1. Computer System Overview

Processor ⇒ CPU

Main memory(DRAM, SRAM)

1. 휘발성 (컴퓨터 끄면 프로세스 다 날아가는거 생각)

System bus

- memory와 I/O module(disk)과의 통로 역할
- 실제로 프로그래밍하면서 bus error가 여기서 문제 생기는 에러를 뜻해

I/O modules

- 주변 장치들인데, 그냥 키보드, 마우스, ssd, hdd 같은 것들이라고 생각하면 편해

이건 컴퓨터 구조 쪽 내용에 가깝긴 한데

Resister ⇒ 일단 제일 빠름, CPU가 레지스터를 통해서 명령 처리를 함

근데 여기서 data I/O 관련 레지스터는 MAR과 MBR을 보면 됨 (load, store 명령)

Memory Address Register(MAR)

- 데이터를 저장(store)하거나 불러올(load) 주소를 받아옴

Memory Buffer Register(MBR)

- 데이터 자체를 들고 있는 레지스터
- STORE 명령이 호출되면 memory에 데이터를 덮어 씌움
- LOAD 명령이 호출되면 memory로부터 데이터를 받아옴(memory → register)

Program Counter(PC)

- 다음 명령어의 주소를 저장하는 레지스터

Instruction Register(IR)

- PC에 있는 명령어를 가져옴
- 명령어를 가져오게 되면 PC의 주소는 +1 (다음 명령어의 주소를 가리킴)
- 가져오는 명령어의 종류
    1. CPU-memory 간의 명령
    2. CPU - I/O 간의 데이터
    3. Data 처리 명령 (Arithmetic , logic)
    4. control (if else 처리 후 다음 명령 주소)

