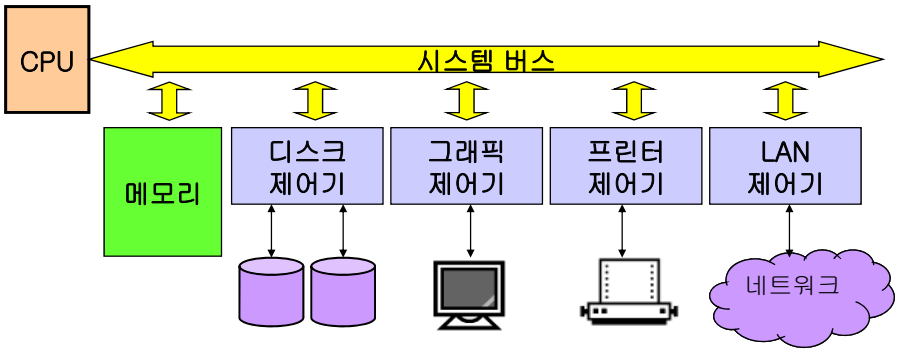
컴퓨터 구조와 어셈블리 언어

- ✓컴퓨터 시스템의 하드웨어 구성
- ✓프로세서의 프로그래밍 모델
- ✓C 언어, 어셈블리 언어, 기계어
- ✓어셈블리 언어의 기초
- ✓C 언어와 어셈블리 언어의 결합
- ✓커널 모드와 유저 모드
- ✓모드전환과 시스템 콜 함수
- ✓인터럽트 발생과 커널의 실행

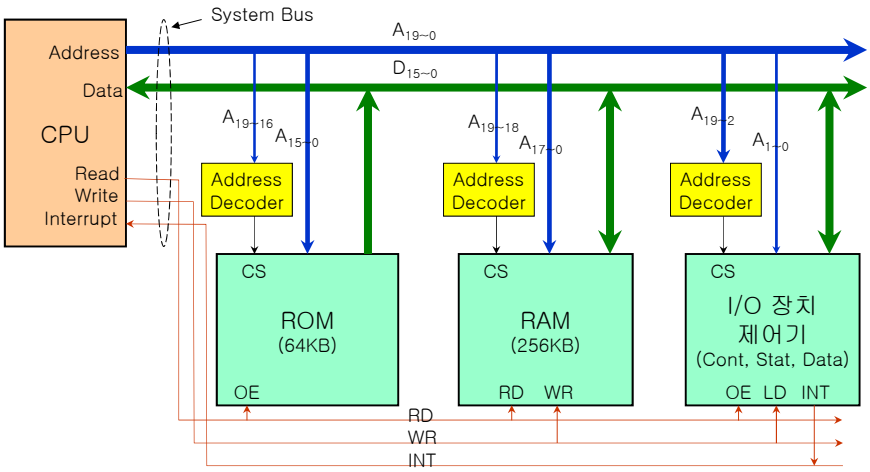
이해하고 넘어가야 할 내용들

- ✓ 컴퓨터 주변장치의 제어 및 인터럽트 처리
- ✓ 프로세서의 레지스터 집합과 프로그래밍 모델
- ✓ 컴파일 과정에서 C 언어, 어셈블리 언어 및 기계어의 연관성
- ✓ 어셈블리 언어 문법 및 C 언어 프로그램 부분과의 통합 방법
- ✓ 커널 모드와 유저 모드의 필요성 및 모드 전환 방법
- ✓ 시스템 콜 함수에서 인수 전달 및 모드 전환 방법
- ✓ 인터럽트 처리과정에서 프로세스와 커널 간의 실행위치의 이동

컴퓨터 시스템의 하드웨어 구성



메모리 및 I/O 장치 주소



주소 디코딩

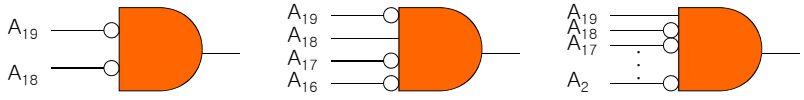
- ✓ 주소 맵 (address map)
  - ✓ 모듈별로 할당된 주소 영역들의 집합
- ✓ 주소 맵의 예
  - ✓ RAM: 0x00000 ~ 0x3FFFF (256K Byte)
  - ✓ ROM: 0x40000 ~ 0x4FFFF (64K Byte)
  - ✓ I/O : 0x80000 ~ 0x80003 (4 byte)
- ✓ 주소 디코더 (address decoder)
  - ✓ 지정된 영역부분의 주소일 때만 출력이 활성화되도록 조합논리회로로 구성

