시스템 프로그래밍 1분반 1번째 과제

32203689 이호영

1.1

프로그램은 프로그래머가 에디터를 이용하여 만든 소스 프로그램으로부터 시작됩니다. 프로그램을 hello.c라는 텍스트 파일로 저장하면 소스 프로그램은 바이트라고 불리는 0과 1로 구성된 8비트 덩어리들의 연속으로 저장됩니다. 저장된 바이트들은 각각 프로그램의 텍스트 문자들을 나타냅니다. 대부분의 현대 시스템은 고유한 바이트 크기의 정수 값으로 각 문자를 나타내는 "ACSII" 표준을 사용하여 텍스트 문자를 나타냅니다.

이진법	팔진법	십진법	십육진법	모양	85진법 (아스키 85)	이진법	팔진법	십진법	십육진법	모양	85진법 (아스키 85)	이진법	팔진법	십진법	십육진법	모양	85진법 (아스키85)
0100000	040	32	20	␣		1000000	100	64	40	@	31	1100000	140	96	60	`	63
0100001	041	33	21	!	0	1000001	101	65	41	A	32	1100001	141	97	61	a	64
0100010	042	34	22	"	1	1000010	102	66	42	B	33	1100010	142	98	62	b	65
0100011	043	35	23	#	2	1000011	103	67	43	C	34	1100011	143	99	63	c	66
0100100	044	36	24	$	3	1000100	104	68	44	D	35	1100100	144	100	64	d	67
0100101	045	37	25	%	4	1000101	105	69	45	E	36	1100101	145	101	65	e	68
0100110	046	38	26	&	5	1000110	106	70	46	F	37	1100110	146	102	66	f	69
0100111	047	39	27	'	6	1000111	107	71	47	G	38	1100111	147	103	67	g	70
0101000	050	40	28	(	7	1001000	110	72	48	H	39	1101000	150	104	68	h	71
0101001	051	41	29	)	8	1001001	111	73	49	I	40	1101001	151	105	69	i	72
0101010	052	42	2A	*	9	1001010	112	74	4A	J	41	1101010	152	106	6A	j	73
0101011	053	43	2B	+	10	1001011	113	75	4B	K	42	1101011	153	107	6B	k	74
0101100	054	44	2C	,	11	1001100	114	76	4C	L	43	1101100	154	108	6C	l	75
0101101	055	45	2D	-	12	1001101	115	77	4D	M	44	1101101	155	109	6D	m	76
0101110	056	46	2E	.	13	1001110	116	78	4E	N	45	1101110	156	110	6E	n	77
0101111	057	47	2F	/	14	1001111	117	79	4F	O	46	1101111	157	111	6F	o	78
0110000	060	48	30	0	15	1010000	120	80	50	P	47	1110000	160	112	70	p	79
0110001	061	49	31	1	16	1010001	121	81	51	Q	48	1110001	161	113	71	q	80
0110010	062	50	32	2	17	1010010	122	82	52	R	49	1110010	162	114	72	r	81
0110011	063	51	33	3	18	1010011	123	83	53	S	50	1110011	163	115	73	s	82
0110100	064	52	34	4	19	1010100	124	84	54	T	51	1110100	164	116	74	t	83
0110101	065	53	35	5	20	1010101	125	85	55	U	52	1110101	165	117	75	u	84
0110110	066	54	36	6	21	1010110	126	86	56	V	53	1110110	166	118	76	v	
0110111	067	55	37	7	22	1010111	127	87	57	W	54	1110111	167	119	77	w	
0111000	070	56	38	8	23	1011000	130	88	58	X	55	1111000	170	120	78	x	
0111001	071	57	39	9	24	1011001	131	89	59	Y	56	1111001	171	121	79	y	
0111010	072	58	3A	:	25	1011010	132	90	5A	Z	57	1111010	172	122	7A	z	
0111011	073	59	3B	;	26	1011011	133	91	5B	[	58	1111011	173	123	7B	{	
0111100	074	60	3C	<	27	1011100	134	92	5C	\	59	1111100	174	124	7C	|	
0111101	075	61	3D	=	28	1011101	135	93	5D	]	60	1111101	175	125	7D	}	
0111110	076	62	3E	>	29	1011110	136	94	5E	^	61	1111110	176	126	7E	~	
0111111	077	63	3F	?	30	1011111	137	95	5F	_	62						

프로그램은 바이트들의 연속으로 저장되며 각각의 바이트는 문자에 대응되는 정수 값을 가집니다. 예를 들어 "a"는 정수 값으로 97를 갖으며 "z"는 정수 값으로 122를 갖습니다. 문자로만 구성된 파일을 텍스트 파일이라고 하며 텍스트 파일의 줄은 정수 값 10으로 표시되는 보이지 않는 줄 바꿈 문자 "₩n"으로 끝납니다. 다른 모든 파일은 이진 파일이라고 합니다. 텍스트 파일을 통해 디스크 파일, 메모리에 저장된 프로그램, 메모리에 저장된 사용자 데이터 및 네트워크를 통해 전송되는 데이터를 포함하여 시스템의 모든 정보는 비트 묶음으로 표시된다는 것을 알 수 있습니다. 유일한 데이터 개체 구별법은 프로그램들의 다른 본문 내용입니다.

