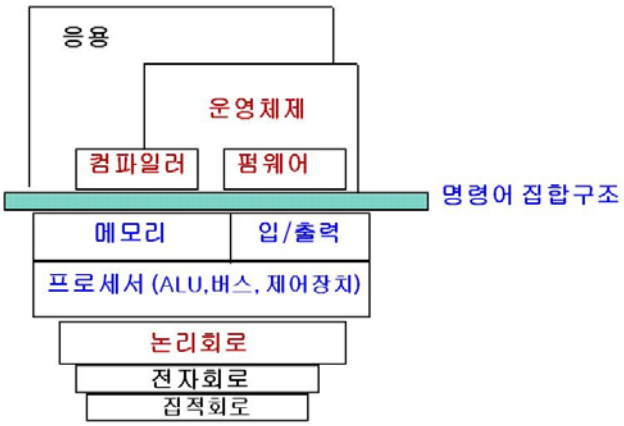
1장. 컴퓨터 시스템으로의 여행

컴퓨터 시스템 (Computer System)

- 컴퓨터 시스템은 하드웨어와 시스템 소프트웨어로 구성되고, 이들이 함께 작동하여 응용 프로그램을 실행
  - **하드웨어: 프로세서, 명령어 집합, 메모리, 입출력장치**
  - 시스템 소프트웨어: 운영체제, 링커, 컴파일러
  - 네트워크



hello 프로그램

- 아래의 hello.c 프로그램 예에 대해 컴퓨터 시스템의 동작 및 실행 과정 소개

```
#include <stdio.h>

int main()
{
    printf("Hello, world₩n");
    return 0;
}
```


1.1 정보는 비트와 컨텍스트로 이루어진다

소스 프로그램 (Source Program)

- 프로그래머가 작성한 소스 프로그램(또는 소스 파일)은 텍스트 파일(text file)로 저장
  - 텍스트 문자는 아스키(ASCII) 코드로 저장
    - 컴퓨터는 1,0 만을 저장하므로 문자는 미리 정의된 2진 비트(코드)으로 표현
    - 한글의 경우 유니코드(Unicode)
  - 그림 1.2: hello.c의 아스키 코드 표시

- 컴퓨터 시스템 내부의 정보 (디스크 파일, 메모리 상의 데이터)는 모두 동일하게 비트로 표시
  - 해석하는 내용(context, 컨텍스트)에 의해 구분
  - 정수, 부동소수, 문자열

ASCII 코드

- ASCII 코드
  - American Standard Code for Information Interchange
  - 미국에서 표준화한 정보교환용 7비트 부호체계
  - 000(0x00)부터 127(0x7F)까지 총 128개의 부호가 사용

baekwlgns.tistory.com | 제어 문자 | 공백 문자 | 구두점 | 숫자 | 알파벳 | baekwlgns.tistory.com

ASCII	10진수	2진수	8진수	16진수	ASCII	10진수	2진수	8진수	16진수	ASCII	10진수	2진수	8진수	16진수	ASCII	10진수	2진수	8진수	16진수
NULL	0	0	0	0x00	SP	32	100000	40	0x20	@	64	1000000	100	0x40	`	96	1100000	140	0x60
SOH	1	1	1	0x01	!	33	100001	41	0x21	A	65	1000001	101	0x41	a	97	1100001	141	0x61
STX	2	10	2	0x02	큰"	34	100010	42	0x22	B	66	1000010	102	0x42	b	98	1100010	142	0x62
ETX	3	11	3	0x03	#	35	100011	43	0x23	C	67	1000011	103	0x43	c	99	1100011	143	0x63
EOT	4	100	4	0x04	$	36	100100	44	0x24	D	68	1000100	104	0x44	d	100	1100100	144	0x64
ENQ	5	101	5	0x05	%	37	100101	45	0x25	E	69	1000101	105	0x45	e	101	1100101	145	0x65
ACK	6	11	6	0x06	&	38	100110	46	0x26	F	70	1000110	106	0x46	f	102	1100110	146	0x66
BEL	7	111	7	0x07	작은'	39	100111	47	0x27	G	71	1000111	107	0x47	g	103	1100111	147	0x67
BS	8	1000	10	0x08	(	40	101000	50	0x28	H	72	1001000	110	0x48	h	104	1101000	150	0x68
HT	9	1001	11	0x09	)	41	101001	51	0x29	I	73	1001001	111	0x49	i	105	1101001	151	0x69
LF	10	1010	12	0x0A	*	42	101010	52	0x2A	J	74	1001010	112	0x4A	j	106	1101010	152	0x6A
VT	11	1011	13	0x0B	+	43	101011	53	0x2B	K	75	1001011	113	0x4B	k	107	1101011	153	0x6B
FF	12	1100	14	0x0C	,	44	101100	54	0x2C	L	76	1001100	114	0x4C	l	108	1101100	154	0x6C
CR	13	1101	15	0x0D	-	45	101101	55	0x2D	M	77	1001101	115	0x4D	m	109	1101101	155	0x6D
SO	14	1110	16	0x0E	.	46	101110	56	0x2E	N	78	1001110	116	0x4E	n	110	1101110	156	0x6E
SI	15	1111	17	0x0F	/	47	101111	57	0x2F	O	79	1001111	117	0x4F	o	111	1101111	157	0x6F
DLE	16	10000	20	0x10	0	48	110000	60	0x30	P	80	1010000	120	0x50	p	112	1110000	160	0x70
DC1	17	10001	21	0X11	1	49	110001	61	0x31	Q	81	1010001	121	0x51	q	113	1110001	161	0x71
DC2	18	10010	22	0x12	2	50	110010	62	0x32	R	82	1010010	122	0x52	r	114	1110010	162	0x72
DC3	19	10011	23	0x13	3	51	110011	63	0x33	S	83	1010011	123	0x53	s	115	1110011	163	0x73
DC4	20	10100	24	0x14	4	52	110100	64	0x34	T	84	1010100	124	0x54	t	116	1110100	164	0x74
NAK	21	10101	25	0x15	5	53	110101	65	0x35	U	85	1010101	125	0x55	u	117	1110101	165	0x75
SYN	22	10110	26	0x16	6	54	110110	66	0x36	V	86	1010110	126	0x56	v	118	1110110	166	0x76
ETB	23	10111	27	0x17	7	55	110111	67	0x37	W	87	1010111	127	0x57	w	119	1110111	167	0x77
CAN	24	11000	30	0x18	8	56	111000	70	0x38	X	88	1011000	130	0x58	x	120	1111000	170	0x78
EM	25	11001	31	0x19	9	57	111001	71	0x39	Y	89	1011001	131	0x59	y	121	1111001	171	0x79
SUB	26	11010	32	0x1A	:	58	111010	72	0x3A	Z	90	1011010	132	0x5A	z	122	1111010	172	0x7A
ESC	27	11011	33	0x1B	;	59	111011	73	0x3B	[	91	1011011	133	0x5B	{	123	1111011	173	0x7B
FS	28	11100	34	0x1C	<	60	111100	74	0x3C	₩	92	1011100	134	0x5C	|	124	1111100	174	0x7C
GS	29	11101	35	0x1D	=	61	111101	75	0x3D	]	93	1011101	135	0x5D	}	125	1111101	175	0x7D
RS	30	11110	36	0x1E	>	62	111110	76	0x3E	^	94	1011110	136	0x5E	~	126	1111110	176	0x7E
US	31	11111	37	0x1F	?	63	111111	77	0x3F	_	95	1011111	137	0x5F	DEL	127	1111111	177	0x7F

유니코드 (Unicode)

- 유니코드는 전 세계 문자를 컴퓨터로 표현하는 표준
  - 31 비트 코드, 대부분 21 비트로 표현
    - U+16진수로 표시
  - ASCII (영문, 숫자, 기호): U+00 – U+7F
    ex) U+41 A, ……., U+7A z
  - 한글 코드 범위: U+AC00 – U+D7AF
    ex) U+AC00 가, U+AC01 각, ……., U+D7A3 힣

