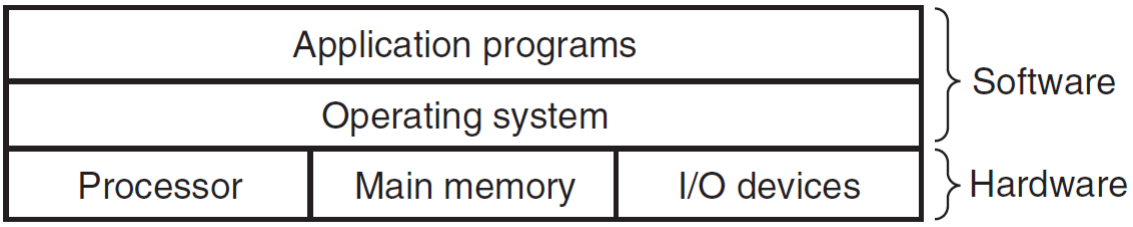


그림 1.4: Layered view of a computer system.

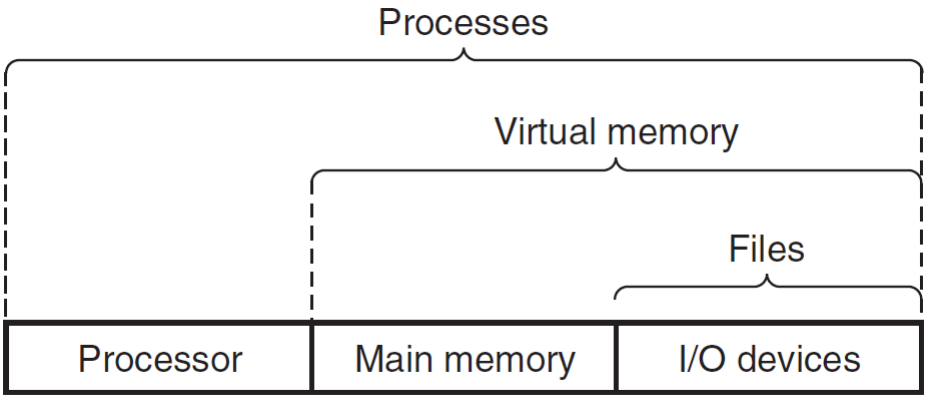


그림 1.5: Abstractions provided by an operating system.