1.2

다른 본문 내용에서 동일한 연속된 바이트들은 정수, 부동 소수점 숫자, 문자 문자열 또는 컴퓨터 명령을 나타낼 수 있습니다. 프로그래머로서, 우리는 숫자의 기계적인 표현을 이해할 필요가 있습니다. 왜냐하면 그것들은 정수나 실수와 같지 않고 예상치 못한 방식으로 작동할 수 있는 유한 근사치이기 때문입니다.

시스템에서 프로그램을 실행하기 위해서는 프로그램의 개별 텍스트 문장이 다른 프로그램에 의해

낮은 수준의 기계 언어 명령으로 변환되어야 합니다. 그런 다음 이러한 명령어는 실행 가능한 개체 프로그램이라는 형식으로 패키지되어 이진 디스크 파일로 저장됩니다. 개체 프로그램은 실행 가능한 개체 파일이라고도 합니다.

유닉스 시스템에서 소스 파일에서 오브젝트 파일로 변환은 컴파일러 드라이버에 의해 수행됩니다.

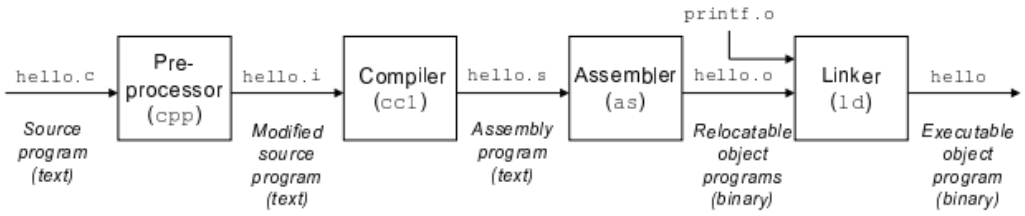


Figure 1.3: **The compilation system.**

예를 들어, $ gcc -o hello hello.c에서 gcc 컴파일러 드라이버는 소스 파일 hello.c를 읽고 실행 가능한 객체 파일 hello로 변환합니다. 변환은 4단계를 거쳐서 수행되는데, 4단계(전처리 프로세서, 컴파일러, 어셈블러, 링커)를 수행하는 프로그램들은 집합적으로 컴파일 시스템이라고 알려져 있습니다.

- 전처리 단계에서는 #문자로 시작하는 지시에 따라 원래 C 프로그램을 수정합니다. 예를 들어 #include <stdio.h>명령어는 전처리 프로세서에게 시스템 헤더 파일 stdio.h의 내용을 읽고 프로그램 텍스트에 직접 삽입하라고 지시한다. 지시한 결과는 ".i"가 붙어서 hello.i로 수정된다.

- 컴파일 단계에서는 텍스트 파일 hello.i를 어셈블러 언어 프로그램을 포함하는 텍스트 파일 hello.s로 변환합니다. 어셈블러 언어 프로그램의 각 문장은 표준 텍스트 형식으로 하나의 낮은 수준의 기계어 명령을 정확하게 설명합니다. 어셈블러 언어는 서로 다른 고급 언어에 대해 서로 다른 컴파일러에 공통 출력 언어를 제공하기 때문에 유용합니다.

- 어셈블러 단계에서는 hello.s를 기계어 명령어로 변환하여 재배치 가능한 오브젝트 프로그램으로 알려진 형태로 패키징하고 결과를 오브젝트 파일 hello.o에 저장합니다. hello.o 파일은 바이트가 문자가 아닌, 기계어 명령을 인코딩한 이진 파일입니다.

- 링킹 단계에서는 hello 프로그램이 모든 C컴파일러가 제공하는 표준 C 라이브러리의 일부인 printf 함수를 호출합니다. printf 함수는 printf.o라는 별도의 사전 컴파일된 오브젝트 파일에 상주하며, 이 파일은 hello.o 프로그램과 병합됩니다. 여기서 링커는 이러한 병합을 처리하며 그 후에는 메모리에 로드되고 시스템에 의해 실행될 준비가 된 실행 가능한 객체 파일 결과물인 hello 파일이 됩니다.