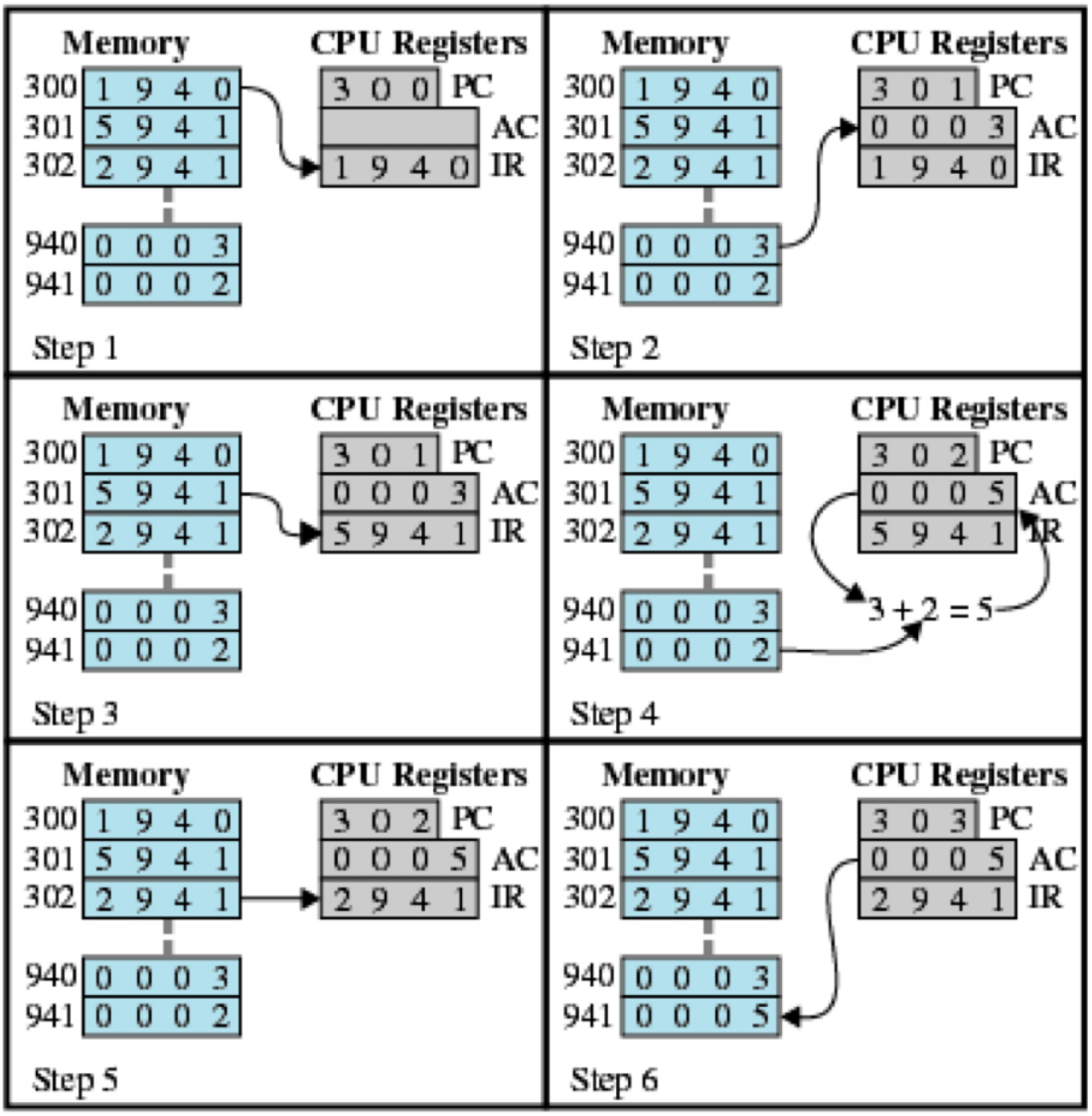
Instruction Execution

1. CPU가 PC로부터 명령어를 읽어 들여(읽고 나면 PC + 1)
2. 그 명령어를 실행

Program Status Word(PSW)

- 주로 조건문에서 사용돼(jump, jal)
- interrupt 허용 여부 (현재 함수에서 interrupt 여부를 세팅할 수 있어)
- authority level 설정 가능