- http://ko.wikipedia.org/wiki/유니코드 참조

한글 유니코드

U+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
AC00	가	각	갂	갃	간	갅	갆	갇	갈	갉	갊	갋	갌	갍	갎	갏
AC10	감	갑	값	갓	갔	강	갖	갗	갘	같	갚	갛	개	객	갞	갟
AC20	갠	갡	갢	갣	갤	갥	갦	갧	갨	갩	갪	갫	갬	갭	갮	갯
AC30	갰	갱	갲	갳	갴	갵	갶	갷	갸	갹	갺	갻	갼	갽	갾	갿
AC40	걀	걁	걂	걃	걄	걅	걆	걇	걈	걉	걊	걋	걌	걍	걎	걏
AC50	걐	걑	걒	걓	개	객	걖	걗	갱	걙	걚	걛	걜	걝	걞	걟

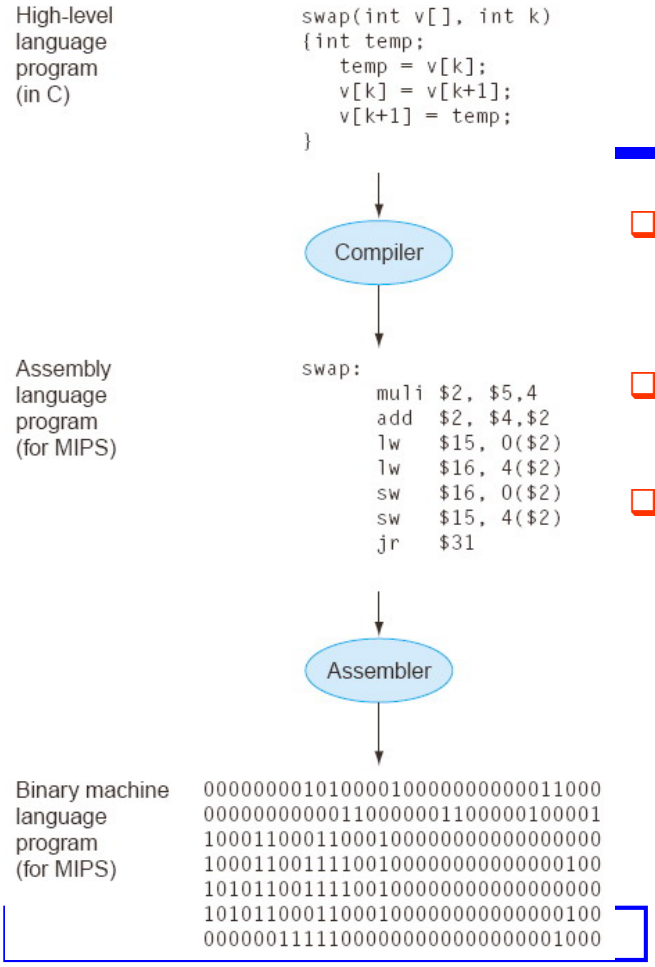
……

U+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D700	휀	휁	휂	휃	휄	휅	휆	휇	휈	휉	휊	휋	휌	휍	휎	휏
D710	휐	휑	휒	휓	휔	휕	휖	휗	휘	휙	휚	휛	휜	휝	휞	휟
D720	휠	휡	휢	휣	휤	휥	휦	휧	휨	휩	휪	휫	휬	휭	휮	휯
D730	휰	휱	휲	휳	휴	휵	휶	휷	휸	휹	휺	휻	휼	휽	휾	휿
D740	흀	흁	흂	흃	흄	흅	흆	흇	흈	흉	흊	흋	흌	흍	흎	흏
D750	흐	흑	흒	흓	흔	흕	흖	흗	흘	흙	흚	흛	흜	흝	흞	흟
D760	흠	흡	흢	흣	흤	흥	흦	흧	흨	흩	흪	흫	희	흭	흮	흯
D770	흰	흱	흲	흳	흴	흵	흶	흷	흸	흹	흺	흻	흼	흽	흾	흿
D780	힀	힁	힂	힃	힄	힅	힆	힇	히	힉	힊	힋	힌	힍	힎	힏
D790	힐	힑	힒	힓	힔	힕	힖	힗	힘	힙	힚	힛	힜	힝	힞	힟
D7A0	힠	힡	힢	힣												

1.2 프로그램은 다른 프로그램에 의해 다른 형태로 번역된다

기계어 변환

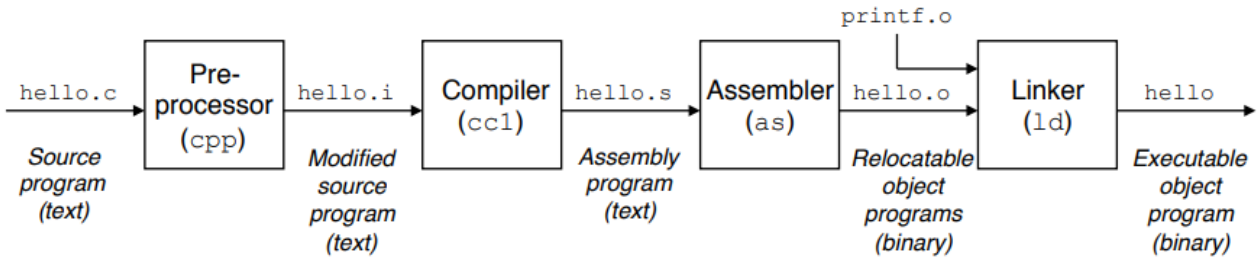
- C 소스 프로그램은 컴파일러에 의해 기계어 명령어(machine instruction)으로 변환되어 실행
  - 변환된 프로그램은 실행 가능한 목적 프로그램(object program, 또는 목적 파일)으로 디스크 파일에 저장
  - 컴파일러는 소스파일을 실행가능한 목적 파일로 번역
- 리눅스의 GCC 컴파일러 예
  - linux> gcc -o hello hello.c
  - 소스파일 hello.c를 읽어서 실행 파일인 hello 생성
- GCC 컴파일 시스템은 네 단계를 수행하여 번역
  - 전처리기, 컴파일러, 어셈블러, 링커



C,어셈블리, 이진코드

- 고급언어
  - 높은 생산성을 유지할 수 있는 상위수준의 언어
- 어셈블리 언어
  - 명령어를 기호로 표시
- 기계어
  - 이진자릿수, 비트로 표현
  - 명령어를 인코드, 데이타

컴파일 시스템 (Compile System)



<그림 1.3>

- 전처리 (preprocess) 단계
  - 전처리기(cpp)는 소스 C 프로그램의 # 지시어(directive)에 따라 처리
  - hello.c의 #include <stdio.h> 지시어는 전처리기에게 시스템 헤더 파일 stdio.h의 삽입을 지시
  - 삽입된 처리 결과는 hello.i 로 생성

컴파일 시스템 – 컴파일 (1)

❑ 컴파일 (compile) 단계

- 컴파일러(cc1)는 hello.i를 어셈블리어 프로그램으로 변환하여 hello.s 를 생성
  - 아래에서 각 줄이 한 개의 저수준 어셈블리 명령어를 텍스트 형태로 표시

```
.......
main:
    subq    $8, %rsp
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
    addq    $8, %rsp
    ret
.......
```


컴파일 시스템 – 컴파일 (2)

- linux> gcc -S -O hello.c
  linux> cat hello.s

- -O 옵션
  - 최적화 레벨 1
- -g 옵션
  - 디버깅 옵션
    - 목적파일에 디버깅 정보 포함

```
linux> gcc -S -O hello.c
linux> cat hello.s
        .file   "hello.c"
        .text
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "Hello. world!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB23:
        .cfi_startproc
        subq    $8, %rsp
        .cfi_def_cfa_offset 16
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $0, %eax
        addq    $8, %rsp
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE23:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0"
        .section        .note.GNU-stack,"",@progbits
linux>
```

