5강

과제 발표

cpu 3.5Gbps,

memory 2.4Gbps,

HDD 읽기 600Mbps, 쓰기 630Mbps,

SDD 읽기 44.2Gbps, 쓰기 3.4Gbps → SDD가 엄청 빨라졌다.

cpu와 HDD의 속도 차을 봐라

병목은 DISK에 있다.

I/O device model

- Device registers
  - 하드웨어 장치는 장치를 제어하는 controller 가 있음

    controller안에는 register가 있다.

  - Controller 는 대부분 4종류의 레지스터를 가짐
  - Control register, Status register, Input register, Output register

    input과 output은 data를 가지고 있음

    CPU가 어떤 I/O를 하고 싶으면 (CPU가 읽고싶으면) Control register에 READ라는 명령어를 보낸다.

    그러면 그 I/O Device의 상태는 Status register에 표시가 된다.

    I/O라는 것은 CPU가 register에 읽고 쓰는 동작인 것이다.

  - 레지스터들은 메인 메모리의 일부 영역에 매핑
    - 매핑 된 영역의 주소만 알면, CPU에서 접근 가능
- I/O = CPU가 register 에 읽고 쓰는 동작

  device를 동작하게 하려면 CPU가 register에 읽고 써야한다.

polling과 DMA방법이 있다.

I/O 처리기법: Polling

CPU가 Device를 계속 체크하는 방법.

- Loop이나 timed-delay loop안에서 특정 이벤트의 도착 여부를 확 인하면서 기다리는 방법

  계속확인 함
- 인터럽트와 반대되는 개념
  - 폴링은 지속적으로 이벤트의 발생 여부를 확인

- 장치가 매우 빠른 경우, 폴링은 event 처리 기법으로 적당함

  빠른 장치들은 빠르게 이벤트가 올라올 수 있기 때문
  - 느린 장치인 경우?
- 이벤트 도착 시간이 길 경우, 폴링은 CPU time을 낭비
  - 컴퓨터 시스템에서 CPU time은 매우 귀중한 자원!!

    → 가장 귀중한 자원이다.

    최적화는 병목을 없애는 과정이다.

    나중에 CPU가 병목이라면 그게 끝임

    여기까지 최적화를 하는 것이다.

    즉, CPU자원이 가장 귀한 자원이라는 것이다.

- 폴링 뒤에 programmed I/O(PIO) 수행
  - CPU 가 I/O를 위한 데이터 이동을 담당

    CPU가 data를 읽어온다. 그것을 PIO라고 한다.

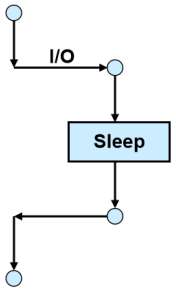
Direct Memory Access (DMA)

- 폴링을 사용할 경우, 모든 동작은 CPU에 의해 진행
- CPU를 장치의 상태 확인 및 데이터 이동에 사용하지 않고
  - I/O를 위한 별도의 장치(DMA) 사용

    CPU, I/O, DMA가 있으면

    CPU ↔ DMA ↔ I/O 끼리 소통함

  - DMA 를 사용하는 경우, I/O 를 호출한 프로세스는 sleep 함



sleep이 끝나는 것은 interrupt가 들어왔을 때이다.

sleep동안에는 다른 프로그램을 수행하는 것이다.

multiprogram을 배울 때 I/O할 때 프로세스가 전환된다는 것이 이것이다.