주소 디코더 회로

- ✓ **대상별 선택주소**

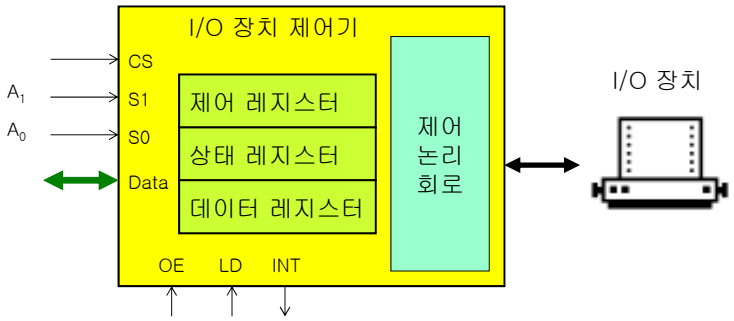
대상	주소 범위 예 (2진수)	선택 주소(2진수)
RAM	0000 0000 0000 0000 0000 ~ 0011 1111 1111 1111 1111	00xx xxxx xxxx xxxx xxxx
ROM	0100 0000 0000 0000 0000 ~ 0100 1111 1111 1111 1111	0100 xxxx xxxx xxxx xxxx
I/O장치	1000 0000 0000 0000 0000 ~ 1000 0000 0000 0000 0011	1000 0000 0000 0000 00xx

- ✓ **주소 디코더**



RAM 주소 디코더 ROM 주소 디코더 I/O 장치 주소 디코더

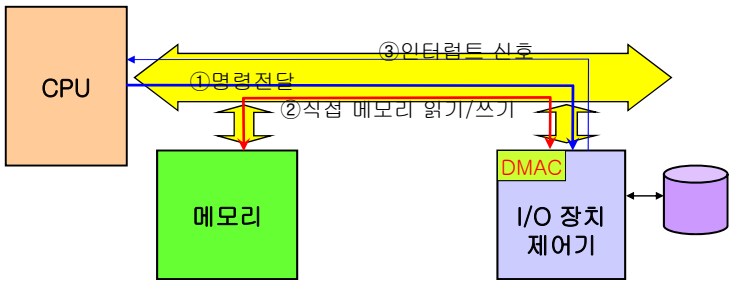
입출력 장치 제어



- ✓ 명령전달: 제어레지스터에 쓰기 작업으로 명령 전달
- ✓ 작업완료 확인
  - ✓ Polling 방식: 주기적으로 상태레지스터를 읽어서 검사
  - ✓ Interrupt 방식: 작업완료시점에 장치제어기가 CPU로 인터럽트 발생
- ✓ 데이터 입출력: 데이터 레지스터에 읽기나 쓰기 작업으로 처리

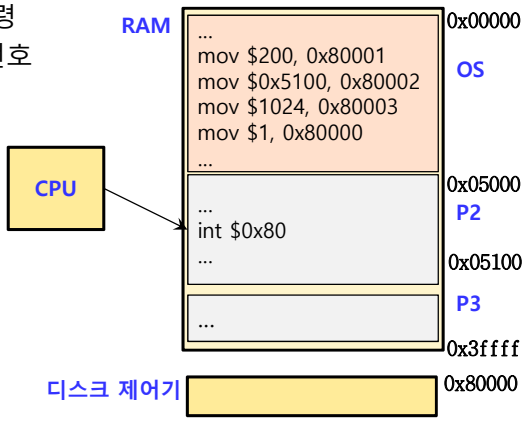
Direct Memory Access

- ✓ DMA (Direct Memory Access)
  - ✓ I/O 장치로부터 직접 메모리에 읽기/쓰기 처리
  - ✓ DMAC: 메모리에 읽기/쓰기를 처리하는 제어기
  - ✓ 입출력 작업시간 동안 CPU는 다른 작업 처리 가능



