Ch2. Operating System Structure (완)

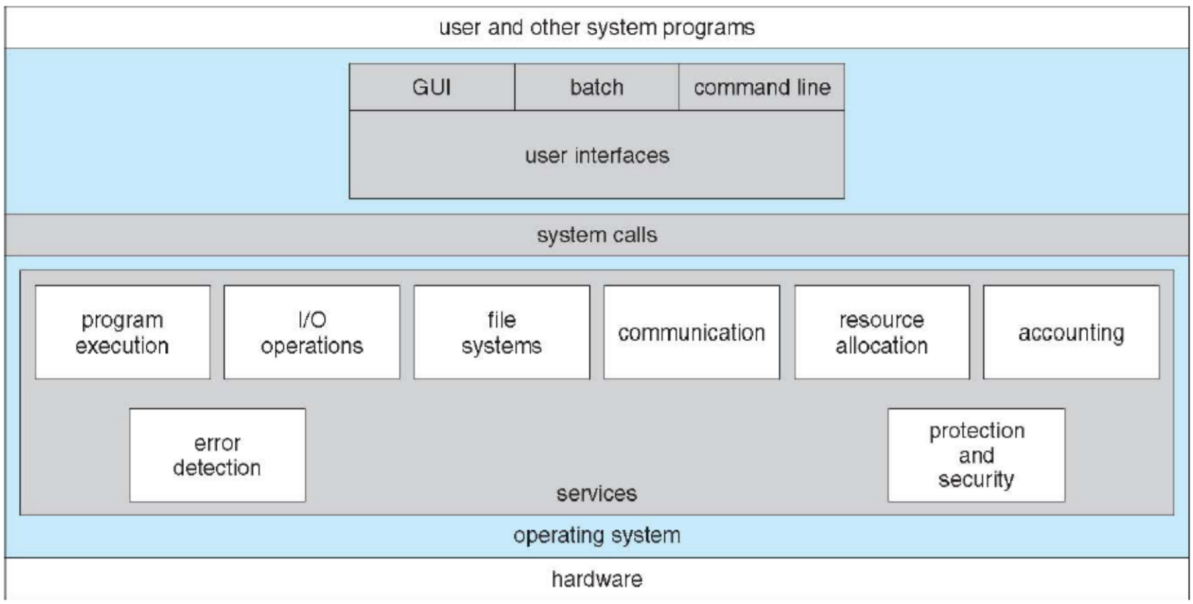
Created	@May 29, 2020 6:25 PM
Tags	운영체제

Not only the concepts, let's learn how they work.

p.2

OS Services

OS service의 관점



- **OS는 프로그램과 하드웨어 사이에서, 프로그램의 실행을 위한 환경을 제공한다.**
  - Applications (User programs, Other system programs)랑 Hardware 사이에 OS가 존재함을 볼 수 있다.
  - OS는 유저와 시스템에게 다양한 서비스를 제공한다.

p.3

OS Services for User Support

Remind: OS는 유저와 시스템에게 다양한 서비스를 제공한다.

- **UI (User Interface)**
  - CLI : Command Interpreters (shell이라고도 불림)

    $ rm file.txt
    → shell이 path에서 'rm' 프로그램을 찾아서, 메모리에 로드해서, 'file.txt'를 파라미터로 해서 실행한다.

  - GUI
- **Program Execution**
  - 시스템은 프로그램을 메모리에 **로드**하고, **실행**하고, **실행을 끝낼 수 있어야** 한다.
- **I/O operations**
  - OS는 효율과 보안을 위해, **사용자가 직접 입출력 장치를 조작하지 않고, OS를 거치도록** 해야 한다.
- **File-system manipulation**
  - 파일 **읽기, 쓰기, 생성, 삭제, 검색**. + 사용자가 파일에 접근하지 못하도록 막기도 한다.
- **Communications**
  - **프로세스끼리 정보 교환**
    - **Shared memory** : 여러 프로세스가 메모리의 한 부분을 공유하도록
    - **Message passing** : 프로세스 간에 message(정보)를 주고받는 것.

    Message passing이 Shared memory보다 더 느리다.

- **Error detection**
  - OS는 하드웨어 수준에서 발생하는 에러 (Memory error, Power failure, ...), 소프트웨어 수준에서 발생하는 에러 (bit-flip, ...)를 탐지하고, 고칠 수 있어야 한다.