컴파일 시스템 – 어셈블리

❑ 어셈블리 (assembly) 단계

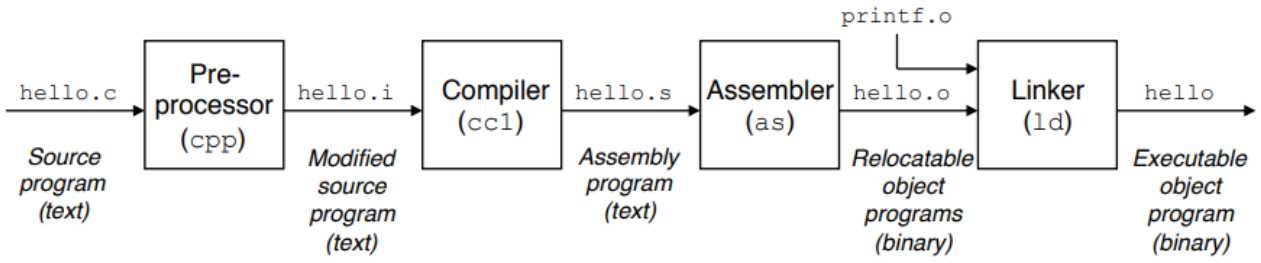
- 어셈블러(as)가 어셈블리 명령어를 기계어 명령어(코드)로 변환
- 메모리에 재배치 가능한 프로그램(relocatable program) 형태의 hello.o의 목적 파일 생성
  - main 함수의 기계어 명령어를 인코딩한 바이너리 파일(binary file)
  - 텍스트 편집기로는 읽을 수 없고 리눅스의 objdump 명령으로 역 어셈블(disassemble, –d)하여 표시
    - linux> gcc -c -O hello.c
      linux> objdump -d hello.o

```
linux> gcc -c -O hello.c
linux> objdump -d hello.o

hello.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # b <main+0xb>
   b:   e8 00 00 00 00          callq  10 <main+0x10>
  10:   b8 00 00 00 00          mov    $0x0,%eax
  15:   48 83 c4 08             add    $0x8,%rsp
  19:   c3                      retq
linux>
```


컴파일 시스템 – 링크

❑ 링크 (link) 단계

- 링커 프로그램(ld)이 라이브러리 등을 결합하여 메모리에 적재되어 실행되는 실행 가능한 목적파일(실행 파일)을 생성
- 표준 C 라이브러리의 printf 함수의 목적 파일 printf.o와 hello.o를 통합한 실행 파일 hello를 생성
- 이 실행 파일은 메모리에 적재(load)되어 시스템에 의해 실행



1.3 컴파일 시스템이 어떻게 동작하는지 이해하는 것은 중요하다

컴파일 시스템 동작 이해 (1)

- 프로그래머 관점에서 아래와 같은 컴퓨터 시스템의 동작 이해가 중요

- 프로그램 성능 최적화하기
  - 성능이 최적화된 효율적인 프로그램 작성을 위해 기계어 수준 코드에 대한 기본적이 이해 필요
    - 3장에서 x86-64 기계어 동작
    - *5장에서 프로그램 최적화하기* 소개
- 링크 에러 이해하기
  - 대규모 소프트웨어 시스템 빌드 시 발생하는 링커의 동작과 관련된 프로그래밍 에러 이해
    - *7장에서 링커* 소개

컴파일 시스템 동작 이해 (2)

- 보안 약점 (security hole) 피하기
  - 안전한 프로그래밍을 위해 스택에 데이터와 제어 정보가 저장되는 방식의 이해가 필요
    - 버퍼 오버플로우(buffer overflow) 취약성이 인터넷과 네트워크 상의 보안 약점의 주요 원인
    - *3장에서 스택과 버퍼 오버플로우 취약성을 다룸*

1.4 프로세서는 메모리에 저장된 명령어를 읽고 해석한다

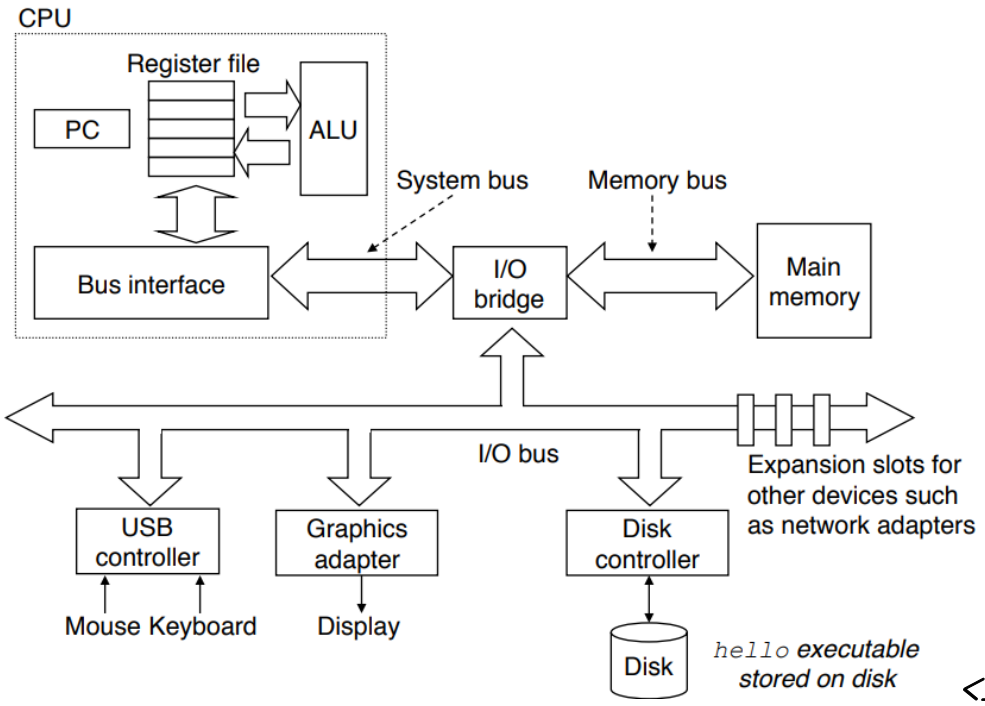
프로그램 실행

- 리눅스 시스템에서 실행파일을 수행
  - 운영체제가 제공하는 명령어 해석기(comman interpreter)인 쉘(shell) 프로그램에서 리눅스 명령어 라인을 입력 받아 명령을 실행

  - linux> ./hello
    Hello, world
    linux>

```
linux> ls hello*
hello  hello.c  hello.o  hello.s
linux> ./hello
Hello. world!
linux>
```


컴퓨터 시스템 하드웨어 구성



<그림 1.4>

하드웨어 구성 – 버스, 입출력 장치

❑ 버스 (Bus)

- 하드웨어 구성 요소들 간의 정보를 전달하는 배선
- 일반적으로 워드(word) 단위라는 고정된 크기의 바이트들 단위로 전송
  - 대부분의 현대 컴퓨터 시스템은 4 바이트 (32 비트) 또는 8 바이트 (64 비트)의 워드 크기를 가짐

❑ 입출력 장치

- 컴퓨터 시스템과 외부 장치와의 연결을 담당
  - 키보드, 디스플레이, 디스크 드라이브 등과 연결
- 입출력 장치는 입출력 버스, 컨트롤러(controller), 어댑터(adaptor) 등을 통해 연결
  - 컨틀롤러는 디바이스 자체가 칩셋이거나 시스템의 마더보드에 장착
  - 어댑터는 마더보드의 슬롯에 장착되는 카드

하드웨어 구성 – 메인 메모리, 프로세서

❑ 메인 메모리 (Main Memory)