Fetch Execution



Load: 0001 == 1

Store: 0010 == 2

Add: 0101 == 5

Step 1 → Step 2

IR ⇒ PC의 주소에서 데이터 1940을 받아온 상태

1이 load 명령

940 주소의 데이터를 register로 받아옴 ⇒ AC가 0003 으로 채워짐

명령이 수행되고 PC값이 1 증가 ⇒ 301

Step 3 → Step 4

IR ⇒ PC의 주소로부터 데이터 5941을 받아온 상태

5가 Add 명령

941 주소의 데이터를 레지스터에 더해줌 ⇒ 0003 + 0002 = 0005

명령이 수행되고 PC값 1증가 → 302

Step 5 → Step 6

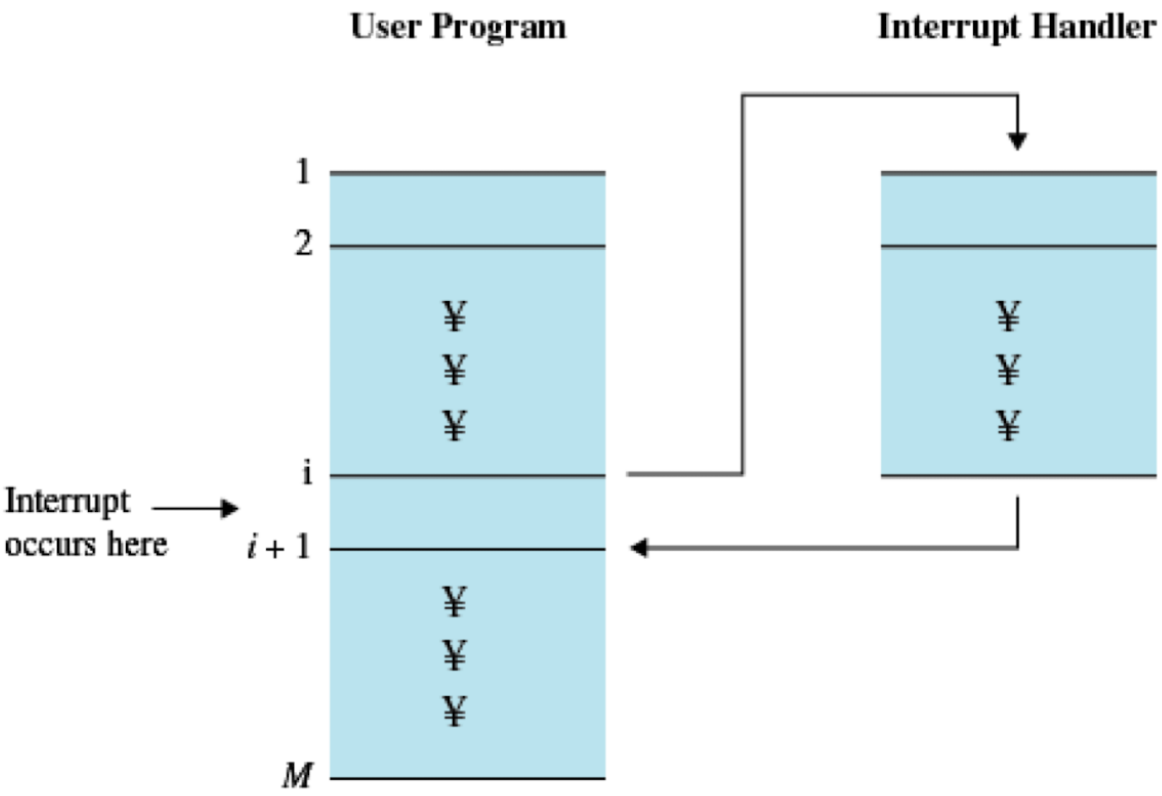
IR ⇒ PC로부터 데이터 2941을 받아온 상태

2가 Store 명령 (register → memory)

941주소에 현재 AC 레지스터 값을 저장

PC 주소는 1증가

Interrupt



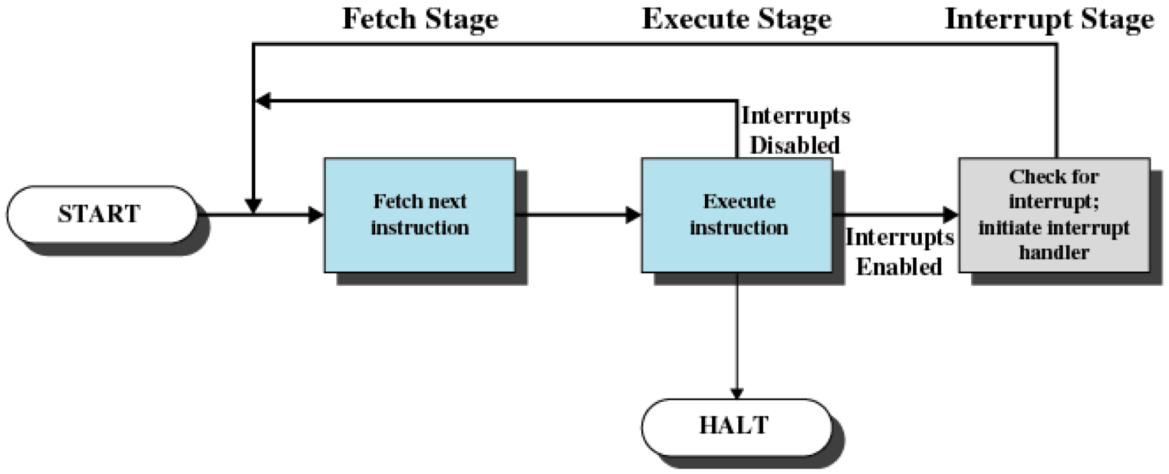
현재 CPU가 수행하는 작업을 중단하고 발생한 interrupt를 수행하고 중단한 작업을 이어서 수행