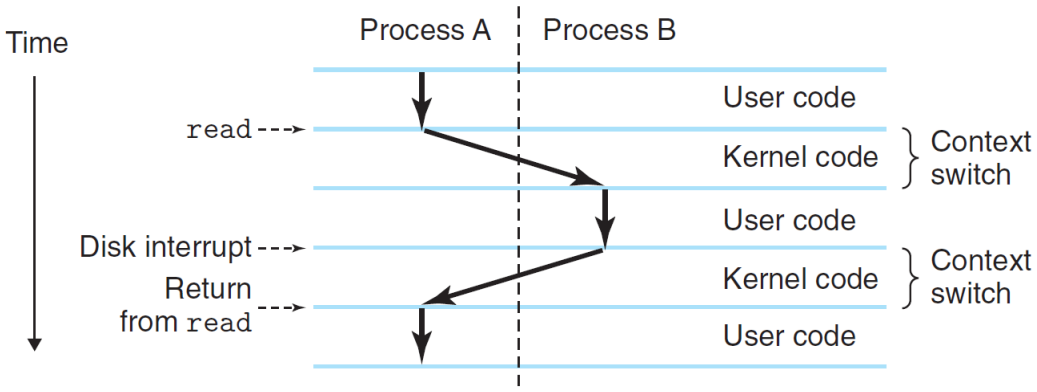


그림 1.6: context switching

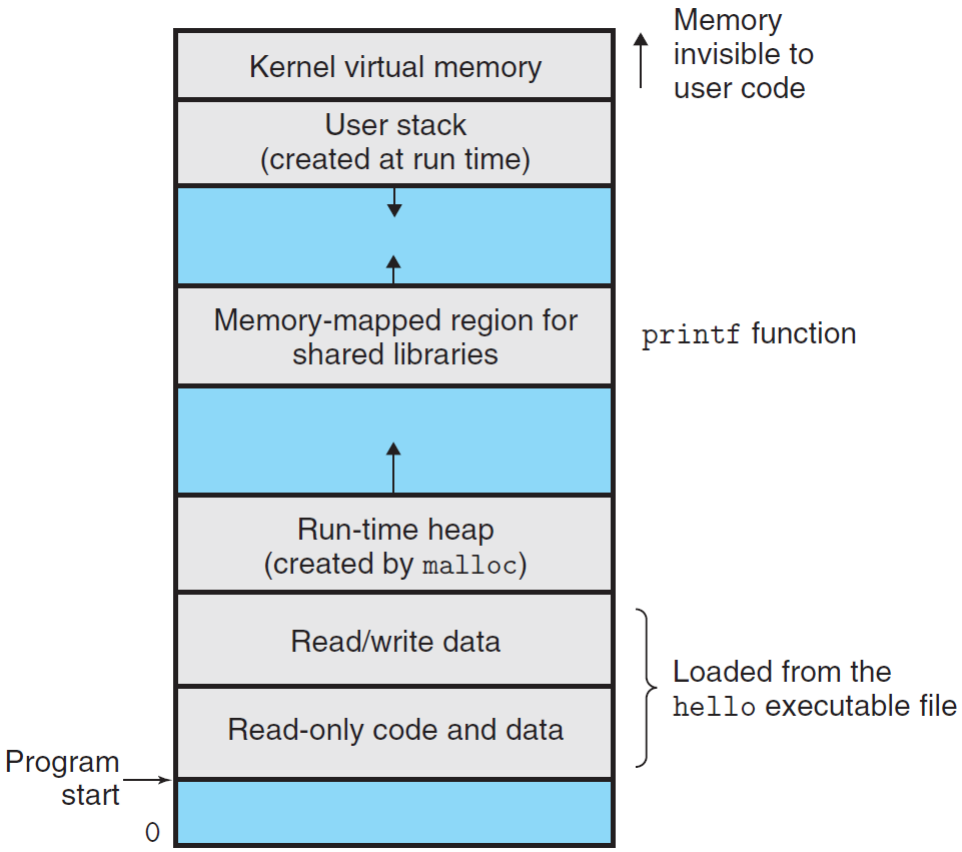


그림 1.7: Process virtual address space.

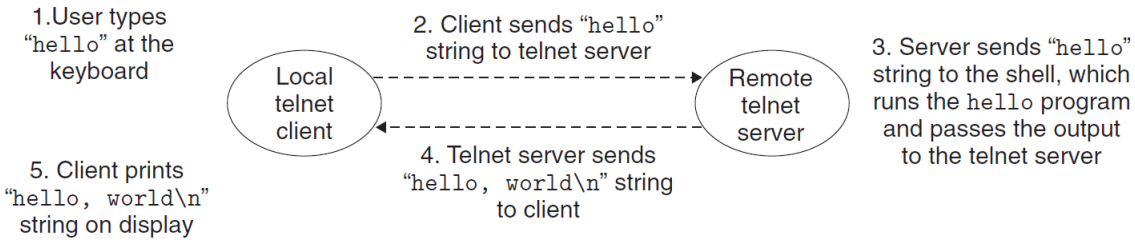


그림 1.8: Using telnet to run hello remotely over a network.