- 실행되는 프로그램의 데이터와 프로그램이 저장
- 물리적으로는 DRAM 칩으로 구성되고, 논리적으로는 연속된 바이트의 배열로 고유의 주소(배열의 인덱스)를 가짐

❑ 프로세서 (Processor)

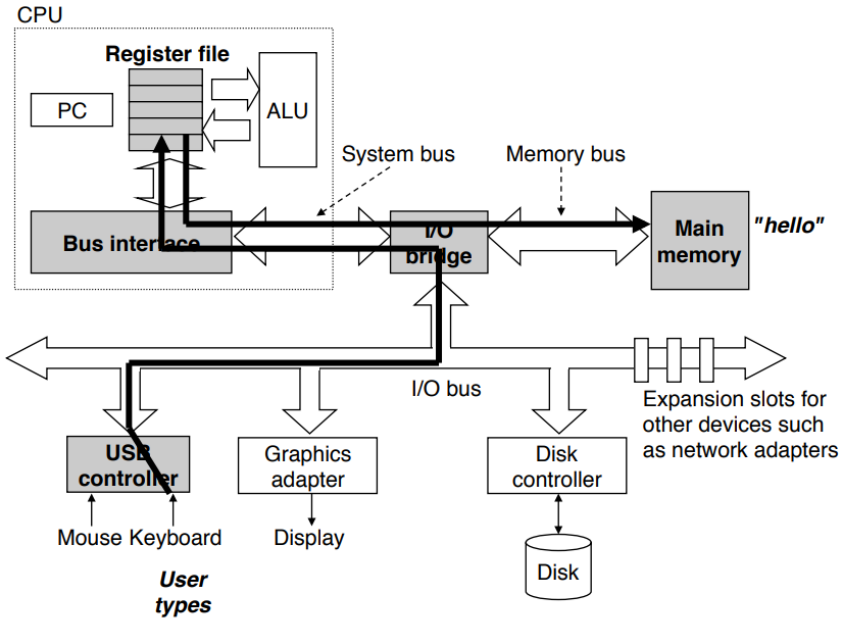
- 중앙처리장치(CPU) 또는 프로세서는 메인 메모리에 저장된 프로그램의 (기계어) 명령어(instruction)를 해독하고 실행
- 프로세서에는 워드 크기의 저장 장치인 레지스터(register) 파일과 현재 실행되는 명령어의 주소를 가리키는 프로그램 카운터(program counter, PC), 산술/논리 연산장치(Arithmetic/Logic Unit, ALU) 등이 있음

하드웨어 구성 – 프로세서

- 명령어의 요청에 의해 프로세서가 실행하는 작업
  - 적재 (Load)
    - 메인 메모리에서 한 워드 (또는 바이트)를 읽어 레지스터에 저장
  - 저장 (Store)
    - 레지스터에서 메인 메모리로 한 워드 (또는 바이트)를 저장
  - 연산 (Operate)
    - 레지스터에 저장된 데이터를 ALU에서 연산 처리하고 결과를 레지스터에 저장
  - 점프 (Jump)
    - 다음에 실행될 명령어 주소를 지정하기 위해 현재 실행되는 명령어에서 지정된 주소로 PC를 덮어쓰기 함
- 4장에서 프로세서의 구현(마이크로 구조, microarchitecture)을 소개

hello 실행 – 쉘 명령 입력

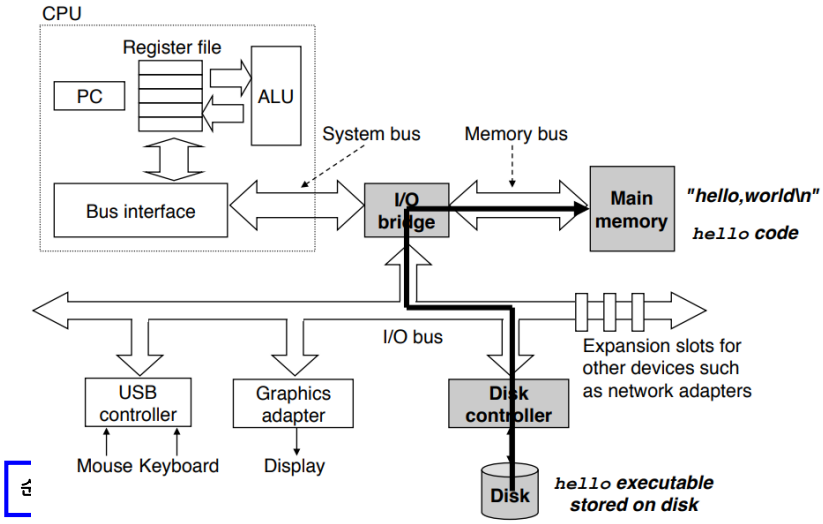
- 쉘 프로그램이 "hello" 명령을 읽어 각 문자를 레지스터를 경유하여 메모리에 저장



<그림 1.5>

hello 실행 – 프로그램 로딩

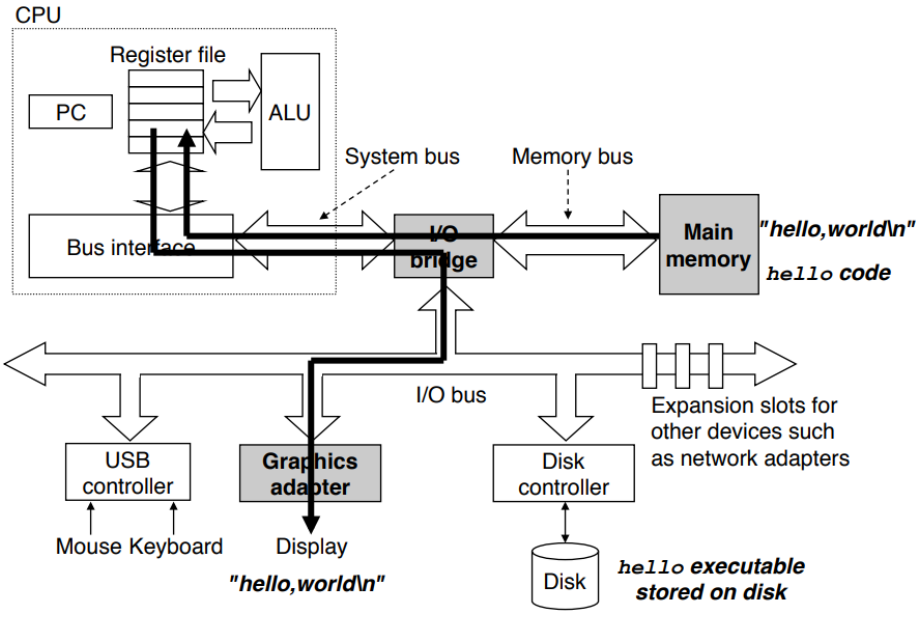
- 운영체제의 로더(loader)가 쉘에서 입력된 명령을 해석하여 hello 실행파일을 디스크에서 메인 메모리로 로딩
  - 프로세서는 직접 메모리 접근(Direct Memory Access, DMA) 기법 – 6장에서 소개 – 으로 프로세서를 거치지 않고 디스크에서 메인 메모리로 직접 이동



<그림 1.6>

hello 실행 – hello 프로그램 수행

- hello 프로그램의 main 루틴의 기계어 명령어를 수행
  - "Hello, world₩n" 문자열을 메모리에서 레지스터로 복사한 후, 디스플레이 장치로 전송하여 화면에 글자로 표시