1.3

간단한 프로그램을 위해 위해, 우리는 컴파일 시스템에 의존하여 정확하고 효율적인 기계 코드를 생산할 수 있다. 그러나 프로그래머가 컴파일 시스템의 작동 방식을 이해해야 하는 중요한 이유가 있습니다.

프로그램 성능 최적화

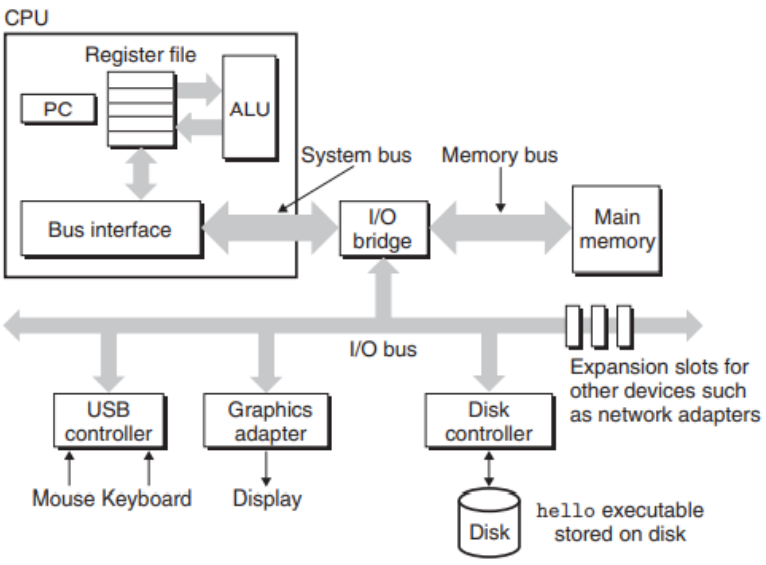
현대의 컴파일러는 보통 좋은 코드를 생성하는 정교한 도구이다. 프로그래머로서, 효율적인 코드를 작성하기 위해 컴파일러의 내부 작업까지는 알 필요가 없다. 그러나 C 프로그램에서 좋은 코딩을 결정하기 위해서는 기계 수준의 코드와 컴파일러가 다른 C 문장을 기계 코드로 변환하는 방법에 대한 기본적인 이해가 필요하다.

1.4

Hello.c라는 소스 프로그램은 컴파일 시스템에 의해 hello라는 실행가능한 목적파일로 번역되어 디스크에 저장되었다. 실행 파일을 유닉스 시스템에서 실행하기 위해서 셸이라는 응용프로그램에 이름을 입력한다.

```
linux> ./hello
hello, world
linux>
```

셸은 커맨드라인 인터프리터로 프롬프트를 출력하고 명령어 라인을 입력받아 그 명령을 실행한다. 만일 명령어 라인이 내장 셸 명령어가 아니면 셸은 실행파일의 이름으로 판단하고 그 파일을 로딩해서 실행한다. 그래서 이러한 경우에 셸은 hello 프로그램을 로딩하고 실행한 뒤에 종료를 기다린다. Hello 프로그램은 메시지를 화면에 출력하고 종료하며 셸은 프롬프트를 출력해 주고 다음 입력 명령어 라인을 기다린다.



Bus

시스템을 관통하는 전기전 배선군을 버스라고 하며, 컴포넌트들 간에 바이트 정보들을 전송한다. 버스는 일반적으로 워드라고 하는 고정 크기의 바이트 단위로 데이터를 전송하도록 설계되며, 한 개의 워드를 구성하는 바이트 수는 시스템마다 보유하는 기본 시스템 변수다. 오늘날 대부분의 컴퓨터들은 4바이트(32비트) 또는 8바이트(64비트) 워드 크기를 갖는다.

입출력 장치

입출력 장치는 시스템과 외부세계와의 연결을 담당합니다. 각 입출력 장치는 입출력 버스와 컨트롤러나 어댑터를 통해 연결됩니다. 이 두 장치의 차이는 패키징에 있는데, 컨트롤러는 디바이스 자체가 칩셋이거나 시스템의 마더보드에 장착됩니다. 어댑터는 마더보드의 슬롯에 장착되는 카드입니다. 이들 각각의 목적은 입출력 버스와 입출력 장치들 간에 정보를 주고받도록 해주는 일입니다.

메인 메모리

메인 메모리는 프로세서가 프로그램을 실행하는 동안 데이터와 프로그램을 모두 저장하는 임시 저장장치입니다. 물리적으로 메인 메모리는 DRAM 칩들로 구성되어 있으며 논리적으로 메모리는 연속적인 바이트들의 배열로, 각각 0부터 시작해서 고유의 주소를 가집니다. 일반적으로 한 개의 프로그램을 구성하는 각 기계어 인스트럭션은 다양한 바이트 크기를 갖습니다.