OS Services for Efficient Operation

- **Resource Allocation**
  - 자원 : CPU 사이클, 메모리 공간, 메모리 대역폭, 파일 시스템 공간, 파일 입출력 대역폭, ...
- **Accounting**
  - 어떠한 프로세스가 자원을 얼마나 사용했는지 계속해서 추적
- **Protection and Security**
  - Multi-user environment에서, 시스템 자원으로의 모든 접근은 통제되어야 한다.
    - 동시에 Resource에 접근하는 프로세스들이 많을 수도 있고, Visibility 문제 등...
    - 프로세스끼리는 서로 방해하지 않아야 한다.

p.6

System calls

예시) 파일 하나 열어서 다른 파일로 copy하는 작업. 여기서는 어떤 시스템 콜들이 사용되는가?

1. **사용자로부터 파일 이름 입력받기** - 시스템 콜을 반드시 사용해야만 한다. 키보드 입력, GUI에서도 마우스 입력, 창 띄우기 등등...
2. **두 파일을 open한다.** - 파일 열기, 에러나면 에러 표시하기 등등... 이런게 다 시스템 콜이다
3. **데이터 복사해서 Destination file에 write.** - 어느 단위만큼 읽고 쓰고 반복. loop. 이 단계에서 또 에러 생기면 에러 출력 등등...
4. **Close both files.** - 당연히 시스템콜
5. **전송 완료 메시지 표시, 전송 끝, 프로그램 종료**

Open, close는 추상적인 단어.

p.7

- **Conclusion : 시스템 콜들은 간단한 작업에도 엄청 많이 쓰인다.**
  - 1초에 몇천 개 나온다.
- **프로그래머들은 API를 이용해서 시스템 콜을 간접적으로 사용한다.**
  - API : 라이브러리와 함수의 집합
  - Windows API, POSIX API, Java API 등이 있다.
  - API끼리 계층을 이루기도 한다. 상위 API가 하위 API를 부른다던가...
    - Java API는 libc를 사용한다.
- 그럼 왜 API를 써서 시스템 콜을 간접적으로 call하는가? 직접 call하지 않고 API를 부르게 하는 이유?
  - **시스템 콜들을 직접 사용하는 것은 어렵다.**
    - 파라미터, 사용법 등...
  - API는 사람이 봤을 때 사용하기 편하게 이루어져 있다.
    버퍼링 같은 부가적인 서비스들을 제공하므로, 프로그래머 입장에서는 API 쓰는 게 훨씬 편하다.
  - **Portability**
    - 컴퓨터마다 전혀 다른 내부 시스템 콜 구조를 가지고 있더라도, 우리는 API를 기준으로 짜 놨기 때문에 그대로 동작한다.
    - 컴퓨터에 맞게 프로그램을 수정할 필요가 없다.

p.8

System Call Interface

User program이 어떠한 과정으로 System call을 부르는가?

1. User program(libc 등도 포함)이 open() 등 System call을 호출한다.
2. System Call Interface에서 핸들링. User mode → Kernel mode로 전환.
3. System call function pointer에서 sys_open() 등 함수 호출, 작업 수행.
4. 다시 System Call Interface로 넘어가서, Kernel mode → User mode로 전환.

- Caller는 시스템 콜이 어떻게 구현되어 있는지 알 필요 없다.
  단순히 시스템 콜 이름, 파라미터, 무엇을 리턴하는지 정도만 알면 된다.

p.10

System Call Types

시스템 콜들을 6개 카테고리로 분류해볼 수 있다.

- **Process control (프로세스 제어)**
  - fork, exec, exit, ...
- File manipulation (파일 관리)
  - create, open, close, read, write, lseek
- Device manipulation (장치 관리)
  - open, close, read, write, ioctl
- Information maintenance (정보 유지)
  - time, date, dump, pid
- **Communications (통신)**
  - open, close, connect, accept, read, write, send, recv, pipe, mmap, sendfile, ...
- Protection (보호)
  - chmod, umask, chown

p.11

System call types : Process Control

Remind : Difference btw Program and Process
Process : instance of Program.
하나의 프로그램이 여러 프로세스를 가질 수 있다. 프로그램은 그저 코드덩어리일 뿐.

Process Control

- 컴퓨터에는 많은 프로세스들이 돌아가고 있다.

- 프로세스의 생성/종료, 메모리 할당/해제, ... OS는 프로세스들을 관리해야 한다.