<그림 1.7>

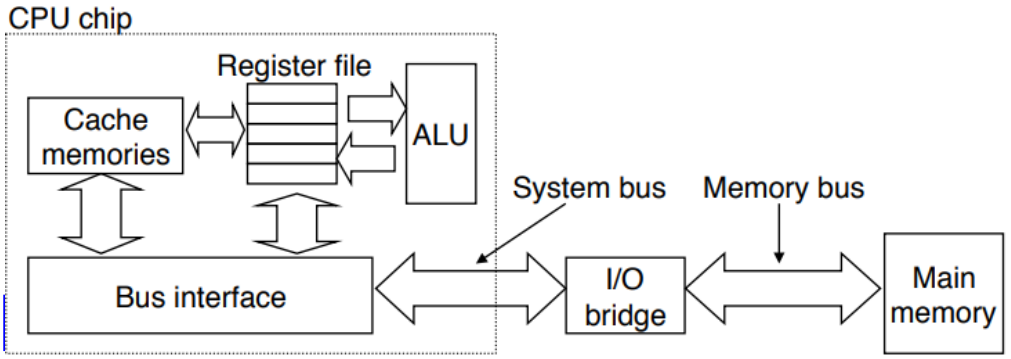
1.5 캐시가 중요하다
1.6 저장장치들은 계층 구조를 이룬다

컴퓨터 시스템에서의 정보 이동

- 앞의 예와 같이 컴퓨터 시스템은 정보의 이동에 많은 시간을 소비
  - 디스크의 hello 프로그램을 메인 메모리로 복사
  - 메인 메모리의 명령어들이 프로세서(레지스터)로 복사
  - “Hello, world₩n” 문자열 데이터도 디스크에서 메인 메모리, 프로세서를 경유하여 디스플레이 장치로 복사
- 컴퓨터 시스템에서 이들 이동(복사) 과정을 빠르게 처리하는 것이 중요
  - 저장 장치를 단일 메모리 구조가 아닌 속도와 가격에 따라서 여러 계층의 메모리 설계

캐시 메모리 (Cache Memory)

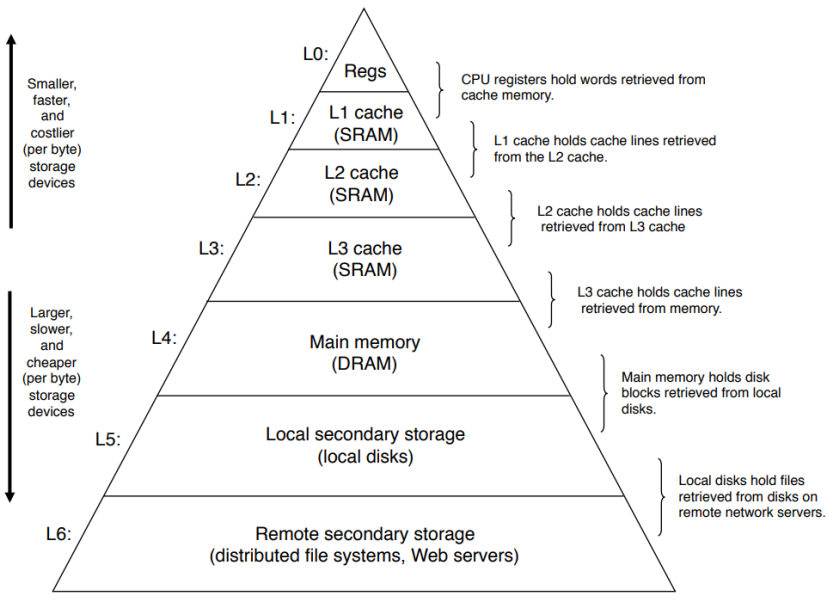
- 레지스터와 메모리
  - 프로세서의 레지스터는 작은 용량(수백 바이트)을 저장하지만 메인 메모리보다 100배 이상 빠름
  - 메인 메모리는 많은 용량(GB 급)을 저장하지만 느림
  - 프로세서와 메모리 간의 속도 격차는 지속적으로 증가
- 프로세서-메모리 간의 격차에 대응하기 위해 작고 빠른 캐시 메모리(간단히 캐시)를 고안하여 프로세서가 단기간에 사용할 가능성이 높은 정보를 저장



<그림 1.8>

메모리 계층 구조 (Memory Hierarchy)

- 프로그램이 특정 영역의 코드와 데이터를 참조하는 경향인 지역성(locality)을 활용하여 메모리 계층 구조를 설계
  - 자주 참조할 가능성이 높은 데이터나 코드를 작고 빠른 저장장치(캐시)에 저장하여 접근

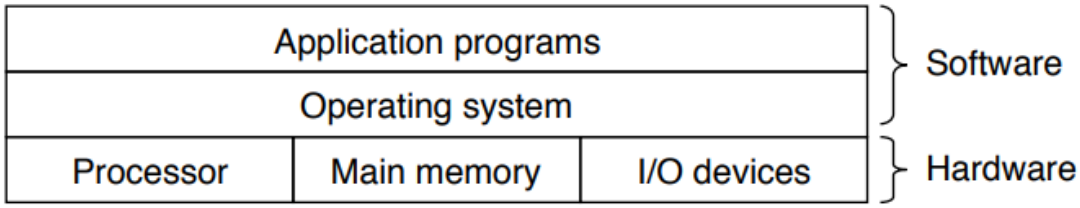


<그림 1.9>

1.7 운영체제는 하드웨어를 관리한다

운영체제 (Operating System)

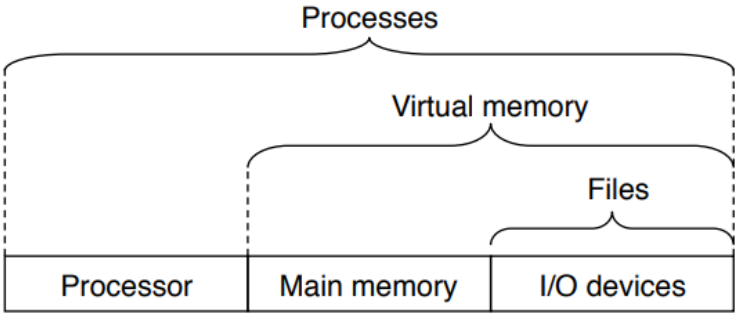
- 운영체제는 하드웨어와 소프트웨어 사이에 위치한 소프트웨어로 하드웨어를 제어하고 조작하는 서비스 제공
  - hello 프로그램이 키보드나 디스플레이, 디스크, 메인 메모리를 직접 접근하지 않고 운영체제가 제공하는 서비스를 활용



<그림 1.10>

운영체제의 추상화

- 운영체제는 추상화(abstraction)를 통해 서비스를 제공
  - 파일 (file)은 입출력 장치의 추상화
  - 가상메모리 (virtual memory)는 메인 메모리와 디스크 입출력 장치의 추상화
  - 프로세스 (process)는 프로세서, 메인 메모리, 입출력 장치 모두의 추상화



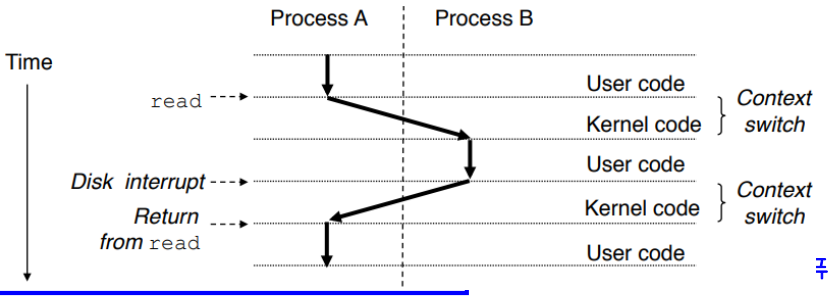
<그림 1.11>

프로세스 (Process)

- 프로세스는 실행 중인 프로그램에 대한 운영체제의 추상화
  - 한 개의 CPU에 다 수의 프로세스들이 동시에 실행(병행실행, concurrent execution)하는 것처럼 보임
    - 각 프로세스는 각 하드웨어를 배타적으로 사용하는 것처럼 느껴짐
- 실제로 운영체제는 문맥전환(context switch) 기법으로 여러 프로세스들을 교차 실행
  - 프로세스가 실행하는데 필요한 모든 상태정보를 문맥(또는 컨텍스트)라고 함
    - PC, 레지스터 파일, 메모리의 현재 값들
  - 현재 프로세스에서 다른 프로세스로 제어를 옮길 때 현재 프로세스의 컨텍스트를 저장하고,새로운 프로세스의 컨텍스트를 복원하는 것이 문맥 전환