프로세서

CPU 또는 프로세서는 메인 메모리에 저장된 인스트럭션들을 실행하는 엔진입니다. 프로세서의 중심에는 워드 크기의 저장장치(레지스터)인 프로그램 카운터(PC)가 있습니다. 시스템에 전원이 공급되는 순간부터 전원이 끊길 때까지 프로세서는 PC가 가리키는 곳의 인스트럭션을 반복적으로 실행하고 PC값이 다음 인스트럭션의 위치를 가리키도록 업데이트 합니다. 인스트럭션들은 규칙적인 순서로 실행되고, 한 개의 인스트럭션을 실행하는 것은 여러 단계를 수행함으로써 이루어집니다. 프로세서는 PC가 가리키는 메모리로부터 인스트럭션을 읽어오고, 인스트럭션에서 비트들을 해석하여 인스트럭션이 지정하는 간단한 동작을 실행하고, PC를 다음 인스트럭션 위치로 업데이트 합니다. 새로운 위치는 방금 수행한 인스트럭션과 메모리 상에서 연속적일 수 도 있고, 그렇지 않을 수도 있습니다.

이러한 동작들은 메인 메모리, 레지스터 파일, ALU 주위를 순환합니다. 레지스터 파일은 각각 고유의 이름을 갖는 워드 크기의 레지스터 집합으로 구성되며 ALU는 새 데이터와 주소 값을 계산합니다.

CPU가 실행하는 작업들

- Load: 메인 메모리에서 레지스터에 한 바이트 또는 워드를 이전 값에 덮어쓰는 방식으로 복사하는 것
- Store: 레지스터에서 메인 메모리로 한 바이트 또는 워드를 이전 값을 덮어쓰는 방식으로 복사하는 것
- Operate: 두 레지스터의 값을 ALU로 복사하고 두 개의 워드로 수식연산을 수행한 뒤, 결과를 덮어쓰기 방식으로 레지스터에 저장하는 것
- Jump: 인스트럭션 자신으로부터 한 개의 워드를 추출하고, 이것을 PC에 덮어쓰기 방식으로 복사하는 것

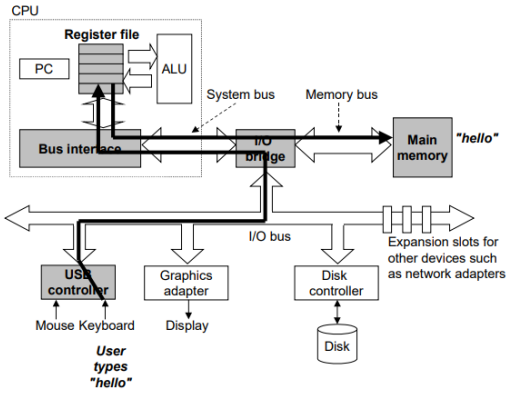


Figure 1.5: **Reading the `hello` command from the keyboard.**

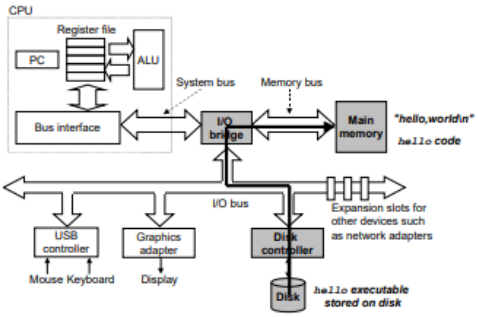


Figure 1.6: **Loading the executable from disk into main memory.**

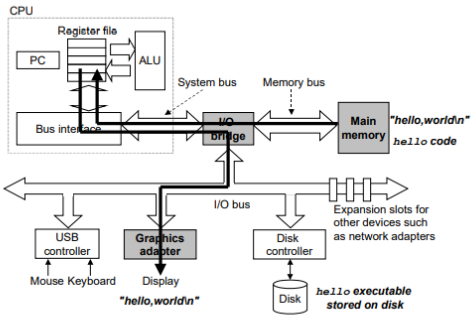


Figure 1.7: **Writing the output string from memory to the display.**

처음에 셸 프로그램은 자신의 인스트럭션을 실행하면서 사용자가 명령을 입력하기를 기다립니다. ":/hello"를 입력하면 셸 프로그램은 각각의 문자를 레지스터에 읽어들인 후 메모리에 저장합니다. (그림 1.5 참조) 키보드의 Enter 키를 누르면 셸이 명령 입력이 끝났음을 알 수 있습니다. 그런 다음 셸은 디스크로부터 메인 메모리로 hello 객체 파일의 코드와 데이터를 복사하는 일련의 명령을 실행하여 실행 파일 hello 파일을 로드합니다. 이 데이터에는 최종적으로 출력될 문자열이 포함됩니다. DMA(Direct memory access)라고 알려진 기술을 사용하여, 데이터는 프로세서를 통과하지 않고 디스크에서 메인 메모리로 직접 이동합니다. (그림 1.6 참조)