디스크 제어기 명령 전달 예시

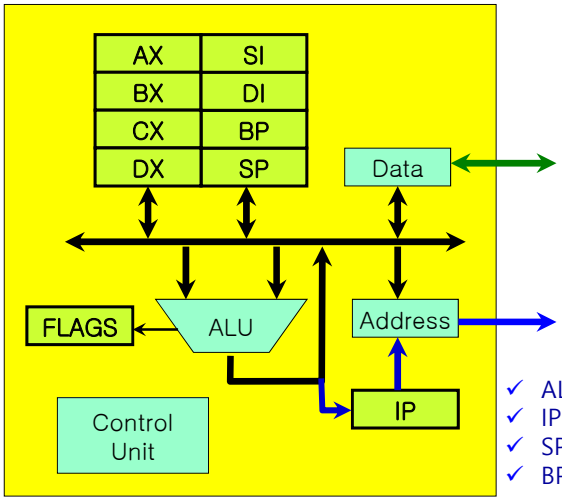
- ✓ 프로세스 P2
  - ✓ 디스크의 200번 블록에서 1024바이트를 메모리 0x5100번지로 읽기
- ✓ 가정: 디스크 제어기의 레지스터 주소
  - ✓ 0x80000: 읽기/쓰기 명령
  - ✓ 0x80001: 디스크 블록번호
  - ✓ 0x80002: 메모리 주소
  - ✓ 0x80003: 크기
  - ✓ 명령: 1(읽기)/2(쓰기)



프로세서의 프로그래밍 모델

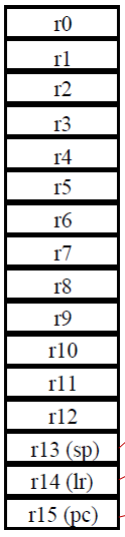
- ✓ 커널 개발 언어
  - ✓ 대부분은 C 언어로 작성하고 부분적으로 어셈블리 언어로 작성하여 하나로 결합
  - ✓ 어셈블리 언어 부분: 문맥교환, 하드웨어 제어, 인터럽트 처리 등
- ✓ 프로세서
  - ✓ CPU를 하나의 칩으로 제작한 것
  - ✓ 메모리 관리, 통신, 주변장치 등의 기능들이 포함되기도 함
  - ✓ 내부에 여러 가지의 레지스터들이 포함됨
- ✓ 프로그래밍
  - ✓ 레지스터들을 활용하여 어셈블리 언어로 프로그램 작성
  - ✓ 프로세서마다 레지스터 집합이 다르고, 어셈블리 언어의 문법도 상이함

8086 프로세서의 내부 구조



- ✓ ALU: Arithmetic Logic Unit
- ✓ IP: Instruction Pointer
- ✓ SP: Stack Pointer
- ✓ BP: Base Pointer

(참고) ARM의 레지스터 집합



Program Status Register

cpsr

범용 레지스터와
프로그램 카운터

명령어 별 대상 레지스터
R0~R14: all instructions
R15 (PC): most instructions
CPSR: specific instructions

Stack Pointer

Link Register

Program Counter

C 언어와 어셈블리 언어

✓C 언어

```
sum = 0;
for (index = 1; index <= 100; index++)
    sum = sum + index;      ①
```

✓① 부분의 8086 어셈블리 언어 표현

```
mov sum, %ax
add index, %ax
mov %ax, sum
```


8086 어셈블리 언어와 기계어

MOV memory to register 명령어
10001011 00rrr110
aaaaaaaa aaaaaaaa

MOV register to memory 명령어
10001001 00rrr110
aaaaaaaa aaaaaaaa

ADD memory to register 명령어
00000011 00rrr110
aaaaaaaa aaaaaaaa

✓**8086 어셈블리 언어**

```
mov sum, %ax
add index, %ax
mov %ax, sum
```

✓**8086 기계어 표현**

```
10001011 00000110  : MOV memory to register T⇐
00000001 00000000  : sumㅌ ㅣ□ (DS + 0x0100)
00000011 00000110  : ADD memory to register T⇐
00000001 00000010  : indexㅌ ㅣ□ (DS + 0x0102)
10001001 00000110  : MOV register to memory T⇐
00000001 00000000  : sumㅌ ㅣ□ (DS + 0x0100)
```

어셈블리 언어의 기초

- ✓ 문법
  - ✓ 어셈블러마다 약간 상이함 (표준이 없음)
  - ✓ 여기서는 GNU 어셈블러 (as) 적용
- ✓ 기본적인 표현 형식

종류	표시 예	의미
레이블	`label:`	표시된 위치의 주소
명령어	`addl %eax, %ebx`	프로세서가 실행할 내용
설명문	`# comments`	읽는 사람에게 정보 전달
지시어	`.globl label`	표시된 'label' 주소를 다른 파일에서도 참조 가능
지시어	`.extern label`	표시된 'label'은 다른 파일에서 정의된 주소임

어드레싱 모드

- ✓ 명령어 구성:
  - ✓ operation operand1, operand2(목적지)
  - ✓ Operand는 operation의 대상
- ✓ 어드레싱 모드 (addressing mode)
  - ✓ Operand를 표시하는 방법