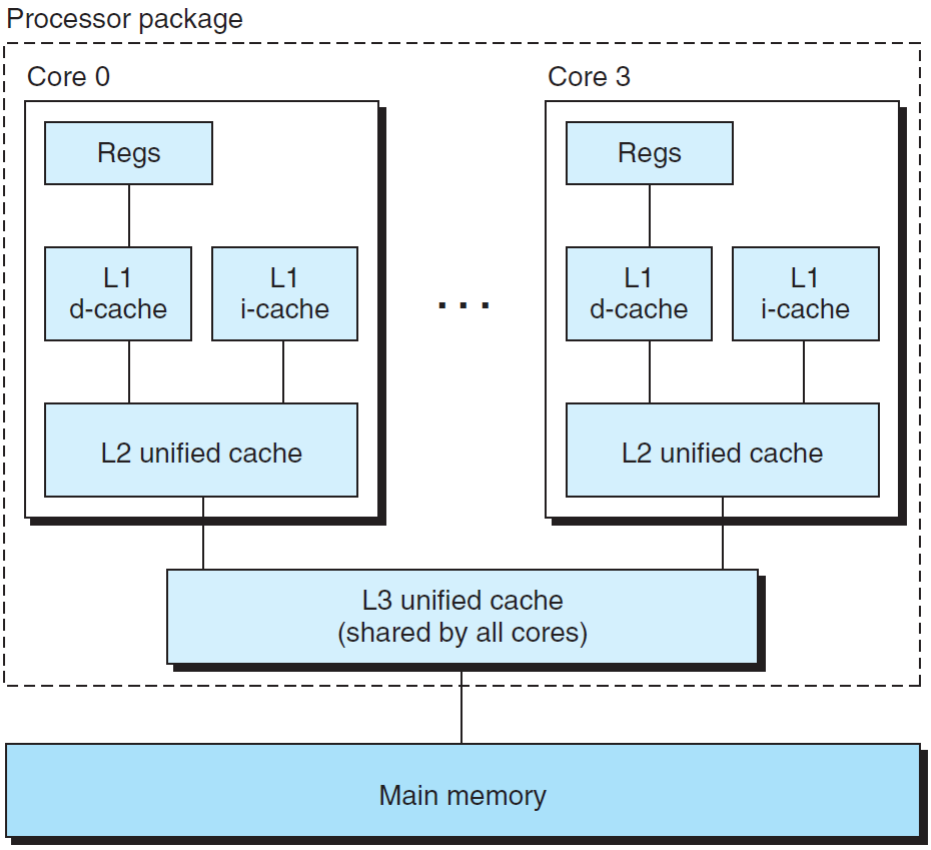


그림 1.9: Multi-core processor organization. Four processor cores are integrated onto a single chip.

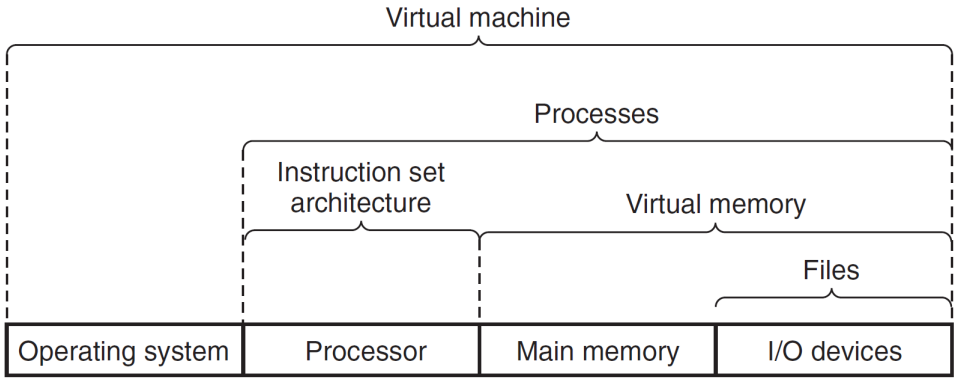


그림 1.10: Some abstractions provided by a computer system

제 I 편

Part I Program Structure and Execution

제 2 장

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.

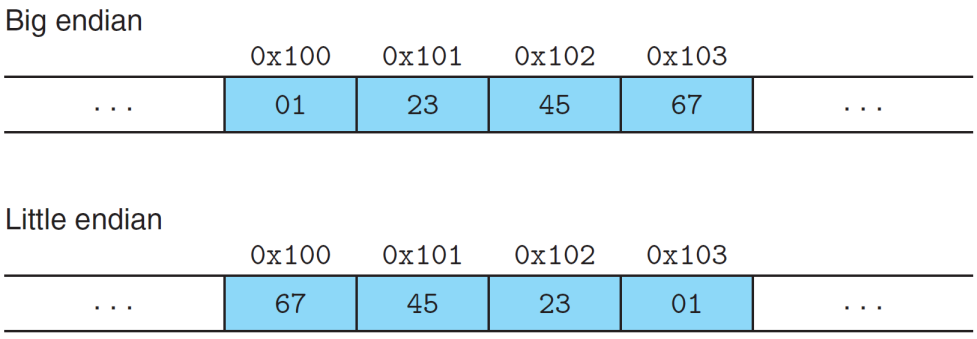


그림 2.1: ex endian

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

```c
#include <stdio.h>
typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, size_t len) {
int i;
for (i = 0; i < len; i++)
printf(" %.2x", start[i]);
printf("\n");
}

void show_int(int x) {
show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x) {
show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x) {
show_bytes((byte_pointer) &x, sizeof(void *));
}


void test_show_bytes(int val) {
    int ival = val;
    float fval = (float) ival;
    int *pval = &ival;
    show_int(ival);
    show_float(fval);
    show_pointer(pval);
    }
```

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.