Hello 객체 파일의 코드와 데이터가 메모리에 로드되면 프로세서는 hello 프로그램의 주 루틴에서 기계어 명령을 실행하기 시작합니다. 이 명령어는 "hello, world₩n" 문자열의 바이트를 메모리에서

레지스터 파일로 복사하고 거기서 디스플레이 장치로 복사합니다. (그림 1.7 참조)

1.5

시스템이 정보를 한 장소에서 다른 장소로 이동시키는 데 많은 시간이 걸립니다. Hello 프로그램의 기계 명령은 원래 디스크에 저장되며, 프로그램이 로드되면 기본 메모리에 복사됩니다. 프로세서가 프로그램을 실행하면 기본 메모리에서 프로세서로 명령이 복사됩니다. 마찬가지로 원래 디스크에 있던 데이터 문자열인 "hello, world₩n"는 메인 메모리에 복사된 다음, 메인 메모리에서 디스플레이 장치로 복사됩니다. 프로그래머의 관점에서, 이 복사물의 대부분은 프로그램의 "실제 작업"을 느리게 하는 오버헤드입니다.

따라서 시스템 설계자의 주요 목표는 이러한 복사 작업을 가능한 빨리 실행하는 것입니다. 물리적 법칙 때문에 대형 스토리지 장치는 소형 스토리지 장치보다 속도가 느립니다. 그리고 속도가 빠른 장치는 속도가 느린 장치보다 제작 비용이 더 많이 듭니다. 예를 들어, 일반적인 시스템의 디스크 드라이브는 기본 메모리보다 1000배 크지만 프로세서는 10,000,000배 더 오래 걸릴 수 있습니다.

일반적인 레지스터 파일은 수백 바이트의 정보를 저장하는 반면, 메인 메모리의 경우는 십억 개의 바이트를 저장한다. 그러나 프로세서는 레지스터 파일의 데이터를 읽는 데 메모리의 경우보다 거의 100배 더 빨리 읽을 수 있다. 메인 메모리를 더 빠르게 동작하도록 만드는 것보다 프로세서를 더 빨리 동작하도록 만드는 것이 더 쉽고 비용이 적게 든다.

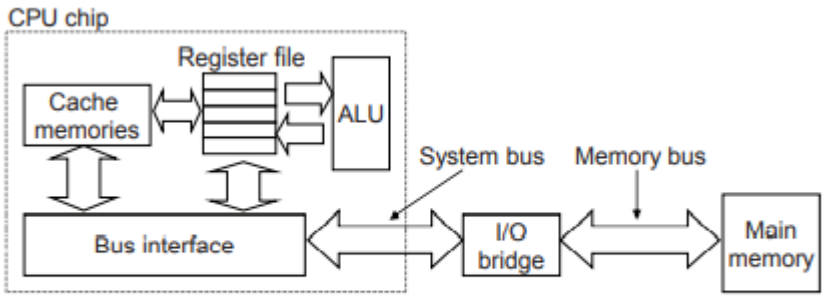


Figure 1.8: **Cache memories.**

그림 1.8은 일반적인 시스템에서의 캐시 메모리를 보여준다. 프로세서 칩 내에 들어 있는 L1 캐시는 대략 수천 바이트의 데이터를 저장할 수 있으며, 거의 레지스터 파일만큼 빠른 속도로 액세스할 수 있다. 이보다 더 큰 L2 캐시는 수백 킬로바이트에서 수 메가 바이트의 용량을 가지며 프로세서와 전용 버스를 통해 연결된다. 캐시 시스템의 이면에 깔려 있는 아이디어는 프로그램이 지엽적인 영역의 코드와 데이터를 액세스하는 경향인 지역성을 활용하여 시스템이 매우 크고 빠른 메모리 효과를 얻을 수 있다는 것이다. 자주 액세스할 가능성이 높은 데이터를 캐시가 보관하도록 설정하면 빠른 캐시를 이용해서 대부분의 메모리 작업을 수행할 수 있게 된다.