대부분 I/O 처리할 때 많이 쓰임

(CPU가 작업을 처리하는 속도가 빠른데, I/O가 처리될때까지 CPU가 기다리면 시간적으로 손해니까

interrupt가 발생하면 CPU는 다른 작업을 하고 있다가 I/O가 끝나면 다시 돌아와서 중단됐던 작업을 처리)

Interrupt Cycle



fetch → execute → interrupt

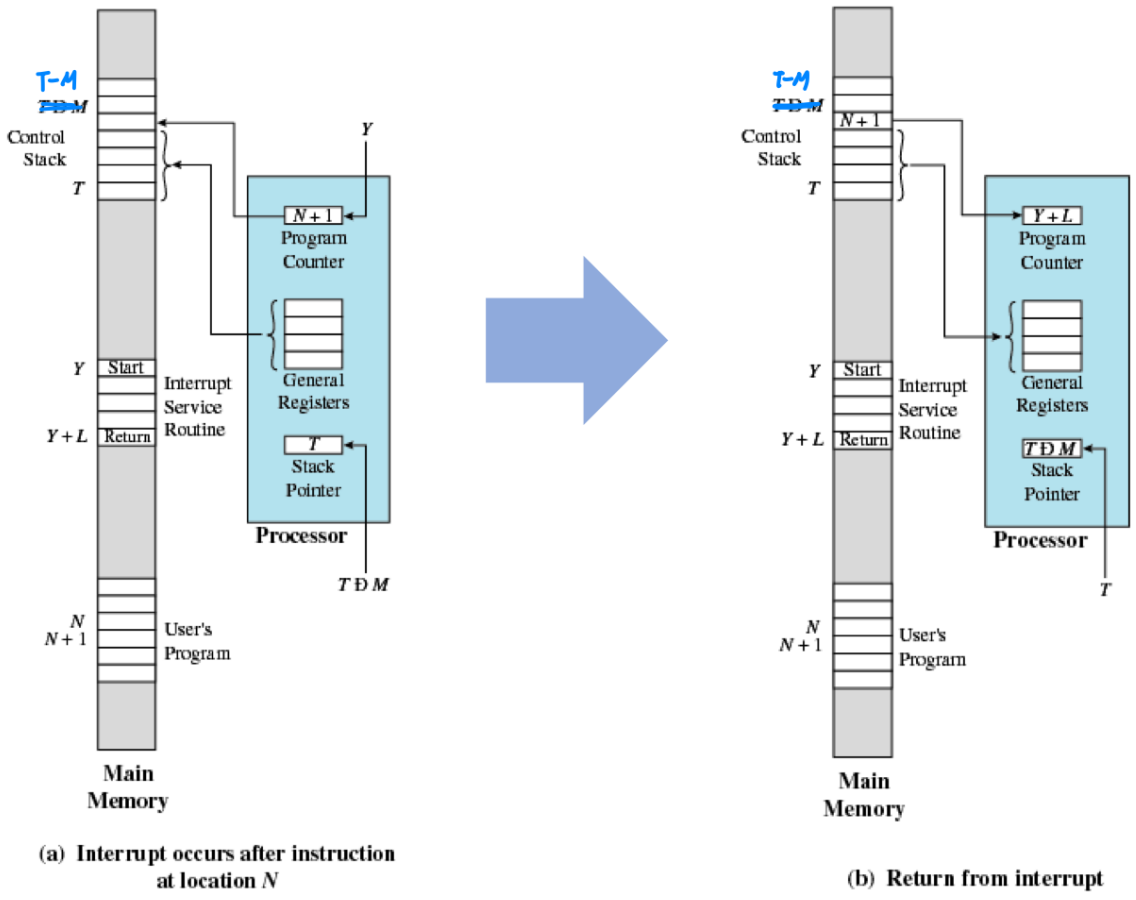
명령어를 fetch

fetch한 명령어를 execute

만약 명령어가 수행되고 있다면 이 때는 interrupt를 막아야 돼 ⇒ 실행 중간에 프로그램이 interrupt가 걸리면 그 때의 데이터는 어떻게 보존할 건지?