프로세스 – hello 프로세스 시나리오 예

- 두 개의 동시성 프로세스 (concurrent process) 존재
  - 명령의 입력을 기다리는 쉘 프로세스 (프로세스 A)
  - hello 프로세스 (프로세스 B)
  - 문맥전환 동작
    - 쉘이 hello 프로그램 실행 명령을 받으면 시스템 콜(system call)이라는 특수 함수를 사용해서 제어권을 운영체제에 넘김
    - 운영체제는 쉘 컨텍스트를 저장하고, 새로운 hello 프로세스 컨텍스트를 생성하고 제어권을 hello 프로세스로 넘겨줌
    - hello 운영체제는 프로세스가 종료되면 쉘 프로세스의 컨텍스트를 복구하고 제어권을 쉘에게 넘겨 다음 명령의 입력을 기다림



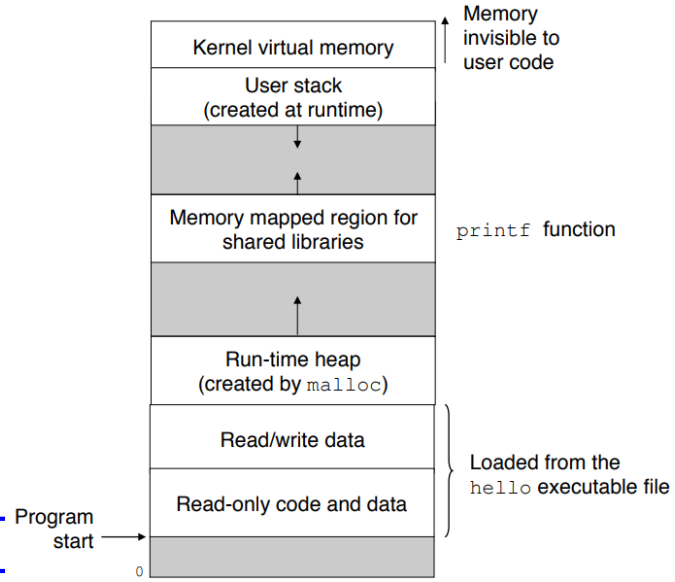
<그림 1.12>

쓰레드 (Thread)

- 프로세스는 쓰레드라는 다 수의 실행 유닛으로 구성
  - 각각의 쓰레드는 해당 프로세스의 컨텍스트에서 실행되며 동일한 코드와 전역 데이터(global data)를 공유
- 쓰레드는 프로그래밍 모델로써 중요성이 점점 커지고 있음
  - 다 수의 프로세스들 간 보다 쓰레드들 간이 데이터의 공유가 쉬움
  - 멀티프로세서(multi-processor) 구조에서 다중 쓰레딩(multi-threading) 프로그래밍은 여러 개의 프로세서를 활용한 병렬처리로 프로그램의 실행 속도를 높임

가상메모리 (Virtual Memory)

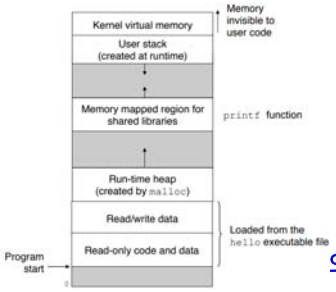
- 가상메모리는 각 프로세스들이 메인 메모리 전체를 독점적으로 사용하는 것과 같은 환상을 제공하는 추상화
  - 각 프로세스는 가상주소 공간이라는 균일한 메모리의 형태를 가짐
  - 리눅스 프로세스의 가상주소공간



<그림 1.13>

가상주소 공간

- 프로세스의 가상주소공간 영역 구분
  - 프로그램 코드와 데이터
    - 실행가능 목적파일로 초기화되는 코드와 데이터(전역변수) 영역
  - 힙 (heap)
    - 실행 시(런타임)에 동적으로 할당되는 고정된 크기의 영역
  - 공유 라이브러리
    - 공유 라이브러리 (C 표준, 수학 등) 코드와 데이터를 저장하는 영역
  - 스택 (stack)
    - 함수 호출 및 리턴 시에 사용되는 지역변수, 리턴 값 등이 저장되는 영역
  - 커널 가상메모리
    - 운영체제의 커널이 저장되는 영역



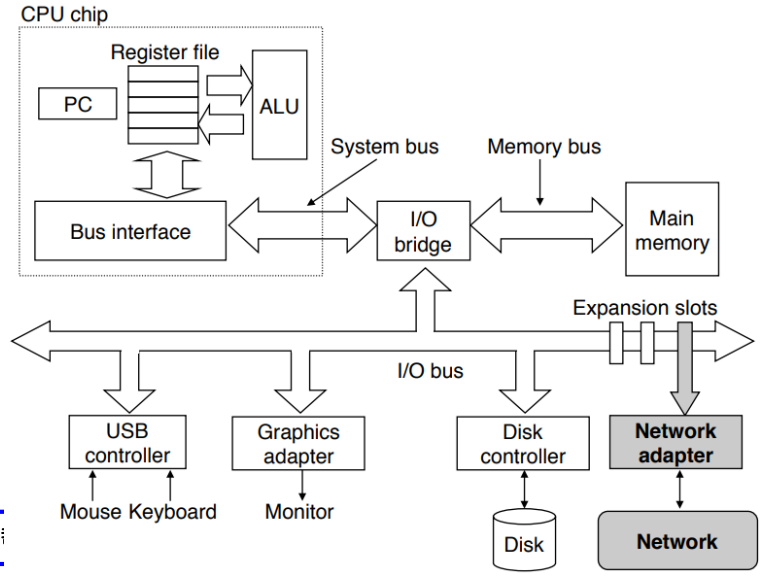
파일 (File)

- 파일은 연속된 바이트들로 구성
  - 디스크, 키보드, 디스플레이, 네트워크 등 모든 입출력 장치는 파일로 모델링
  - 시스템의 모든 입출력은 유닉스 I/O 시스템 콜을 사용하여 접근
  - 응용 프로그램에 시스템의 당야한 입출력 장치의 접근 시 통일된 관점을 제공

1.8 시스템은 네트워크를 사용하여 다른 시스템과 통신한다

네트워크 (Network)

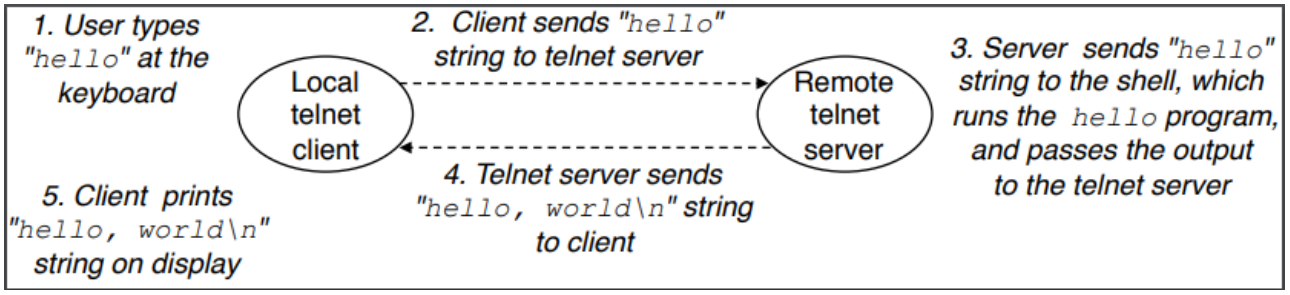
- 컴퓨터 시스템들은 네트워크를 통해 서로 연결
  - 시스템 입장에서는 네트워크는 또 다른 입출력 장치
  - 메인 메모리에서 네트워크 어댑터로 일련의 바이트들을 복사할 때 네트워크를 통해 다른 컴퓨터로 이동