예시 시나리오 몇 가지

- Error handling
  - 프로세스를 종료하려면 → **Clean-up 과정**이 필요하다.
  - **Memory dump를 만든다.**
    - 프로그램이 오류로 인해 종료되었다면, 그 때의 메모리 상태 복사본을 만들어 두면, 프로그래머가 Memory dump를 보고 어디가 잘못되었는지 디버깅할 수 있다.
  - 메시지를 출력하고, 프로세스를 끝낸다.
  - 다음 실행할 명령어를 결정하고, 실행한다.
- 다른 프로그램 로딩, 실행
  - 새로운 프로세스가 종료되면 어디로 Control을 넘겨줄 것인가
  - Concurrently하게 실행할 것인가, 말 것인가
- Multiprogramming에서도 프로세스들을 관리해야 한다.
  - 프로세스 Attributes들을 Get/Set (Priority, Time limit, Resource limit, ...)
  - 필요 없는 프로세스들을 선택적으로 종료

p.13

System call types : Communications

Communication : 네트워킹, 소켓 프로그래밍 등...

커뮤니케이션에는 두 가지 모델(접근, 구현)이 있다.

- **Message passing**
  - 한 프로세스가 다른 프로세스에세 메시지 전달.
  - Connection이 처음에 먼저 열려있어야 한다.
  - 프로세스는 메시지 전송 요청만 하고, 실제 전달은 OS에 의해 완료된다.
  - 장점
    - OS가 전달해주기 때문에 동기화 문제 X
  - 단점

        - 성능이 다소 떨어진다.
- **Shared-memory** :
    - 기본적으로, 프로세스는 다른 프로세스의 메모리 공간에 접근할 수 없다.
    - 공통으로 사용하는 메모리 공간을 만들고, 서로 공간을 공유한다.
    - 프로세스들은 OS에게 시스템 콜로 Shared memory 접근 요청을 보내고, OS가 이를 받아 프로세스들에게 알린다.
    - 장점
        - 성능이 좋음
    - 단점
        - 동기화 문제. 각 프로세스들이 언제 메모리에 접근할지 모른다.

p.14

Other types

- File management
- Device management
- Information maintenance

p.15

OS Design and Implementation

OS 설계의 목표

- **사용자 관점**에서의 목표
    - **사용이 쉬워야 한다.** Reliable하고, 배우기 쉬워야 하고, 빨라야 한다.
- **시스템 관점**에서의 목표
    - 구현이 쉬워야 한다. 관리(패치 적용 등)가 쉬워야 하고, 작동이 쉬워야 하고, Flexible하고, 효율적이어야 하고, 빨라야 한다.

막연한 요구 사항일 뿐이고, 명확한 해결책은 없다. **절충이 필요**하다.

메커니즘(Mechanism)과 정책(Policy)을 분리 → OS 설계를 더 모듈화할 수 있다.

- 메커니즘 : 무엇을 **어떻게** 할 것인가?
- 정책 : **무엇이** 되게 할 것인가?

정책이 더 상위의 레벨로, 메커니즘을 사용하는 것.

정책의 변경이 메커니즘의 변경을 필요로 한다면, 바람직한 설계가 아니다.

OS implementation

어셈블리어로 구현?

- 재미없고, 어렵고, 에러나기 쉽고, 포팅하기 어렵다.

High-level lang으로 구현하면? (C, C++: lowest high-level lang)

- 구현이 빠르고, 이해하기 쉽다.
- 개선된 컴파일러 → 개선된 코드가 생성된다.
- 포팅이 쉽다.
    - MS-DOS 어셈은 에뮬레이터 필요. 리눅스는 C로 짜여져 있으므로 Native하게 구동 가능.
- 그래도 단점...
    - 성능 - 최적이 아닌 코드가 생성될 가능성이 있다.
        - 그래도 요즘 컴파일러 기술은 괜찮다.
        - 복잡한 Dependencies 다루기에는 컴파일러가 더 나음
        - 치명적인 코드(critical code)들은 High-level lang으로 먼저 작성 후, 최적화된 코드로 대체할 수 있다.

p.18

OS Structure

- **Monolithic structure**
    - 모든 게 하나로 합쳐져 있다. 모듈로 나뉘어지지 않는다.