만약 interrupt가 걸리면 다음 명령어를 실행하는데 이 때는 interrupt를 켜놓고 가야 interrupt로 갔던 작업이 끝나고 돌아올 때 해당 CPU에 알려줄 수 있어

Memory와 Register 변화(interrupt, 명령 수행시)



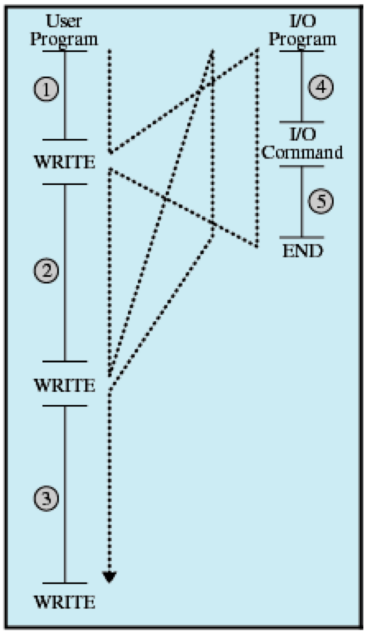
interrupt 발생시

1. stack이 현재 T를 가리킴
2. 현재위치 N+1에서 interrupt 발생(N 주소의 명령이 끝난 이후에 int 발생)
3. CPU가 갖고 있는 모든 레지스터를 stack에 쌓고, interrupt가 발생한 주소를 저장
4. stack pointer에 마지막 stack 주소 저장(T - M)
5. interrupt가 발생했으니 interrupt service routine 수행

interrupt routine 종료

1. stack pointer에서 순차적으로 데이터 가져옴
2. 기존에 정지되었던 주소값 가져오고 (N+1)
3. Register를 차례대로 다 채움
4. 해당 데이터를 가져오면서 stack 주소 차례대로 증가