종류	표시 예	의미
즉석값	`$20`	20의 값 자체
레지스터	`%eax`	EAX 레지스터의 내용
직접주소	`label`	label이 표시된 주소의 메모리 내용
	`0x1234`	0x1234 번지의 메모리 내용
간접주소	`(%esp)`	ESP 레지스터의 값이 지정하는 주소의 메모리 내용
	`8(%esp)`	ESP 레지스터의 값에 8을 더한 주소의 메모리 내용

대표적인 명령어들

종류	표시 예	의미
move	`movl $4, label` `movl 8(%esp),%ebx`	- 수치 4를 label 주소의 메모리에 기록 - ESP 레지스터의 값에 8을 더한 주소의 메모리 내용을 EBX 레지스터에 기록
add	`addl label, %eax`	- label 주소의 메모리 내용을 EAX 레지스터에 더해줌
subtract	`subl $4, %ecx`	- ECX 레지스터 값에서 수치 4를 빼줌
compare	`cmpl $0, %eax`	- EAX 레시스터 값과 0을 비교함
condition al jump	`jge label`	- 직전의 실행결과가 0보다 크거나 같으면 label 주소로 분기 (jump if greater than or equal)
negate	`negl %eax`	- EAX 레지스터 값을 부호를 반대로 변경
push	`pushl %ebx`	- EBX 레지스터 값을 스택에 푸시
pop	`popl %eax`	- 스택에서 꺼내어 EAX 레시스터에 저장
call	`call label`	- label 주소의 함수를 호출
return	`ret`	- 현재의 함수로부터 호출 위치로 복귀
interrupt	`int $80`	- 80번 인터럽트 발생
interrupt return	`iret`	- 인터럽트 처리루틴으로부터 발생 이전의 실행 위치로 복귀

주요 명령어 설명

- ✓ 산술처리 명령어
  - ✓ addl (add), subl (subtract), cmpl (compare), negl (negate) 등
  - ✓ FLAGS 레지스터: 실행과정에서 발생하는 Carry, Overflow, Sign, Zero 비트 등 저장
  - ✓ FLAGS 레지스터의 비트들은 조건부 분기 (jge 등) 명령의 조건에 사용
  - ✓ Compare는 subtract와 동일하나 결과를 저장하지 않음
- ✓ 스택처리 명령어
  - ✓ Push: SP를 4만큼 줄인다음 SP가 가리키는 곳에 내용 저장
  - ✓ Pop: SP가 가리키는 곳의 내용을 읽어내고 SP를 4만큼 증가
- ✓ 함수호출 및 복귀 명령어
  - ✓ Call: 바로 다음의 명령어 주소를 스택에 push, operand에 명시된 주소를 EIP에 저장 (operand 에 명시된 주소로 이동)
  - ✓ Return (ret): 스택에서 pop 동작으로 EIP 값 복구 (call 명령어 다로 다음위치로 복귀)

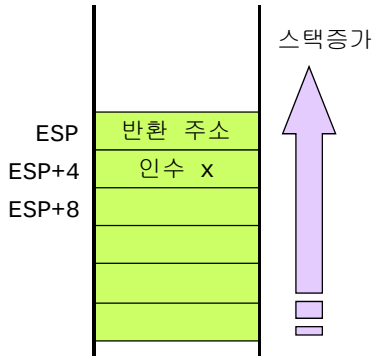
C 언어와 어셈블리 언어의 결합

- ✓ C 컴파일러 (gcc 기준)의 함수 호출
  - ✓ 인수는 스택에 push 하여 전달 (push 순서는 마지막 인수부터)
  - ✓ 반환값은 EAX 레지스터로 전달
- ✓ "value = abs(x);" 부분의 번역 예

```
…
x = …;
value = abs(x);
…
```

```
…
pushl x                 # 변수 x의 값을 스택에 쌓기
call abs                # 함수 abs의 위치로 이동하여 실행
movl %eax, value        # 함수 abs의 반환 값을 value에 복사
addl $4, %esp           # 스택 포인터를 원래 값이 되도록 복구
…
```


abs 함수 실행시의 스택



어셈블리언어 함수의 작성 예

- ✓ 절대값을 반환하는 함수 abs
  - ✓ 인수는 스택에 push 하여 전달
  - ✓ 반화값은 EAX 레지스터로 전달

```
.globl abs              # abs는 다른 파일에서 호출이 가능하게 선언
abs:                    # abs 함수의 시작 주소
   movl 4(%esp), %eax   # ESP+4의 주소의 내용을 EAX에 복사
   cmpl $0, %eax        # EAX 값에서 0을 빼는 방법으로 비교
   jge done             # 결과가 0보다 크거나 같으면 done 주소로 분기
   negl %eax            # EAX 값의 부호를 반대로 변경
done:
   ret                  # abs 함수를 호출한 위치로 복귀
```

picoKernel의 어셈블리 언어 함수 사용

```
contextSwitch(&prev->tContext, &next->tContext)
contextSet(thread->tContext, threadStartup)
eflags = interruptDisable()
interruptRestore(eflags)
```