1.6

모든 컴퓨터 시스템의 저장장치들은 그림 1.9와 같은 메모리 계층 구조로 구성되어 있다. 계층의 꼭대기에서부터 맨 밑바닥까지 이동할수록 저장장치들은 더 느리고, 더 크고, 바이트당 가격이 싸진다. 레지스터 파일은 계층 구조의 최상위인 L0을 차지하며 메인 메모리는 L4에 위치한다. 한 레벨의 저장장치가 다음 하위레벨 저장장치의 캐시 역할을 한다는 것이다. L1과 L2의 캐시는 각각 L2와 L3의 캐시이며 L3 캐시는 메인 메모리의 캐시이고, 이 캐시는 디스크의 캐시 역할을 한다. 일부 분산 파일시스템을 가지는 네트워크 시스템에서 로컬 디스크는 다른 시스템의 디스크에 저장된 데이터의 캐시 역할을 수행한다. 로컬 디스크들은 원격 네트워크 서버에서 파일들을 가져와 보관한다. 프로그래머들이 성능을 개선하기 위해서 다른 종류의 캐시들을 활용할 수 있듯이 프로그래머는 전체 메모리 계층 구조에 대한 지식을 활용할 수 있다.

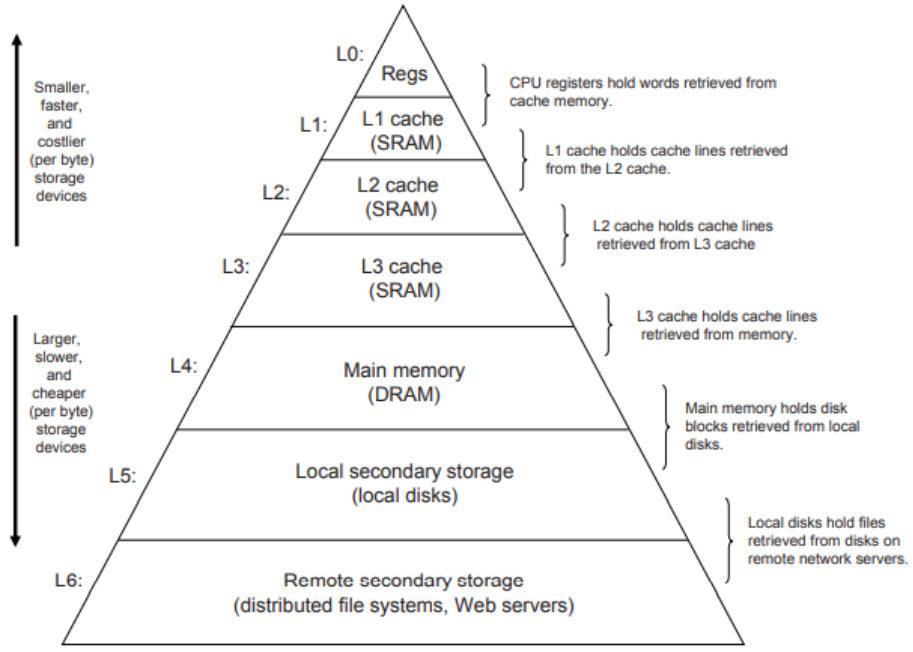


Figure 1.9: **An example of a memory hierarchy.**

1.7

셸 프로그램이 hello 프로그램을 로드하고 실행했을 때와 hello 프로그램이 메시지를 출력할 때 운영체제가 제공하는 서비스를 활용한다. 운영체제는 그림 1.10처럼 하드웨어와 소프트웨어 사이에 위치한 소프트웨어 계층으로 생각할 수 있다. 응용프로그램이 하드웨어를 제어하려면 언제나 운영체제를 통해서 해야 한다.

운영체제는 두 가지 주요 목적을 갖고 있다.

1. 제멋대로 동작하는 응용프로그램들이 하드웨어를 잘못 사용하는 것을 막기 위해
2. 응용프로그램들이 단순하고 균일한 메커니즘을 사용하여 복잡하고 매우 다른 저수준 하

드웨어 장치들을 조작할 수 있도록 하기 위해

운영체제는 위 두 가지 목표를 그림 1.11에서와 같이 근본적인 추상화를 통해 달성한다. 파일은 입출력장치의 추상화이고, 가상 메모리는 메인 메모리와 디스크 입출력 장치의 추상화, 프로세스는 프로세서, 메인 메모리, 입출력장치 모두의 추상화 결과이다.