Program Flow without Ints



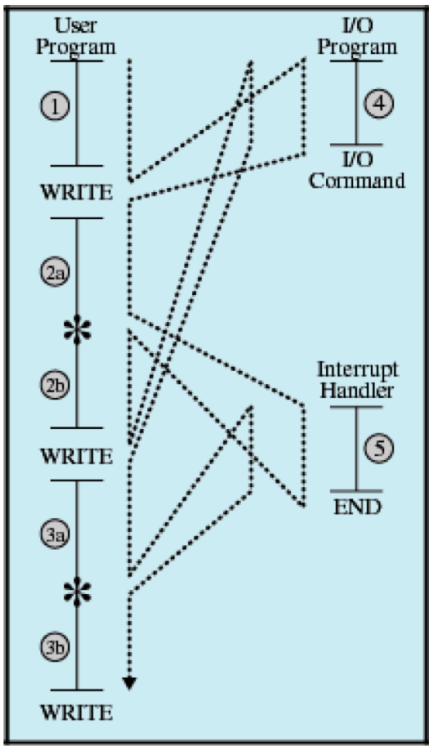
(a) No interrupts

이건 interrupt 처리 없이 프로그램 진행

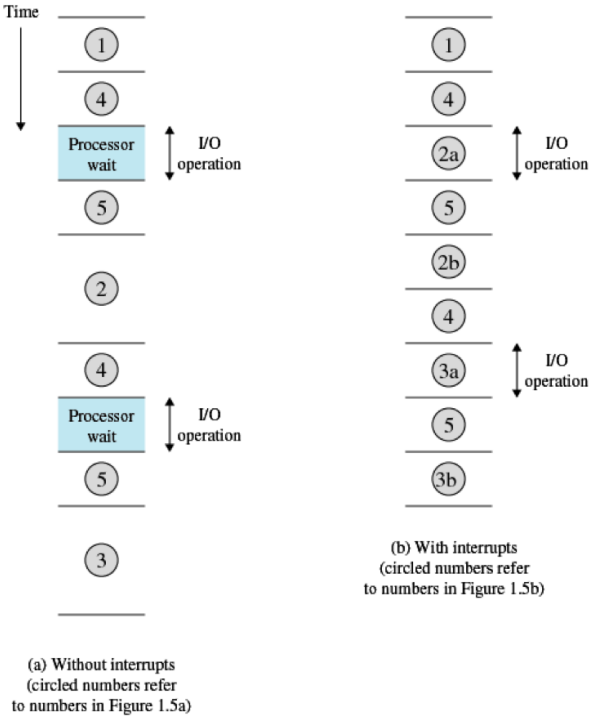
write 발생하면 I/O 다 처리하고 다시 돌아와서 마저 일해

(이 때 CPU는 어떤 작업도 하지 않아)

Short I/O Wait



(b) Interrupts; short I/O wait



(a) Without interrupts (circled numbers refer to numbers in Figure 1.5a)

(b) With interrupts (circled numbers refer to numbers in Figure 1.5b)

Figure 1.8 Program Timing: Short I/O Wait

interrupt 처리한거랑 interrupt 없이 한거랑 시간 비교

여기서는 write랑 *에서 interrupt가 발생

4에서 interrupt가 걸려서 I/O를 수행하고 5에서 I/O 작업이 완료되었다고 알려줌

이 processor wait 시간을 줄이기 위해 일단 일을 마저 수행하도록

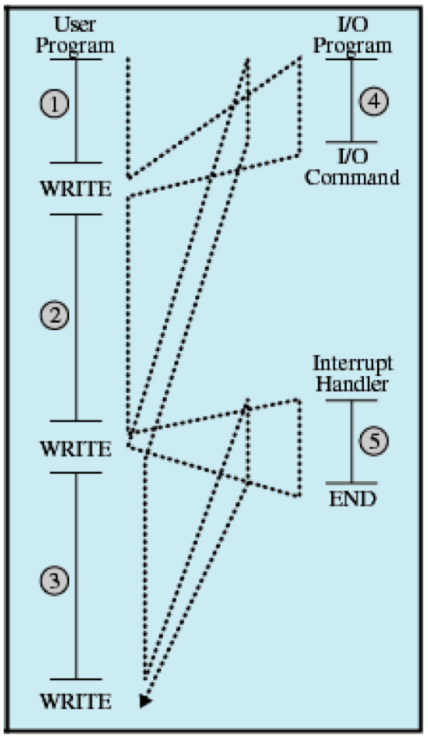
1 → 4(4를 실행하는 동안 2를 수행) → 2a

→ 5(2를 수행하던 중에 interrupt 걸려서 4가 끝났다고 알려주는 interrupt)