Q. DMA를 사용하는 경우에 DMA READ가 아직 끝나지 않았는데 CPU가 READ를 더 호출할수도 있나요?

A. 내가 read 명령을 내린 프로세스가 아닌 다른 프로세스에 대해서 READ 명령을 수행하는 것이 가능하다.

Q. DMA가 여러가지 READ를 동시에 처리할 수 있는건가요?

A. 가능하다. 이것이 계속 메모리와 디스크 사이에서 계속 데이터를 넘기는 것이다.

DMA와 CPU

DMA (engine)라는 특정 목적 프로세서가 있음

- CPU와 DMA 간 통신으로 I/O를 수행

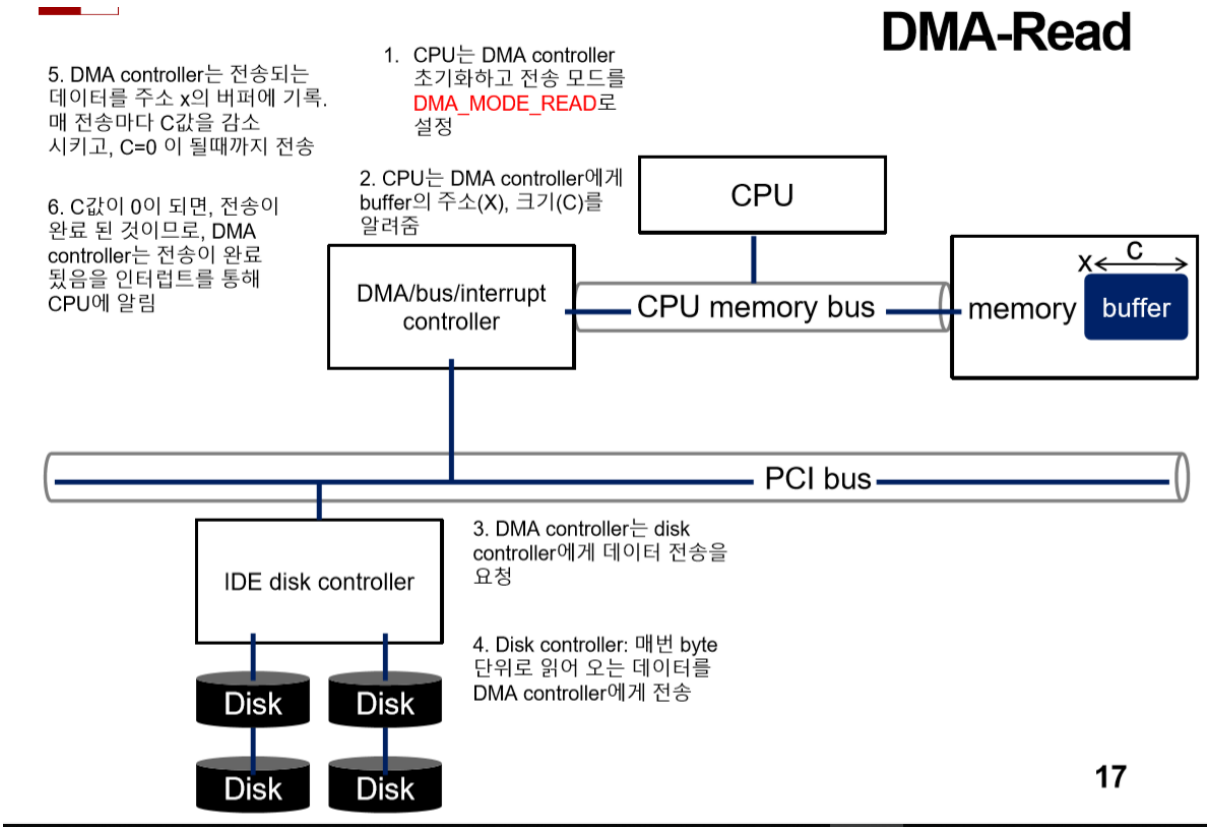
- CPU가 DMA 에게 I/O를 요청하면

  DMA는 CPU를 대신하여 I/O장치와 메인 메모리 사이 데이터 전송을 수행

  - CPU는 I/O 기간 동안 다른 일을 수행

- DMA 는 bus stealing 을 사용하여 동작
- DMA 가 동작을 마치면 CPU에 interrupt 발생시킴

DMA-Read



1. CPU는 DMA controller 초기화하고 전송 모드를 DMA_MODE_READ로 설정
2. CPU는 DMA controller에게 buffer의주소(X), 크기(C)를 알려줌

그리고 CPU는 돌아간다. 즉, I/O에 개입하지 않는다.

3. DMA controller는 disk controller에게데이터 전송을 요청

4. Disk controller: 매번 byte 단위로 읽어 오는 데이터를 DMA controller에게 전송

5. DMA controller는 전송되는 데이터를 주소 x의 버퍼에 기록. 매 전송마다 C값을 감소 시키고, C=0 이 될때까지 전송

6. C값이 0이 되면, 전송이 완료 된 것이므로, DMA controller는 전송이 완료 됬음을 인터럽트를 통해 CPU에 알림

CPU는 그때가서 내가 요청한 read가 끝났다는 것을 아는 것이다.

Q. DMA의 경우는 Disk의 값을 가져올때 Disk controller 등의 장치를 사용하는데, Pooling은 Disk의 값을 가져올때 Disk controller없이 CPU가 직접 값을 가져오는 작업을 하는건가요?

A. polling에서는 CPU가 직접 Disk controller와 대화하는 것이다.

polling을 하면 Status register를 CPU가 체크하는 것이다.

읽거나 쓰거나 하는 명령을 할때는 Control register를 쓴다.

DMA vs Polling: 어느 것이 나은가

- DMA – 추가적인 hardware가 필요하다

  그리고 속도가 빨라질 때마다 DMA를 설계하는 것은 매우 어렵다.

  과거에는 스마트폰에서는 DMA를 사용하지 않았다.

  즉, 가격이 올라간다.

- 성능 – DMA를 최대로 활용하기 위해서는 적당한 parallelism이 필요

  내가 DMA에 요청을하면 CPU는 다른 일을 해야하는데,

  embeded device는 특정 일을 하기 위해 설계되었기에 다른 일을 할 것이 없다.

- 예/질문: 휴대전화 camera pixel을 읽어 들이려고 할 때 DMA가 필요한가?

  필요없다. 옛날에는 화면이 작으면 읽어드릴 픽셀이 작아서 시간도 얼마 안걸려서 DMA가 필요가 없었는데,

  요즘의 DMA는 화면이 너무 커서 픽셀을 읽어내는 것이 너무 오래걸리기에 분리시키는 것이다. DMA를 사용한다.

Q. DMA 와 Polling을 동시에 활용할 수도 있는건가요?

A. Device의 특성에 따라서 polling을 사용하기도 한다.

하나의 CPU에 대해서 여러 I/O가 존재한다. DISK, Keyboard 등

Keyboard는 Polling을 사용한다. 왜냐하면 보내는 데이터의 양이 얼마되지 않기 때문이다.

DISK는 Mbs의 속도이고, flash 메모리는 Gbs의 속도이다. 그럼 어떤 정책이 더 적절한가?

I/O device access 기법: I/O instruction

- CPU는 I/O instruction 을 수행

  특정한 I/O instruction을 수행하는 것이다.
- I/O instruction 은 해당 장치 레지스터의 값을 읽고 씀으로써 장치와 통신함
  - 예: Intel 의 I/O 명령어
    - in, out, ins, outs ...

Memory Mapped I/O

- 장치 레지스터들을 메모리 공간에 매핑하여 사용
  - 장치 레지스터가 메모리 주소에 매핑이 되면, 레지스터들은 주소 공간의 일부로 접근

    Device의 레지스터들을 메모리 주소에 매핑시켜서 메모리 주소 공간으로 취급

- CPU는 일반적인 명령어를 사용하여 I/O 작업을 수행
  - 예: mov, and, or, xor ...

어떤 것이 더 scalarable하겠는가? 어떤 것이 더 확장성이 좋은가?

예시) Memory Mapped I/O가 더 확장용이할 것 같습니다.

I/O Instruction은 Intel 처럼 제조사에서 정해진 명령어으로만 가능하므로

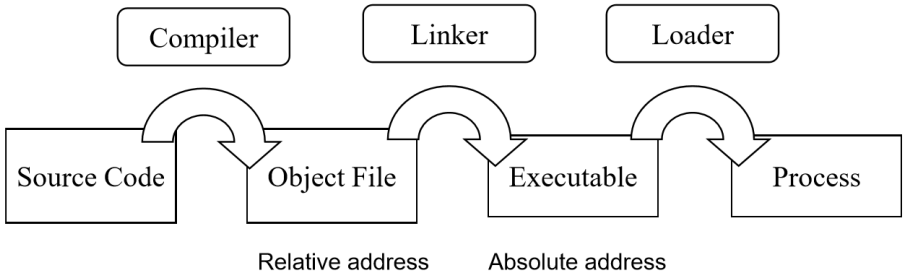
사전에 명령어들을 정의하는 과정이 필요하다 생각합니다.

Memory Mapped I/O는 그저 메모리로 보기 때문에 접근하기더 쉽다. 그렇기에 어떤 device가 붙던 그것에 접근하기는 더 용이하다.

Scope of Operating System

- 프로세스 관리
- 메모리 관리
- 파일 관리

프로그램과 프로세스



Executable이 프로그램이다. 이것이 바이너리 프로그램이다.

그리고 이것이 loader에 의해서 메모리에 올라가면 프로세스가 된다.

프로세스는 동적이며, 프로그램은 정적인 존재이다.

전원이 끊어지면 프로세스는 없어지고, 프로그램은 남아 있다.

프로세스는 persistent하지 않고, 프로그램은 persistent하다.

Object File은 Relative 주소가 있다.

그 이유는 그 파일 안에서의 주소가 정의되기 때문이다.

각각의 주소의 0번지 기준이 있는 것이다.

Executable은 실행을 해야 하기에 절대주소가 필요하다.

소스코드

- 소스코드 (.c file)
    - 프로그램이 수행하고자 하는 작업이 프로그래밍 언어로 표현되어 있다

- 컴파일러 (Compiler)
  - 사람이 이해할 수 있는 프로그래밍 언어로 작성된 소스 코드를 컴퓨터 (CPU)가 이해할 수 있는 기계어로 표현된 오브젝트 파일로 변환
- 오브젝트 파일 (.o file)
  - 컴퓨터가 이해할 수 있는 기계어로 구성된 파일. 자체로는 수행이 이루 어지지 못함. 프로세스로 변환되기 위한 정보가 삽입되어야 함 (그저 바이너리 정보일 뿐이다.)
  - 소스코드 하나에 대하여 오브젝트 파일 하나 생성 (소스코드 10개를 컴파일하면 오브젝트 파일 10개가 생기는 것이다.)
    - 10개 소스코드 → 10개 오브젝트 파일

실행파일

- 링커 (Linker, Linkage editor)
  - 관련된 여러 오브젝트 파일들과 라이브러리들을 연결하여 메모리로 로드 될 수 있는 하나의 실행파일로 작성
- 실행파일 (.exe와 같은 executable file)
  - 특정한 환경(OS)에서 수행될 수 있는 파일. 프로세스로의 변환을 위한 header, 작업 내용인 text, 필요한 데이터인 data를 포함한다
  - 오브젝트 파일은 여러 개이더라도 실행 파일은 하나 임
  - Header 에 대한 설명

  Header-Text-Data의 구조이다.

  → Text는 우리가 쓴 코드

  → Data는 우리가 define한 gloval 변수

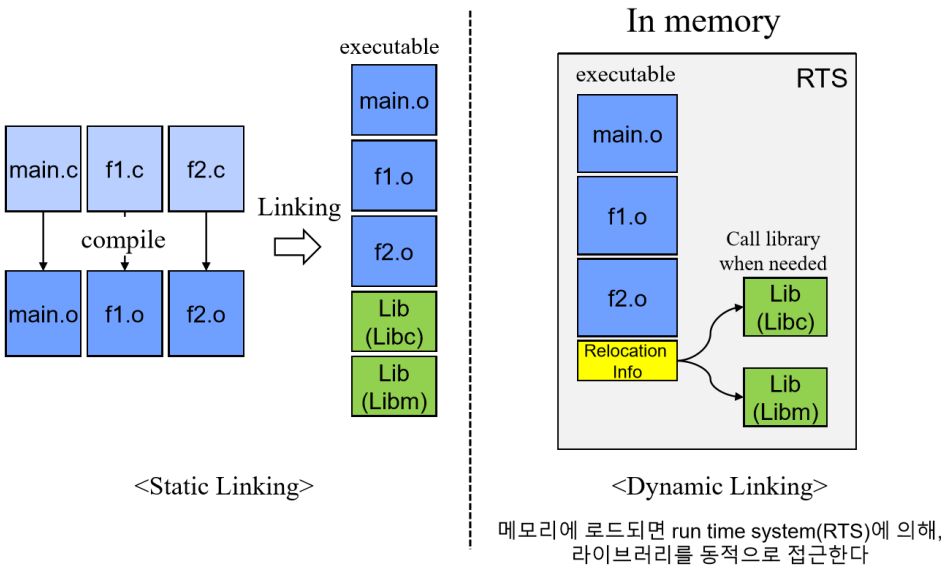
  → 그리고 Header가 executable을 define한다. → a.out, COFF 등이 있다.

- 컴파일러와 링커는 결과물이 수행될 OS와 CPU에 따라 다른 형태의 파일을 만든다

  container는 OS를 가상화하는 것이다. 그렇다면 container는 실행파일과 어떤 관계가 있는가?

  이것이 여기서 말하는 프로그램은 어떻게 다른가?

Static Linking vs Dynamic Linking



Static Linking:

→ 물리적으로 Lib가 excutable에 들어가 버린다.

Static Linking은 그 자체로 완전체이다.

그러나 내용물이 커지고, 똑같은 lib이 계속 올라갈 수 있다.

Dynamic Linking

→ excutable 에서 사용되는 라이브러리만 파악하여 Relocation Info에 넣어서 excutable에 준다.

→ 그리고 실행될 때 Lib을 동적으로 찾아가게 한다.

→ 강력한 보안이 요구되는 경우에는 사용하지 않는 것이 좋다.

dll

→ 동적으로 메모리에 올라가 있다가 그것을 필요로하는 프로그램에게 필요한 함수를 제공해준다.

메모리에 executable과 Libs가 올라가 있는 그 전체 구성을 RTS라고 하는 것이다.

Q. 그렇다면 라이브러리들은 항상 메모리에 올라가있나요??

A. 메모리에 RTS가 올라갈 때 같이 올라간다.

Q. 라이브러리에 동적으로 접근한다는 건 어떤 뜻인가요?? 프로그램에 따라 필요한 라이브러리의 구성 요소만 메모리에 올라간다는 뜻인가요??

A. excutable안에 libs가 저장되지 않고 메모리 어딘가에 저장되어있고, 그 라이브러리의 함수가 필요할 때 접근한다라는 의미이다.

Q. 내가 어떤 프로그램을 실행하던간에 libc와 같은 라이브러리들은 항상 메모리에 올라가있는지가 궁금합니다 ㅠ

A. 이것은 버전마다 다르다. 짐작컨데 libc는 아마도 올라갔을 것이다.

daemon은 시작한 프로세스가 없는데 돌고 있는 것이다. 즉, 시스템이 돌리는 것이다

이 daemon중에 libc가 들어있을 것 같다. 아마도.

Q. 런타임 시스템이 이해가 잘 안되는데 다시 설명해주실 수 있나요??

A. excutable 프로그램이 loader로 메모리로 올라간다.

사실 이 프로세스라고 하는 것은 RTS에서 만들어주는 것이다.

Runtime System (RTS)

- Programming language defines execution model

  프로그래밍 언어는 execution model를 정의하는 것이다.

  RTS는 프로그램을 받아서 프로세스로 만들고 실행시키는 것이다.

  Loader는 단순 메모리 copy, 그저 옮겨주는 역할이다.

  loader는 프로그래밍 언어와 관련이 없다.

  - RTS implements it
  - Behavior not attributable to the program
    - Process memory layout (H-T-D) 이것은 우리가 만드는 프로그램이 결정하지 않는다.

알고리즘을 define해도 그 주변의 enviroment를 정의하지는 않는다.

- Parameter passing between procedures

- Every language has one
  - ART: android runtime
  - Node.js: Javascript

- Provide runtime environment
  - Call main()

    C프로그래밍에서 가장 먼저하는 것이 MAIN이다.

    왜 제일먼저 main()인가?

    → RTS가 프로그램을 호출할 때 제일 먼저 main()을 부른다.

  - Environment variables: HOME, PATH

    HOME, PATH이런 것들은 RTS에서 사용된다.

    argv는 enviroment의 값을 받는 array이다.

Q. RTS는 현재 메모리에 있는 라이브러리 정보를 가져오거나 필요한 라이브러리를 메모리에 로딩하는 것을 OS에 요청하는건가요?

A. yes. RTS가 라이즈러리들을 다 가지고 있고 프로세스를 동작시키기전에 Relocation Info를 다 채워서 필요할 때 lib의 함수가 호출 될 수 있도록 하는 것이다.

Q. RTS와 Process는 1:1 관계인가요?

A. 1 : n이다. RTS에서 C프로세스를 계속만드는 것이다.

Q. RTS도 프로세스인가요?

A. 프로세스일 것이다. 그리고 작업관리자에서는 안보인다.

Q. 프로세스라면 본인 스스로는 어떻게 프로세스로 되는건가요?

A. 커널이 가능하게 한다.

Q. 그렇다면 loader가 메모리에 프로그램을 올려주고, 메모리에 있는 해당되는(프로그램에 맞는) RTS가 올려진 프로그램을 프로세스로 변환해주는 것인가요?

A. 그렇게 이해하면 된다. 정확하다.

container가 RTS까지 같이 묶여서 이미지를 만들면 어디서든지 수행이 가능한 것이다.