<그림 1.14>

hello 프로그램 원격 실행 예

- telnet 네트워크 응용 프로그램을 사용하여 hello 프로그램을 원격 실행
  - telnet 클라이언트(사용자 로컬 컴퓨터)에서 명령 입력
  - telnet 서버 (원격 컴퓨터)의 hello 프로그램이 실행되어 출력 전송



<그림 1.15>

1.9 중요한 주제들

암달(Amdahl)의 법칙

- 암달의 법칙 (Amdal's law)
  - 컴퓨터의 어떤 개선책으로부터 얻을 수 있는 성능의 증가는 개선된 부분이 얼마나 많이 사용하느냐에 제한됨
  - 성능 개선 후 실행 시간
    = 개선에 영향을 받지 않는 시간 + 개선에 영향을 받는 시간/개선의 크기
  - 응용을 실행하는데 걸리는 시간 T_{old}, 개선하고자 하는 부분이 이 시간의 α 비율 만큼만 소모하고 이를 k배 개선하고자 할 때 개선된 시간 T_{new}

$$T_{new} = (1-\alpha)T_{old} + (\alpha T_{old})/k$$
$$= T_{old}[(1-\alpha) + \alpha/k]$$

  - 개선된 속도 $S = T_{old}/T_{new}$

$$S = 1 / [(1-\alpha) + \alpha/k]$$

암달의 법칙 예

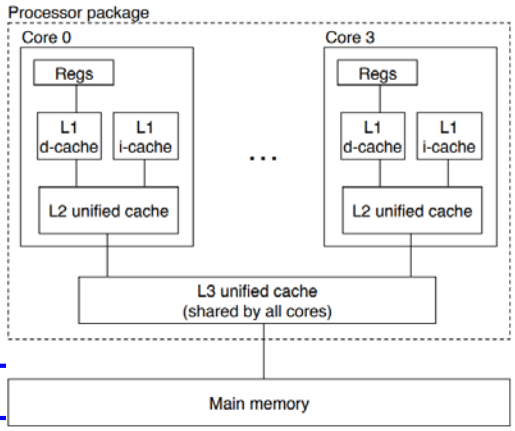
- 어떤 응용의 전체 수행 시간의 60%(α=0.6)를 소모하는 시스템의 일부분을 3배(k=3) 빠르게 속도를 개선하였다면 전체 속도 향상은?
  - S = 1 / [(1- α) + α/k] = 1 / [(0.4 + 0.6/3] = 1.67
  - 총 속도 향상은 1.67배로 3보다 적음
  - 전체 시스템의 속도 향상을 위해서는 개선에 영향을 많이 받는 부분의 개선이 필요

암달의 법칙 예 – 연습문제 1.2

- 어떤 회사의 소프트웨어를 4배 성능 개선을 하고자 한다. 이를 위해 소프트웨어의 90% 시간을 소모하는 시스템의 부분은 얼마나 개선해야 하는가?
  - 암달의 법칙에서 S = 4, α = 0.9, k = ?
  - 4 = 1 / [(1-0.9) + 0.9/k]
    0.4 + 3.6/k = 1.0
    3.6/k = 1.0-0.4 = 0.6
    k = 3.6/0.6 = 6
  - 6배 빠르게 개선해야 함

멀티프로세서 (Multiprocessor) 시스템

- 멀티프로세서 시스템은 여러 개의 프로세서들이 하나의 운영체제 커널 제어 하에서 동작하는 시스템
  - 멀티코어(multicore) 프로세서들과 하이퍼쓰레딩(hyperthreading) 기법 적용
- 멀티코어 프로세서
  - 여러 개의 CPU(코어)를 하나의 집적화된 칩에 내장
  - 인텔 i7 프로세서 코어 구성



<그림 1.17>

하이퍼쓰레딩 (Hyperthreading)

- 하이퍼쓰레딩 – 일반적으로 멀티쓰레딩(multithreading) – 은 한 CPU가 여러 개의 제어 흐름을 실행
  - 한 CPU에 여러개의 프로그램 카운터, 레지스터 파일들로 구성
  - 쓰레드들 간의 전환이 빠름
    - 매 사이클 마다 실행할 쓰레드를 결정
    - 한 쓰레드가 데이터를 캐시에 로드하기 위해 대기하면 CPU는 다른 쓰레드를 실행
  - 인텔 코어 i7 프로세서는 각 코어 당 두 개의 쓰레드를 실행
    - 4 코어 시스템에서는 8개의 쓰레드를 병렬로 실행

멀티프로세서 성능 개선

- 멀티프로세서를 이용하는 성능 개선에는 두 가지 방법이 있음

- 멀티프로그래밍 병렬성 이용
  - 동시성(concurrency)으로 병행 실행(concurrent execution)되는 다 수의 프로세스들이 여러 프로세서에서 실질적으로 동시에 병렬 실행(parallel execution)되어 성능 개선
  - 한 명의 사용자의 다수의 태스크를 동시에 수행
- 쓰레드 수준 병렬성 이용
  - 한 개의 응용 프로그램을 병렬로 효율적으로 실행할 수 있는 다 수의 쓰레드 (멀티쓰레드) 형태로 표현하여 각 쓰레드들이 병렬 실행하여 성능 개선

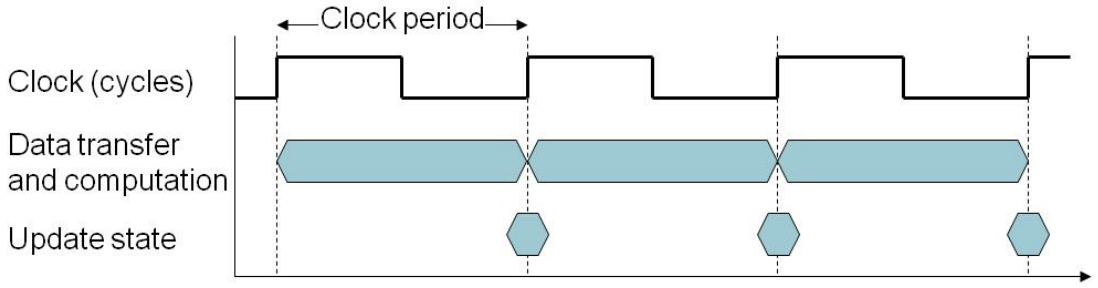
명령어 수준 병렬성 (Instruction Level Parallelism)

- 최근의 프로세서들은 여러 개의 명령어를 한 번에(한 클럭 사이클)에 실행하고, 이러한 특성을 명령어 수준 병렬성이라고 함
  - 매 클럭 마다 2개 이상의 명령어를 실행
  - 파이프라이닝 기법(4장에서 소개)을 적용하여 한 번에 100개의 명령까지 처리
    - 파이프라이닝은 한 명령어 처리를 여러 단계로 나누어 수행
    - 각 단계는 병렬로 처리

- 클럭 사이클 당 한 개 이상의 명령어를 실행할 수 있는 프로세서를 슈퍼스칼라 프로세서 (superscalar processor)라고 함

CPU 클럭

- 모든 컴퓨터는 하드웨어의 이벤트가 발생하는 시점(기간)을 정의하는 일정한 주기의 클럭을 가짐



  - 클럭 사이클 시간 (clock period, T), T = 1/f
    - 한 클럭 사이클 기간
    - 예, 250ps = 0.25ns = 250 x 10^{-12}s
  - 클럭 속도 (주파수) (clock frequency, rate, f), f = 1/T
    - 초 당 사이클 수
    - 예, 4.0GHz = 4000MHz = 4.0 x 10^{9}Hz