→ 2b → 4(write 작업) → 3a → 5 → 3b

그래서 원래는 4랑 5 사이에 계속 기다려야 하는데 그 기다리는 시간에 2와 3을 수행해버려서 시간을 단축시킴

Long I/O Wait



(c) Interrupts; long I/O wait

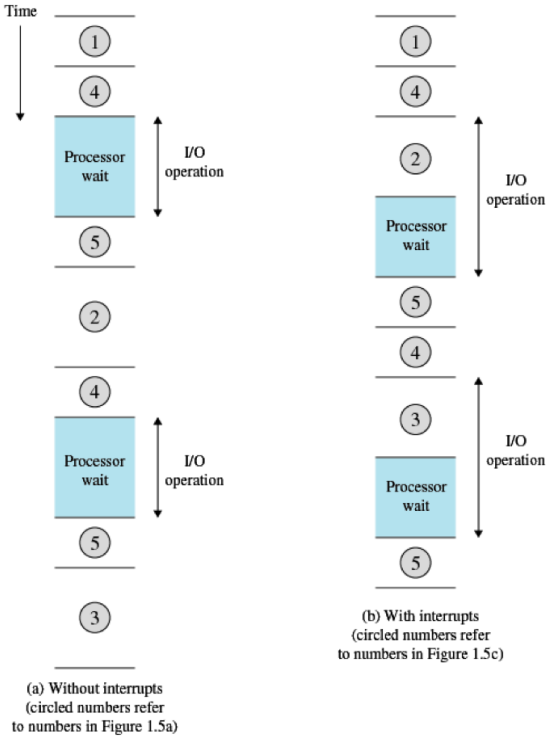


Figure 1.9 Program Timing: Long I/O Wait

이 상황은 interrupt를 기다리는 시간이 너무 길어지는 상황

short I/O와 기본적으로는 동일한데 차이점은

다음 write 명령이 호출되기 전에 이전 I/O 작업이 끝나지 않아서 interrupt를 호출할 수 없는 상황

이 때는 어쩔수 없이 기다려야 돼 (동시에 2개의 명령을 내릴 수 없어서, write작업이 진행중인데 또 write 작업을 시키면 결과는 어떤걸 보존? 데이터 무결성 어긋남)

⇒ solution) 그래서 나온게 Multiprogramming

이제는 하나의 CPU가 하나의 작업을 처리하게 하지 말고, CPU가 여러개의 프로그램을 실행하도록 하자

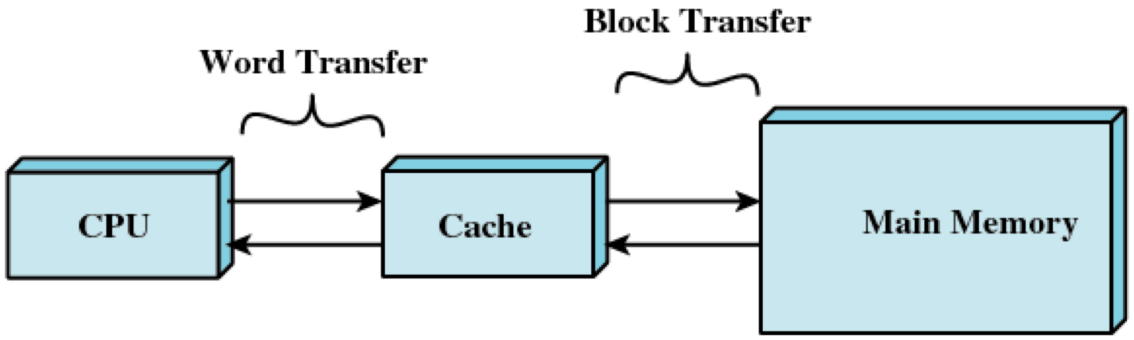
이러면 interrupt가 발생한 순간 다른 프로세스를 처리하러 가면 됨

메모리 구조…

register → cache(SRAM) → main memory → ssd → hdd

뒤로 갈수록 가격이 싸고, 성능이 안좋아

Cache의 데이터 처리 방식



CPU는 기본적으로 Word 단위로 데이터 처리 가능

그래서 memory에서 자주 접근되는 데이터들을 cache에 넣음(Block 단위로)

그러면 CPU는 데이터가 필요할 때마다 cache에서 꺼내 씀

만약 cache에 데이터가 없다면 memory에서 데이터 가져와야 됨

이 때 locality 덕분에 cache가 효과적임

ex)

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        cout << arr[i][j] << ' ';
    }
}