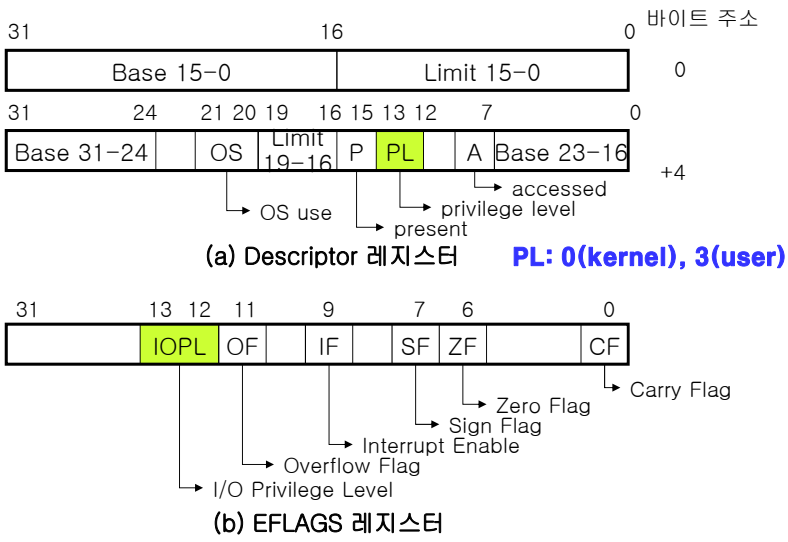

컴파일 및 링킹

- ✓ 소스 프로그램
  - ✓ C 언어 프로그램: main.c test.c
  - ✓ 어셈블리 언어 (as for x86) 프로그램 : abs.s
- ✓ 컴파일, 어셈블링, 링킹
  - ✓ gcc –c main.c
  - ✓ gcc –c test.c
  - ✓ as abs.s
  - ✓ ld –o test main.o test.o abs.o
- ✓ 컴파일2
  - ✓ gcc –o test main.c test.c abs.s

커널 모드와 유저 모드

- ✓ 모드의 구분
  - ✓ 프로세스가 실행되고 있을 때와 커널이 실행되고 있을 때에 따라 CPU의 실행 권한을 구분해야 함
  - ✓ 커널이 실행될 때는 모든 권한을 가지고 실행
  - ✓ 프로세스가 실행되고 있을 때에는 제한된 권한으로만 실행
- ✓ 커널 모드 (kernel mode)
  - ✓ 하드웨어 접근, 특수 레지스터의 접근 및 수정 (인터럽트 금지 포함)
  - ➔ 특권 명령 (privileged instruction) 실행 가능
  - ✓ 전체 메모리 영역 접근 가능 (MMU 기능과 연계)
  - ✓ supervisor mode, privileged mode, 또는 system mode 라고도 함
- ✓ 유저 모드 (user mode)
  - ✓ 프로세스 자신에게 할당된 메모리 영역만 접근 허용
  - ✓ 하드웨어 접근이나 특수 레지스터 접근을 제한함
- ✓ CPU의 모드 구별
  - ✓ 모드를 표시하는 특수 레지스터
  - ✓ 상태 레지스터, Descriptor 레지스터 등

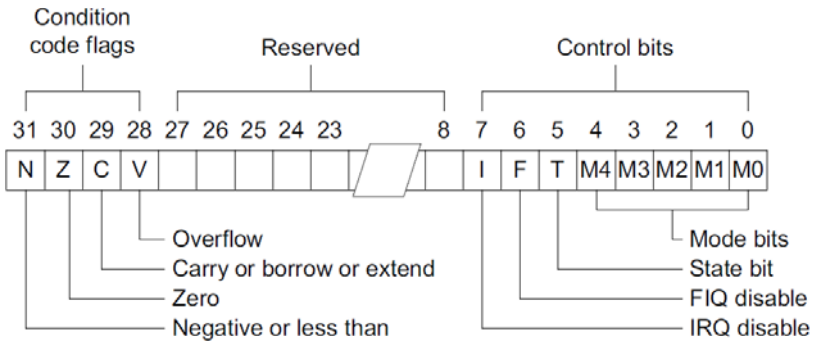
x86 프로세서의 모드 구별



(a) Descriptor 레지스터 **PL: 0(kernel), 3(user)**

(b) EFLAGS 레지스터

(참고) ARM의 CPSR 비트 구성



Mode bits
- 10000: User
- 10001: FIQ
- 10010: IRQ
- 10011: Supervisor (SWI)
- 11011: Undefined (undefined inst.)
- 11111: System (privileged mode)