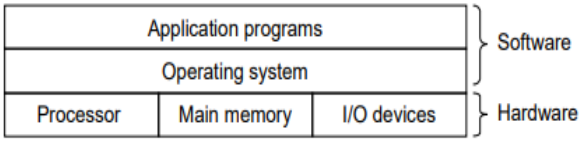


Figure 1.10: **Layered view of a computer system.**

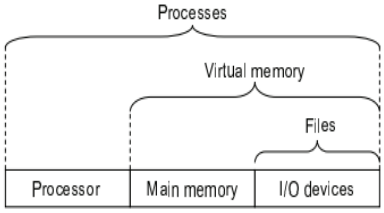


Figure 1.11: **Abstractions provided by an operating system.**

프로세스

프로세스는 실행 중인 프로그램에 대한 운영체제의 추상화다. 다수의 프로세스들은 동일한 시스템에서 동시에 실행될 수 있으며, 한 프로세스의 인스트럭션들이 다른 프로세스의 인스트럭션들과 섞인다는 것을 의미한다. 요즘의 멀티코어 프로세스들은 여러 개의 프로그램을 동시에 실행할 수 있다. 운영체제는 문맥 전환이라는 방법을 사용해서 이러한 교차실행을 수행하며, 프로세스가 실행하는 데 필요한 모든 상태정보의 변화를 추적한다. 이 상태정보는 PC, 레지스터 파일, 메인 메모리의 현재 값을 포함하고 있다. 운영체제는 현재 프로세스에서 다른 새로운 프로세스로 제어를 옮기려고 할 때 현재 프로세스의 컨텍스트를 저장하고 새 프로세스의 컨텍스트를 복원시키는 문맥전환을 실행하여 제어권을 새 프로세스로 넘겨준다.

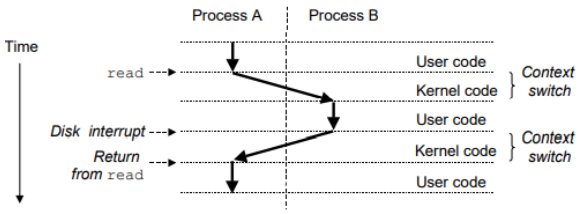


Figure 1.12: **Process context switching.**

그림 1.12에는 두 개의 동시성 프로세스인 쉘 프로세스와 hello 프로세스가 존재한다. 처음에는 셸 프로세스가 혼자서 작동하고 있다가 명령줄에서 입력을 기다린다. Hello 프로그램을 실행하려는 명령을 받으면, 셸은 시스템 콜이라는 특수 함수를 호출하여 운영체제로 제어권을 넘겨준다. 운영체제는 셸의 컨텍스트를 저장하고 새로운 hello 프로세스와 컨텍스트를 생성한 뒤 제어권을 새 hello 프로세스로 넘겨준다. Hello가 종료되면 운영체제는 셸 프로세스의 컨텍스트를 복구시키고 제어권을 넘겨주면서 다음 명령 줄 입력을 기다린다.

하나의 프로세스에서 다른 프로세스로의 전환은 운영체제 커널에 의해 관리된다. 커널은 운영체제 코드의 일부분으로 메모리에 상주한다. 응용프로그램이 운영체제에 의한 어떤 작업을 요청하면, 컴퓨터는 파일 읽기나 쓰기와 같은 특정 시스템 콜을 실행해서 커널에 제어를 넘겨준다. 그러면 커널은 요청된 작업을 수행하고 응용프로그램으로 리턴한다. 커널은 별도의 프로세스가 아니라는 점에 유의해야 한다. 커널은 모든 프로세스를 관리하기 위해 시스템이 이용하는 코드와 자료 구조의 집합이다.

쓰레드

프로세스는 쓰레드라고 하는 다수의 실행 유닛으로 구성되어 있다. 각각의 쓰레드는 해당 프로세스의 컨텍스트에서 실행되며 동일한 코드와 전역 데이터를 공유한다. 쓰레드는 다수의 프로세스들에서보다 데이터의 공유가 더 쉽고 프로세스보다 더 효율적이다.

가상 메모리

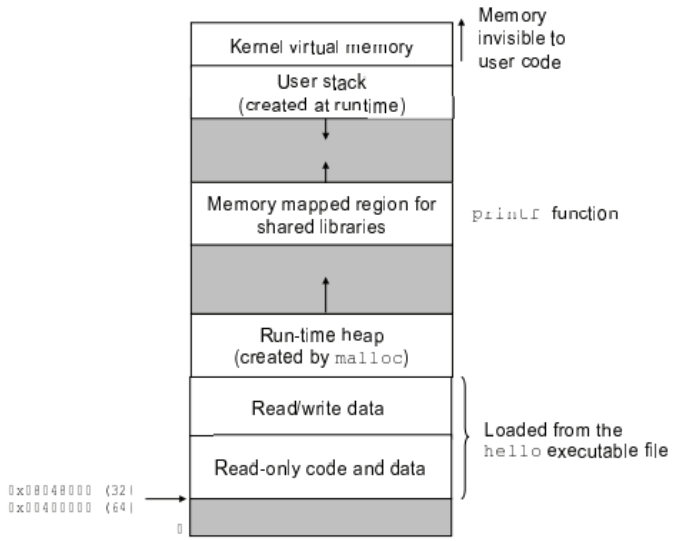


Figure 1.13: **Process virtual address space.**

가상 메모리는 각 프로세스가 메인 메모리를 독점적으로 사용하는 착각을 제공하는 추상화이다. 각 프로세스는 가상 주소 공간으로 알려진 동일한 메모리 뷰를 가진다. 리눅스에서 주소 공간의 맨 위 영역은 모든 프로세스에 공통적인 운영 체제의 코드와 데이터를 위해 예약된다. 주소 공간의 하단 영역에는 사용자의 프로세스에 의해 정의된 코드와 데이터가 있으며 그림의 주소는 아래에서 위로 증가한다.