    - *결코 좋은 디자인이 아닌 걸 알지만, 고칠 수 없다. 모든 걸 다시 Rebuild할 강력한 이유가 없다.*
- **Componentized, Modularized approach**
    - Component (Module) : Input, Output, Functions로 구성된, Well-defined된 시스템의 일부

Simple Structure

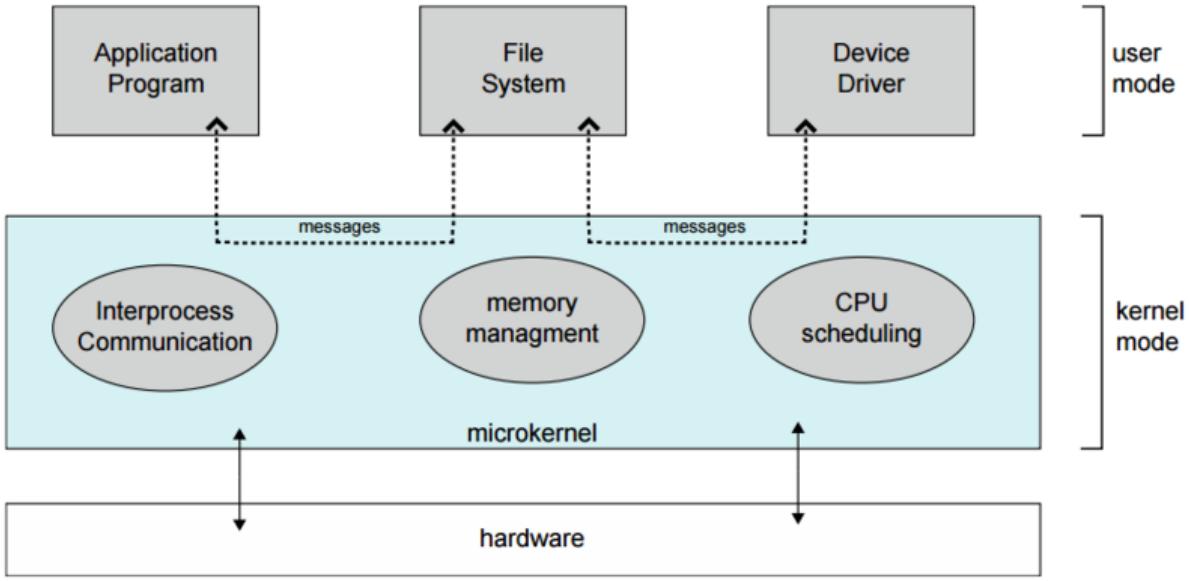
- **MS-DOS**
    - 구조가 있긴 했지만, 그렇게 잘 나뉘어지지는 않았다.
    - Application(사용자 프로그램)이 입출력 루틴에 직접 접근할 수 있었다.
        - → Application에 문제가 생기면, 전체 시스템에 문제가 생겼다.
- **Original UNIX**
    - 사용자 계층과 하드웨어 사이의 'Kernel'이 모든 기능을 제공했다.
    - 이것도 여전히 하나의 계층이 너무 많은 일을 했다.

Layered Approach

- OS가 여러 개의 레이어들로 나뉘어짐.
    - Layer 0 (최하위 계층) : 하드웨어
    - Layer 1, 2, 3, ...
    - Layer M (최상위 계층) : User Interface
- 하나의 레이어는 인접한 레이어끼리만 통신한다.
    - 상위 레이어는 하위 레이어의 작업을 수행시키고,
      하위 레이어는 상위 레이어에서 Call할 수 있는 작업을 제공한다.
- 장점
    - 구현이 간단하다. 디버깅이 쉽다.
        - 레이어들은 단지 자신의 바로 하위 레이어의 작업만 수행할 수 있도록 설계된다.

        - 최하위 계층부터 디버깅하다가, 어느 계층에서 오류가 발생하면 그 계층만 디버깅하면 된다.
- 단점
    - 레이어를 정의하는 것은 어렵다.
        - OS의 어느 부분이 어느 레이어에 들어가야 하는지 명확히 정의할 수 없다.
    - 레이어들을 거치는 과정이 오버헤드를 야기할 수 있다.

Microkernels



- 커널을 최소화하자.
- **필수적이지 않은 함수들은 User-level library로 구현**하자. → 대부분의 기능을 User space로 옮김.
- **마이크로커널 : 커널에서 핵심적인 요소만 남긴 가벼운 커널** → 마이크로커널은 서비스 콜 전달 등 단순한 기능만 제공.
- Application과 서비스 사이의 효율적인 커뮤니케이션 제공
- 장점
    - 확장 가능
    - 포팅(다른 시스템에 이식)이 쉬움

    - More secure and reliable
        - 대부분의 기능이 Kernel space가 아닌 User space에 있기 때문에, 문제 있는 서비스는 해당 부분만을 재시작하여 해결할 수 있음
        - 대부분의 코드가 Protected memory에서 실행되므로 안전
- 단점
    - 비교적 낮은 성능
        - Context switching, System function call의 오버헤드 등

Ch2. OS Structure 끝!