IRQ: interrupt request
FIQ: fast interrupt request

모드 전환

- ✓ 커널 모드로의 전환
  - ✓ 프로세스가 실행중에 커널의 기능을 호출할때
    - ✓ 인터럽트를 발생시키는 명령어 (x86은 int 명령어) 실행으로 처리
  - ✓ 인터럽트가 발생하여 커널의 인터럽트 처리 루틴을 실행할 때
- ✓ 유저 모드로의 복귀
  - ✓ 인터럽트가 발생할 당시에 저장된 모드로 복귀 (x86은 iret 명령어)
  - ✓ 필요시 모드 표시 레지스터 (상태 레지스터) 를 강제로 변경하는 것도 가능
- ✓ 인터럽트가 발생하면 CPU는 (소프트웨어 인터럽트 포함)
  - ✓ 실행중이던 주소와 모드를 보관 (스택이나 별도 레지스터 사용)
    - ✓ 커널모드용 스택이 사용됨 (커널모드용 SP 따로 있음)
  - ✓ 상태를 커널모드로 변경
  - ✓ 인터럽트 원인별로 지정된 처리루틴(ISR)으로 이동하여 실행
- ✓ 인터럽트로부터 복귀 명령어(iret)를 실행하면 CPU는
  - ✓ 저장된 모드와 실행위치로 복귀하여 실행 계속

Linux의 write 함수

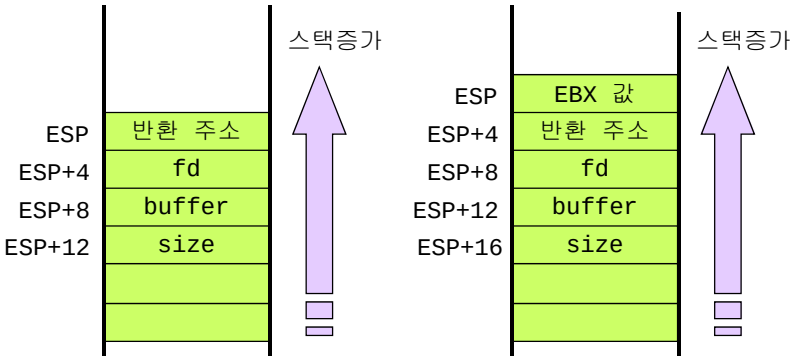
```
.globl write                # write 함수는 외부에서 호출 가능함
.extern errno               # errno 는 다른 파일에서 선언되어 있음

# 함수호출: write(int fd, char *buffer, int size)

write:                      # write 함수의 시작 주소
    pushl %ebx              # 스택에 EBX 저장
    movl 8(%esp), %ebx      # fd 값을 EBX로 읽어 들임
    movl 12(%esp), %ecx     # buffer 값을 ECX로
    movl 16(%esp), %edx     # size 값을 EDX로
    movl $4, %eax           # EAX에 write 지정번호 4
    int $0x80               # 시스템 콜을 위한 인터럽트 0x80번 발생
    cmpl $0, %eax           # EAX에 저장된 반환값 검사
    jge .L1                 # 음수(에러)가 아니면 .L1으로
    negl %eax               # 음수이면 양수로 부호 변경 (오류 번호)
    movl %eax, errno        # errno에 오류 번호 기록
    movl $-1, %eax          # 반환 값을 -1로 설정
.L1: popl %ebx              # 스택에 저장해 둔 EBX 값 복구
    ret                     # write를 호출한 위치로 복귀
```


Write 함수 실행시의 스택 사용

✓ "pushl %ebx" 실행직전과 실행후



여러 파일의 링크 과정

✓ test.c

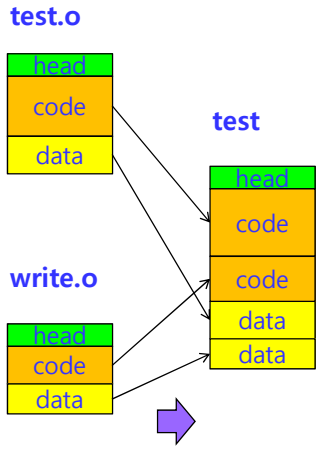
```
extern int write(int, char *, int);
int errno;
main() {
    ...
    write(fd, buf, 100);
    ...
}
```