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

```

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]

제 3 장

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

그림 3.1: ward 0x1234567이 저장되는 방식.

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

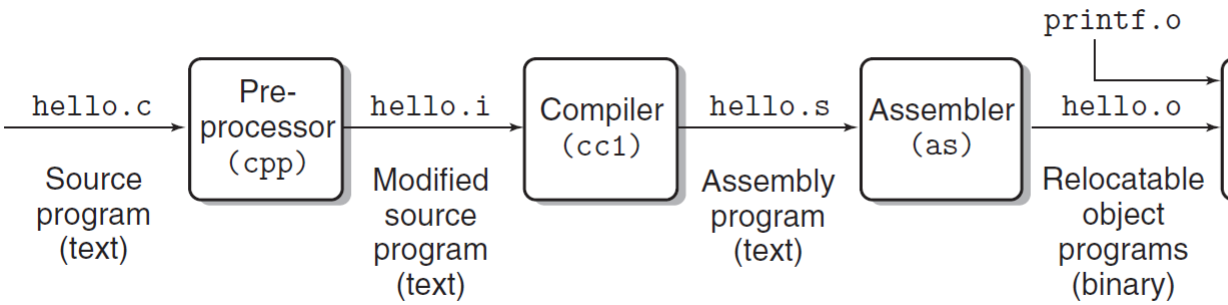


그림 3.2: ward 0x1234567이 저장되는 방식.

제 4 장

Processor Architecture

제 5 장

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.

```
long f();
long func1() {
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;
 }
 }
```

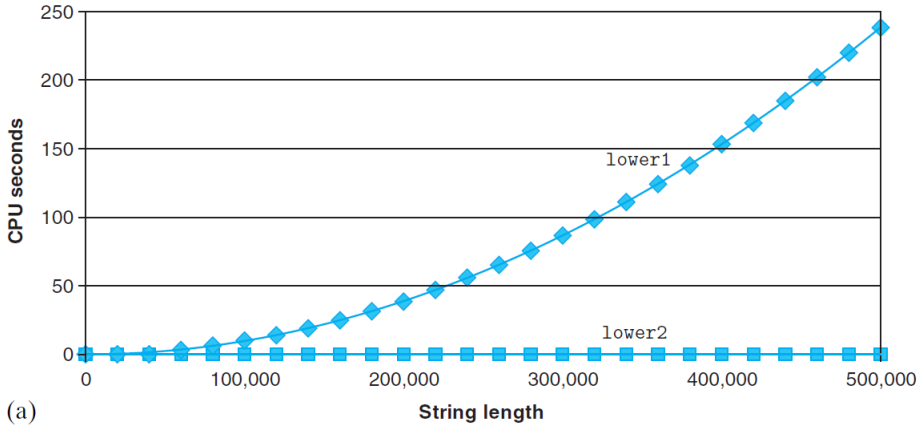
이게 어떻게 성능 개선이되는지 보자.

```
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;
}
```

시간복잡도 계산으로도 충분히 알 수 있다. 문자열 길이가 변하는게 아니라면 strlen을 반복문안에 넣는 짓은 하지말자.



(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)

그림 5.1: lower 성능비교

5.3 Reducing Procedure Calls

```
1  void combine3(vec_ptr v, data_t *dest)
2  {
3      long i;
4      long length = vec_length(v);
5      data_t *data = get_vec_start(v);
6      *dest = IDENT;
7      for (i = 0; i < length; i++) {
8      *dest = *dest OP data[i];
9      }
10 }
```

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

5.4 Eliminating Unneeded Memory References

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

Function	Page	Method	Integer +	Integer *	Floating point +	Floating point *
`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

```
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;
 }
```

Function	Page	Method	Integer +	Integer *	Floating point +	Floating point *
`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.

```
/* 2 x 1 loop unrolling */
void combine5(vec_ptr v, data_t *dest)
    {
    long i;
    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/œ/loop_unrolling`
`https://en.wikipedia.org/wiki/Loop_unrolling`