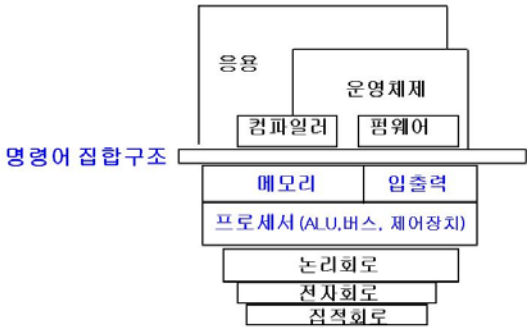
SIMD (Single Instruction Multiple Data) 병렬성

- SIMD (단일 명령어, 다중 데이터) 병렬성
  - 하나의 명령어가 병렬로 여러 개의 데이터(벡터형 데이터)의 연산을 수행
- 최신 인텔과 AMD의 프로세서들은 8개의 단일 정밀도 부동소수(C 언어의 float 데이터)들 간의 덧셈 연산을 병렬로 처리하는 SIMD 명령어 제공
  - SSE (Streaming SIMD Extension) 명령어
- SIMD 명령어들은 멀티미디어(영상, 소리, 동영상 등) 데이터를 처리하는 응용 프로그램의 속도 개선
  - 높은 데이터 수준 병렬성(data-level parallelism)을 갖는 응용

컴퓨터 시스템에서의 추상화 (1)

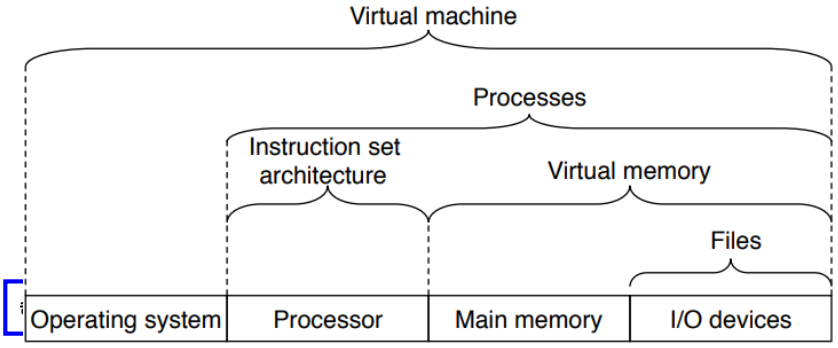
- 추상화(abstraction)는 컴퓨터 과학에서 중요한 개념
  - 프로그래밍 언어들은 프로그래머가 내부의 동작을 고려하지 않으면서 코드를 사용할 수 있도록 여러가지 형태의 추상화 지원
    - 자바의 클래스 선언, C의 함수 프로토타입 선언 등

- 컴퓨터 시스템에서의 추상화
  - 복잡한 시스템의 설계를 쉽게 하기 위해 컴퓨터 시스템의 하위계층의 상세한 내용을 보이지 않도록 구성한 모델



컴퓨터 시스템에서의 추상화 (2)

- 프로세서 관점에서의 추상화
  - 명령어 집합 구조는 실제 프로세서 하드웨어의 추상화를 제공
  - 하나의 명령어가 프로세서 내부에서 실행되는 모든 복잡한 과정을 추상화
- 운영체제 관점에서의 추상화
  - 파일은 입출력 장치, 가상메모리는 프로그램 메모리, 프로세스는 실행 중인 프로그램을 각각 추상화
- 가상머신은 운영체제, 프로세서, 프로그램 모두를 포함하는 컴퓨터 전체의 추상화



<그림 1.18>

1.10 요약

요약 (1)

- 컴퓨터 시스템은 응용 프로그램의 실행을 위해 동작하는 하드웨어와 시스템 소프트웨어로 구성
  - 컴퓨터 내의 정보는 해석되는 내용(context)에 따라 구분되는 비트들의 그룹으로 표시
  - 프로그램은 컴파일러와 링커에 의해 이진(binary) 실행파일로 번역
- 프로세서는 메인 메모리에 저장된 이진 명령어를 읽고 해석하여 실행
  - 컴퓨터는 대부분의 시간을 메모리, 입출력장치, CPU 레지스터 간의 데이터 복사에 소비
  - 컴퓨터 시스템의 저장장치들은 계층구조를 형성하여 CPU 레지스터가 최상위에, 하드웨어 캐시 메모리, DRAM 메인 메모리, 디스크 저장장치 등이 순차적으로 위치

요약 (2)

- 계층구조 상부의 저장장치들은 하부의 장치들을 위한 캐시 역할을 수행
- 운영체제 커널은 응용 프로그램과 하드웨어 사이에서 추상화를 통해 응용 프로그램에 서비스를 제공
  - 파일은 입출력 장치의 추상화
  - 가상메모리는 메인 메모리와 디스크의 추상화
  - 프로세스는 프로세서, 메인 메모리, 입출력장치의 추상화
- 네트워크는 컴퓨터 시스템이 서로 통신할 수 있는 방법 제공
  - 컴퓨터 시스템의 관점에서 네트워크는 또 하나의 입출력장치로 간주

과 제

과제 1-1: 암달의 법칙 적용

- 어떤 컴퓨터에서 100초 수행하는 프로그램이 있고, 그 중 80초는 곱하기 계산에 소요된다. 이 프로그램을 5배 빠르게 하기 위한 곱셈 속도의 개선 크기는?
  - 힌트: 다음 식 사용
    - 성능 개선 후 실행 시간
      = 개선에 영향을 받지 않는 시간 + 개선에 영향을 받는 시간/개선의 크기
- 과제 제출 시 유의 사항
  - 과제는 PPT로 작성하여 학습 플랫폼 과제에 업로드
  - 업로드 파일 이름: 학번-이름-과제이름.pptx
  - 프로그램 소스 시작 코드에 아래와 같이 학과, 학년, 학번, 이름 작성
    ### 컴퓨터공학과 1학년
    ### 20191234 홍길동
  - 타 학생과 복사본 발견 시 양 측 모두 감점
  - 과제 내용 및 발표 등을 고려하여 평가
    - 제출 기한이 지나면 학습 플랫폼 업로드 안됨

과제 1-2: VirtualBox 실습 플랫폼 구축 (실습과제)

- 윈도우 PC에서 상에서 리눅스 운영체제의 실습 플랫폼 구축을 위해 VirtualBox 가상머신을 구축
- VirtualBox 가상머신을 생성하고, 우분투 리눅스 배포판 설치
  - 실습 강의노트 및 인터넷 참조
  - 우분투는 64비트 버전으로 설치해야 함
    - 이미 설치된 우분투 사용도 무방 (학습 플랫폼 강의.ova)
      - VirtualBox 가상머신 -> 파일 -> 가상 시스템 가져오기
      - 비번: lecture / lecture
- 과제 작성 방법
  - PPT에 과정을 캡쳐하고 설명 (이 후 과제 동일)
  - 가상머신 생성 시 부여되는 가상머신 이름은 "학번-학생이름" 으로 지정
  - 리눅스 프롬트는 "학생이름> " 으로 지정
    - $ export PS1 = "학생이름> "

과제 1-3: C 프로그램 컴파일 (실습과제)

- 가상머신에서 의미있는 임의의 아래와 같이 C 프로그램을 작성, 컴파일, 실행
  - gcc 컴파일러 설치
  - 에디터를 사용하여 소스 C 프로그램 작성
    - C 프로그램은 C 언어와 자료구조 등에서 배운 의미 있는 프로그램
    - 코드의 내용이 다른 사람과 중복되면 감점
  - 소스 프로그램을 컴파일하고 실행
  - 소스 프로그램을 어셈블리어 프로그램으로 변환하여 확인
  - 목적파일을 생성하고 objdump 명령으로 역 어셈블(disassemble)하여 확인

- 리눅스 사용법은 참고자료 강의노트와 인터넷 참조