✓ write.s

```
.globl write
.extern errno
write:
    ...
    ret
```

gcc –c test.c

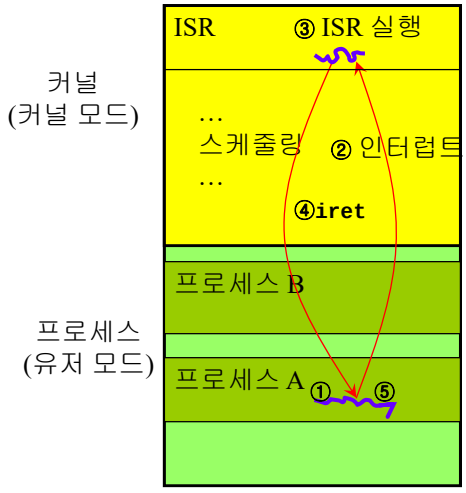
(as write.s)
gcc –c write.s



(ld –o test test.o write.o)
gcc –o test test.o write.o

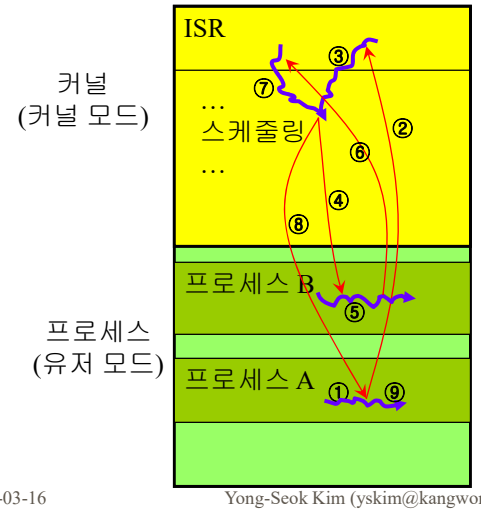
인터럽트 발생과 커널 실행

✓ 프로세스 A ➔ 커널의 ISR 실행 ➔ 프로세스 A 실행 재개



인터럽트와 문맥교환

프로세스 A ➔ (파일읽기) ISR ③ ➔ 스케줄링/문맥교환 ➔ 프로세스 B ➔ (파일읽기 완료) ISR ⑦ ➔ 스케줄링/문맥교환 ➔ 프로세스 A



A에서 B로의 문맥교환 처리방법 힌트
1. 스택 상단의 A의 복귀주소 백업
2. 스택 상단을 B의 복귀주소로 변경
3. iret 명령어 실행

[참고] 프로세스 별 정보 관리

- ✓ 커널 내에서 프로세스 별로 필요한 정보를 자료구조로 관리
- ✓ 프로세스 별로 필요한 정보
  - ✓ 사용중인 메모리 영역 (시작주소와 크기)
  - ✓ 선택되었을 때 복귀할 실행 주소 (Context_IP)
  - ✓ …
- ✓ 문맥교환 처리 (프로세스 A에서 B로) (3장에서 설명)
  1. A의 Context_IP에 현재 스택 상단의 복귀주소를 복사
  2. 현재 스택 상단의 복귀주소 부분에 B의 Context_IP를 기록
  3. iret 명령어 실행 (유저 모드로 복귀)
  - ✓ *(스택도 B의 것으로 변경해야 함)*

커널 내부의 인터럽트 처리

```
.globl int80Isr
.extern syscallRead, syscallWrite
int80Isr:               # "int $0x80" 명령어 인터럽트 처리루틴
    pushl %edx          # 3번째 인수
    pushl %ecx          # 2번째 인수
    pushl %ebx          # 1번째 인수
# check read system call; %eax: read 3, write 4, ...
    cmpl $3, %eax
    jne _write
    call syscallRead    # (프로그램 7.6)
    jmp _isrDone
# check write system call
_write: cmpl $4, %eax
    jne _open
    call syscallWrite # (프로그램 7.6)
    jmp _isrDone
_open:
#   ...
_isrDone:
    addl $12, %esp      # 3번의 pushl에 의한 ESP 감소분 복구
    iret                # int 0x80write을 호출한 위치로 복귀
```


x86 인터럽트 처리 절차

- ✓ 인터럽트 원인별로 지정된 부분을 실행후 원래 위치로 복귀해야 함
- ✓ 인터럽트 발생시 (INT 명령어나 인터럽트 신호 발생)
  - ✓ 복귀할 주소와 현재의 CPU 모드를 스택에 푸시
    - ✓ SP ← SP-4; Memory[SP] ← CPU 모드
    - ✓ SP ← SP-4; memory[SP] ← IP
  - ✓ CPU 모드는 커널모드로 변경
  - ✓ IP에는 인터럽트 원인 별로 지정된 주소 값을 기록
    - ✓ IP ← Memory[n x 4]
- ✓ 인터럽트로 부터 복귀 (IRET 명령어 실행)
  - ✓ 스택에서 CPU 모드와 IP 값을 팝
    - ✓ IP ← Memory[SP]; SP ← SP+4
    - ✓ CPU 모드 ← Memory[SP]; SP ← SP+4

[참고] x86 인터럽트 벡터 테이블

- ✓ 0: divide error
- ✓ 1: single step (debugger)
- ✓ 2: non-maskable interrupt
- ✓ 3: break point (debugger)
- ✓ 4: signed integer overflow
- ✓ 5~20: device interrupt
- ✓ 21: DOS function call
- ✓ …
- ✓ 128 (0x80): Linux system call

[참고] Interrupt Vector 설정 예

ivector.s

```
.extern _IsrDisk

.org 0x0000
.long IntIgnore     # interrupt 0
.long IntIgnore     # interrupt 1
.long IntNMI        # interrupt 2
.long IntIgnore     # interrupt 3
.long IntIgnore     # interrupt 4
.long IntKeyBoard   # interrupt 5
.long IntDisk       # interrupt 6
.long IntIgnore     # interrupt 7
……
.long int80Isr      # interrupt 0x80
……

.org 0x2000
IntIgnore:
    iret
IntNMI:
    # ...
    iret
IntKeyBoard:
    # ...
    iret
IntDisk:
    # ...
    call _IsrDisk
    iret
```

disk.c

```
IsrDisk()
{
   …
}
```

(참고) ARM의 Exception Vector

Exception type	Mode	VE[a]	Normal address
Reset	Supervisor		0x00000000
Undefined instructions	Undefined		0x00000004
Software interrupt (SWI)	Supervisor		0x00000008
Prefetch Abort (instruction fetch memory abort)	Abort		0x0000000C
Data Abort (data access memory abort)	Abort		0x00000010
IRQ (interrupt)	IRQ	0	0x00000018
		1	IMPLEMENT.
FIQ (fast interrupt)	FIQ	0	0x0000001C

(참고) ARM의 SWI 처리

- ✓ System call 호출 (user mode에서)
  - ✓ SWI n 명령어로 supervisor mode로 전환
  - ✓ SWI n 처리과정
    - ✓ 1. R14_svc ← SWI 명령어 다음 주소 (PC)
    - ✓ 1. SPSR_svc ← CPSR
    - ✓ 2. CPSR[4:0] ← 10011B (supervisor mode)
    - ✓ 2. CPSR[7] ← 1 (disable IRQ)
    - ✓ 2. PC ← 0x00000008 (SWI의 exception vector)
- ✓ System call의 처리 (supervisor mode 에서)
  - ✓ 1. n의 값에 따라서 OS의 적절한 기능 실행
  - ✓ 2. MOVS PC, R14 명령어로 user mode로 복귀
  - ✓ MOVS PC, R14 처리과정
    - ✓ 1. CPSR ← SPSR_svc
    - ✓ 1. PC ← R14_svc

(참고) ARM의 인터럽트 처리

- ✓ IRQ 신호가 CPU에 전달되면
  - ✓ 현재실행중인 명령어 완료 후
  - ✓ IRQ 예외 처리 절차 진행
    - ✓ 1. R14_irq ← 복귀할 주소 + 4 (PC)
    - ✓ 1. SPSR_irq ← CPSR
    - ✓ 2. CPSR[4:0] ← 10010B (IRQ mode)
    - ✓ 2. CPSR[7] ← 1 (disable IRQ)
    - ✓ 2. PC ← 0x00000018 (IRQ의 exception vector)
- ✓ IRQ 예외처리 후 복귀 처리
  - ✓ SUBS PC, R14_irq, #4
  - ✓ 처리과정
    - ✓ 1. CPSR ← SPSR_irq
    - ✓ 1. PC ← R14_irq – 4
- ✓ (참고) FIQ도 IRQ와 유사
  - ✓ R14_fiq 와 SPSR_fiq 사용
  - ✓ CPSR[6] 도 1로 설정 (disable FIQ)