1. 코드 크기 증가

2. 가독성 저해

3. 함수호출이 있을 경우 캐시 미스율 향상

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

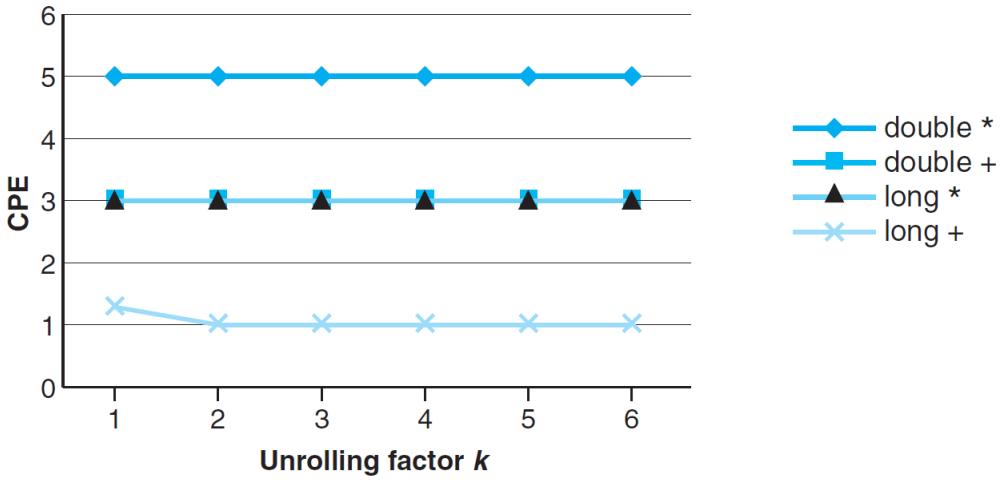


그림 5.2: kx1 에따른 성능비교

5.7 Enhancing Parallelism

Multiple Accumulators

연산을 나눠서 계산하고 마지막에 합치는 방식

```
1  /* 2 x 2 loop unrolling */
2  void combine6(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 acc0 = IDENT;
9      data_t acc1 = IDENT;
10
11     /* Combine 2 elements at a time */
12     for (i = 0; i < limit; i+=2) {
13     acc0 = acc0 OP data[i];
14     acc1 = acc1 OP data[i+1];
15     }
16
17     /* Finish any remaining elements */
18     for (; i < length; i++) {
19     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

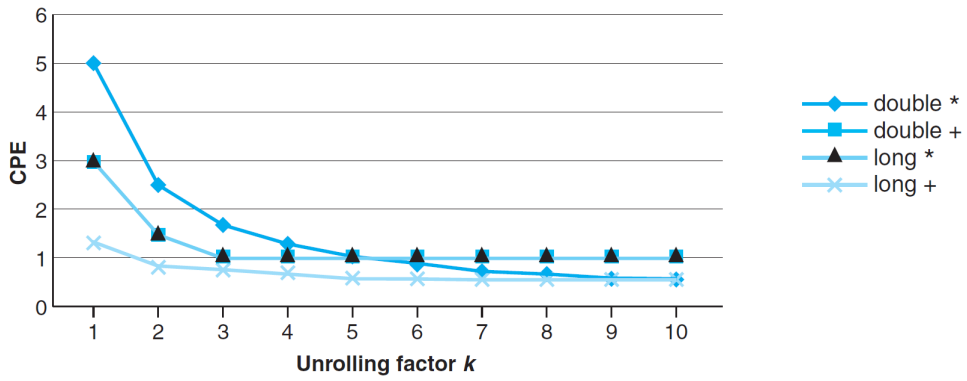


그림 5.3: kxk

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

combine5의 다음코드를

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

그림 5.4:

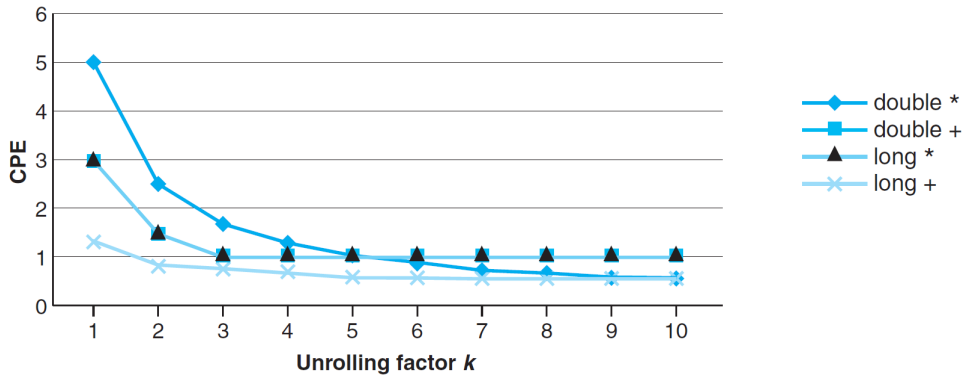

6
5
4
3
2
1
0
CPE
1
2
3
4
5
6
7
8
9
10
Unrolling factor k
double *
double +
long *
long +

그림 5.5: