[Chapter 2]

소프트웨어융합학과/2020105742/한지훈

1. OS 구조
   A. OS는 커널 뿐만 아니라 인터페이스 역할의 system call, GUI 등의 system software을 포함한 개념
   B. 커널보다 넓은 범위의 개념이라고 할 수 있음

2. User Interface
   A. Graphical User Interface (GUI): 직관적이고, 사용하기 쉽지만 개발자는 CLI를 대신 사용하기를 권장
   B. Command-Line Interpreter (CLI): 리눅스 터미널과 같이 그래픽 대신 명령어로 업무 수행
      I. Bourne shell: shell은 추상화(abstraction)의 개념으로, 커널을 몰라도 사용할 수 있음
      II. Bash(Born again shell): Bourne shell 기반으로 개발되어 현재 Unix, Linux 등에서 기본 쉘로 사용

3. OS의 역할: OS는 개발자를 위한 것으로, 개발자가 프로그램 개발/실행을 편리하게 해주는 환경을 제공
   - 개발자 대신 File 처리, Communications(통신), Background 서비스 관리 등 수행

4. 프로그램 실행 과정
   A. Compiler 통과
      I. 이유: 텍스트 형태의 소스코드(.c)를 실행가능한 형태로 만들기 위해
      II. 구성:
         - 전처리기: # include, define 등으로 라이브러리든 뭐든 간에 뭔지 모르겠지만 일단 뒤에 있다고 가정
         - 번역기: 사용자가 작성한 텍스트를 기계가 이해할 수 있는 형태인 binary code로 변환
   B. .obj 파일 생성
      I. 특징: 전처리내용이 아직 처리되지 않았고, 하나로 합쳐져야 하기 때문에 실행이 아직 되지 않음
   C. Linker 통과
      I. 역할: .obj 파일들 결합, # include로 처리된 stdio.h 등의 라이브러리를 코드 파일과 결합
   D. 최종 실행파일(a.out) 생성, 실행하여 결과값 확인

5. Linker
   A. 종류:
      I. Static linking
         - 방법: obj 파일에 라이브러리 함수의 binary code를 직접 결합하여 사용하는 방식
         - 장점: 시스템 라이브러리가 없는 환경에서도 실행 가능
         - 단점:
            - 여러 개 실행시 중복 code가 존재
            - 실행 함수가 많아질수록 메모리 효율 낮아짐
      II. Dynamic linking
         - 방법: 라이브러리 copy는 1개 존재하며, 함수에서 주소를 참조하여 사용하는 형태
         - 장점: 메모리 사용 효율 높음, binary code size 작음, compile time 적음
   B. Default 방식: Dynamic(커널 프로그램에 printf() 등의 code가 중복되는 것은 비효율적이기 때문)
   C. Static 쓰는 경우: Embedded 환경
      I. 이유:
         - Embedded 환경에서는 사용할 수 있는 하드웨어의 수준에 제약이 있음
         - 라이브러리를 올려서 쓸 수 없는 경우가 존재할 수 있음

6. (Operating) System Call Service

A. 개요: code가 OS 안에 있는 Function Call 등 발생시 user/kernel간의 mode change가 일어남

B. Function Call 예시: open(), read(), write(), close(), …

C. 특징: 스케줄러에 의해서 실행되는 어플이 선택되기 때문에 user mode로 돌아올 때 read 실행한 어플이 1순위 아닐 수도 있음

D. 직접 실행하는 경우: 적음

I. 이유: 함수 등 실행 시 라이브러리 딴에서 간접 실행되는 경우 더 많음

- Ex) printf() 실행 시 간접적으로 write() 하기 위해 system call 발생, 끝나면 user mode로 돌아옴

7. System call standard API

A. 코드 예시: ssize_t read(int fd, void *buf, size_t count)

I. fd: file descriptor(Unix, Linux에서 관리를 위해 부여되는 I/O 디바이스의 고유 번호를 의미

II. *buf: 파일로부터 읽어 들인 내용을 저장하기 위한 공간으로, 배열을 사용하여 저장

III. count: 읽어 들일 파일의 내용의 크기를 지정하며, 일반적으로 바이트 단위로 기술됨

8. Vector table

A. 개념: 종류가 많은 HW interrupt 중에서 잘 쓰이는 것을 넘버링

B. 이유: 인덱싱을 통해 처리를 빠르게 하여 OS의 개입 줄이고, 오버헤드 줄이기 위함

C. 예시: Timing: 0번, DMA: 1번, …

D. 활용: Software, System call도 마찬가지로 vector table을 통해 종류별로 인덱싱하여 성능 증가

9. Parameter passing

A. 개념: param 수와 상관없이 커널을 통해 레지스터에 직접 할당하여 속도를 높임

B. 원리: 레지스터는 SRAM으로 속도가 빠르고, CPU에 가까워서 속도 높일 수 있음

10. Linux, Windows

A. 윈도우에서 만든 코드가 Unix에서 돌아가지 않는 이유: 서로 호환되지 않음

I. 이유:

- 명령어 이름이 서로 다름

  ex) 새 프로세스 생성: 리눅스: fork()/윈도우: CreateProcess()

- 한 OS에 정의된 명령어가 다른 언어에 없는 경우도 있음

  ex) 신호 보내기: 리눅스: kill/윈도우: 없음

- 한 OS에서 만든 코드를 다른 OS에서 실행하면 알아먹지 못함

II. 해결법:

- 리눅스는 POSIX라는 통일된 규격 생성
- unix 기반 OS들 간에는 100% 호환성 보장(한 OS에서 작성된 코드는 다른 unix 기반 OS에서도 동작)

11. Operating System Structure

A. Simple structure

I. 구조: 2세대 batch system

II. 특징:

- 한 번에 하나의 프로세스만 실행 가능(동시에 여러 개 프로세스 동작 불가)

- 매우 단순하여 OS로 부르기 애매한 면이 있음

III. 예시: MS-DOS

B. Monolithic kernel

I. 구조: Integrated Kernel(커널의 모든 기능 포함), System Call, Hardware

II. 장점: File system, virtual memory, I/O driver 등 모든 기능들이 통합되어 있어 성능이 높음

III. 단점:

- 각각 기능의 dependency가 매우 높음
- code 하나 수정되면 다른 여러 기능에 문제가 생길 수 있음
- Product 개발 비용보다 유지보수 비용이 많이 듦 > 해결법: 객체 지향 프로그래밍

IV. 예시: Traditional Unix

C. Layered approach

I. 구조: Hardware, User Interface 등 모든 기능들을 계층적으로 분리하고, 계층 간 dependency 없앰

II. 특징: 이상적인 개념으로, 실현이 불가능함

- 이유: OS 기능들은 쉽게 분할할 수 없고, 필연적으로 서로 간에 dependency 발생

D. Microkernel

I. 구조: Microkernel (커널의 핵심 기능), server(커널 나머지 기능), System Call, Hardware

II. 특징:

- Microkernel에는 CPU 스케줄링, 메모리 관리 등의 필수적인 기능들만 존재
- 커널의 나머지 기능들을 어플에 갖다붙힘
- 문제:
  - 커널(kernel mode)에 있어야 할 기능들이 전부 user mode에 존재함
  - kernel mode로 진입하는 횟수가 크게 증가하여 성능 감소

E. Modular approach

I. 구조: 하나의 커널이 존재하고, 커널 각 기능들을 모듈화 시켜놓은 형태

II. 특징:

- 이전 구조들의 단점 모두 해결 가능
- Dependency를 줄이고 유지보수가 용이하게 하여 Monolithic kernel의 문제 해결
- 커널모드 진입횟수를 줄여서 Microkernel의 문제 해결

III. 예시: 현재의 리눅스, Solaris, …

12. Hybrid approach

A. Android: HW dependency를 줄이기 위한 Dalvik virtual machine 존재

- OS라고 불리기 애매한 면이 있음

B. iOS: virtual machine이 별도로 존재하지 않음

C. 차이

I. 애플은 HW, SW를 자체제작하기 때문에 HW dependency 걱정할 필요 없음, VM 필요 없음

II. 동성능 기준으로 VM 가격차이로 인해 Android가 좀 더 비싼 경향이 있음

13. Process Management

A. 개요: 프로세스는 프로그램이 메모리로 올라가 실행되는 것으로, 프로세스 관리는 곧 CPU 자원관리 의미

B. 구성

- Scheduler: Ready 상태의 여러 개의 프로세스 중에서 누가 먼저 CPU를 사용할 것인가
- IPC(Inter-Process Communication): 프로세스 간에 통신을 하고 data를 주고받기 위한 방법
- Synchronization: data에 프로세스 여러 개가 동시 접근하려 할 때 실행 순서 지정