vs

for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        cout << arr[j][i] << ' ';
    }
}
```

어떤 코드가 locality를 잘 지킨 편?

Cache Design

cache size: 큰게 좋을까 작은게 좋을까? ⇒ 너무 크면 성능이 오히려 떨어짐(접근하지 않는 인기없는 데이터들이 cache로 올라와서 오히려 손해)

block size: locality와 관련 있음, instruction을 수행할 때 그 주변 instruction을 수행할 확률이 매우 높기 때문에 (기본적으로 반복문만 봐도 그 주변 데이터를 엄청 사용하니까)

mapping function: cache에서 데이터 찾는데 시간 오래 걸리면 안되니까 바로 찾을 수 있도록 function으로 처리

replacement algorithm: 요즘에는 좀 더 개선된 알고리즘 쓴다고는 하는데 LRU 쓴다고 생각해도 될듯

write policy: 2가지

1. write back policy: can occur every time block is updated
2. write through policy: can occur only when block is replaced
    a. cache에서 memory로 데이터가 내려갈 때 memory를 갱신해주는 정책
    b. 당연히 memory 갱신이 적어지니까 성능은 뛰어남
    c. 근데 multiprocessor 환경에서 cache coherence problem이 발생
    d. 그래서 이거 해결책 아시는분?

I/O 작업은 어떻게 처리되는지

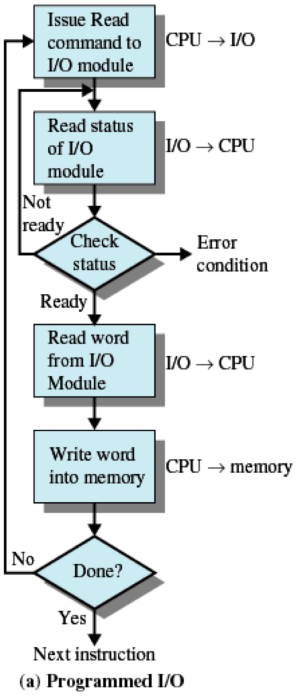
기본적으로 I/O device랑 CPU는 병렬적으로 실행됨

각각의 device는 buffer를 갖고 있는데, 이 때 buffer에다가 데이터를 쓰고 interrupt를 보내

그러면 interrupt handler가 device buffer에서 데이터를 메모리로 가져옴

주변 장치는 main memory에 접근 불가능! CPU만 접근할 수 있음

Programmed I/O



(a) Programmed I/O

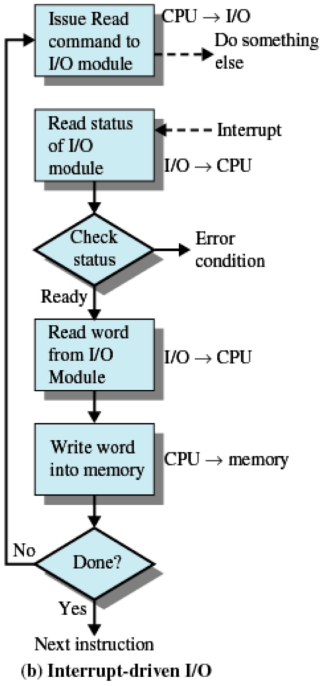
중간에 I/O를 확인하는 loop ⇒ CPU를 쓰면서 대기(busy waiting)

I/O가 완료되었는지 반복문을 돌면서 확인

I/O가 완료되면 CPU가 다음 작업으로 넘어감

interrupt 없음

Interrupt-Driven I/O



(b) Interrupt-driven I/O

이제는 I/O를 CPU가 기다리는게 아니라 interrupt를 받아서 처리

이러면 일단 안쪽 loop는 제거 가능

그런데 이제 문제가 CPU가 하나의 word 밖에 못 읽으니까 하나의 word 단위로 interrupt가 발생

⇒ 너무 비효율적이야

그래서 애초에 device가 memory에 data를 올릴 수 있으면? **(원래는 접근 불가능)**

그러면 CPU가 memory에서 data 받아오면 되니까 하나의 interrupt만으로 block을 올릴 수 있어

그래서 나온게 Direct Memory Access(DMA)

Direct Memory Access

memory에 block 단위로 데이터 교환

이러면 CPU가 메모리 접근해서 데이터 교환하면 되니까 interrupt 횟수가 확연히 줄어듦

그래서 CPU가 이제는 I/O device buffer가 아니라 DMA에 데이터 요청

Bootstrap program

컴퓨터 부팅할 때 쓰는 메모리, 정확히는 부팅할 때 필요한 명령어들의 주소들을 저장하고 있는 메모리

그래서 이 bootstrap이 실행되면 해당 주소에 있는 명령어들을 쭉 실행해서 운영체제를 실행