각 프로세스에서 볼 수 있는 가상 주소 공간은 각각 특정 목적을 가진 여러 가지 잘 정의된 영역으로 구성된다. 프로세스의 가상 메모리의 내용을 디스크에 저장한 다음 메인 메모리를 디스크의 캐시로 사용한다.

- 프로그램 코드 및 데이터: 코드는 모든 프로세스에 대해 동일한 고정 주소에서 시작되며, 그 다음 전역 변수에 해당하는 데이터 위치에서 시작됩니다.
- Heap: 코드 및 데이터 영역 바로 뒤에 런타임 힙이 나타납니다. 프로세스가 실행되면 크기가 고정되는 코드와 데이터 영역과는 달리 힙은 malloc 및 free와 같은 C 표준 라이브러리 루틴에 대한 호출의 결과로 런타임에 동적으로 확장 및 수축한다.
- 공유 라이브러리: 주소 공간의 중간 근처에는 C 표준 라이브러리 및 수학 라이브러리와 같은 공유 라이브러리의 코드와 데이터를 보관하는 영역이 있다.
- Stack: 사용자의 가상 주소 공간 맨 위에는 컴파일러가 함수 호출을 구현하기 위해 사용하는 사용자 스택이 있다. 힙과 마찬가지로 프로그램 실행 중에 사용자 스택이 동적으로 확장 및 축소되며 함수를 호출할 때마다 스택이 증가하고 함수가 return 될 때마다 수축된다.

- 커널 가상 메모리: 커널은 항상 메모리에 상주하는 운영체제의 일부이다. 주소 공간의 위쪽 영역은 커널을 위한 공간이다. 응용 프로그램은 이 영역의 내용을 읽거나 쓰거나 커널 코드에 정의된 함수를 직접 호출할 수 없습니다.

파일

파일은 바이트의 연속이다. 디스크, 키보드, 디스플레이 및 네트워크를 포함한 모든 I/O 장치는 파일로 모델링 되며 시스템의 모든 입력과 출력은 유닉스 I/O라고 알려진 시스템 호출의 작은 집합을 사용하여 파일을 읽고 쓰면서 수행된다.

배우고 싶은 목표

제가 시스템 프로그래밍을 통해 배우고 싶은 부분은 프로그래밍에서 마주할 수 있는 오류 코드들에 당황하지 않고 차분하게 오류를 해결할 수 있는 능력을 기르는 것입니다. 현재 아마존 AWS에서 직접 서버를 구입하여 저만의 웹 페이지 작업을 하고 있는데, 처음으로 익숙하지 않은 우분투 환경에서 작업하다 보니 생소한 부분들을 많이 접했습니다. 셸부터 시작해서 다양한 리눅스 명령어들과 vi, 웹서버, 다른 패키지 파일들을 실행하거나 설치할 때 발생하는 코드들에 대해 처음엔 낯설고 어려웠지만 점점 우분투 환경에서 다양한 프로젝트들을 진행하면서 시스템 명령에 익숙해지고 이러한 문제가 왜 발생하는 가에 대한 고민도 많이 하게 됐습니다.

하지만 아직도 저에게 부족한 부분이 많다고 느끼고 있으며 이러한 부족한 부분들을 이번 수업을 통해 채워나가려 합니다! 이 글을 쓰는 시점에서 강의를 현재 3주차까지 들었는데 강의 중간중간 제가 마주했던 우분투 명령어들을 보게 되니 반갑기도 하고 이 명령어가 이러한 역할을 하고 있었구나를 다시 한 번 느낄 수 있었습니다! 앞으로 수업을 열심히 듣고, 우분투 환경에서 마주할 수 있는 여러 코드들을 이해하고 정밀하게 분석해 볼 생각입니다. 지금까지 열심히 강의 들어온 것처럼 앞으로도 수업 열심히 참여해서 좋은 성적도 받고 제가 얻고자 하는 부분도 얻어가겠습니다! 좋은 수업 진행해주셔서 감사합니다! 앞으로도 수업 열심히 듣겠습니다😊