운영 체제

CH 0.

DAY-1

- 기본적인 동작

  1) 파일의 내용을 저장할 공간을 할당 받고 저장. (= Disk Block, 일반적으론 4kb 단위)

  2) Disk에 inode라는 파일의 정보(만들어진 시간, 사람, 접근 제어 정보 등)을 담고 있는 공간을 할당 받음.

  3) Inode와 파일을 연결함

  4) 컴파일러를 통해 파일을 수행 가능한 바이너리 파일(0,1로 이루어진 파일)로 만듦.

  5) 바이너리 파일을 실행할 때에는 task라는 객체를 만듦.

  6) task라는 객체를 만들기 위해 바이너리 파일을 우선 RAM에 load함.

  7) Task는 RAM의 일부 공간(page frame)을 할당 받고 관리.

  8) 기존에 존재하던 task들과 경쟁하며 CPU를 할당받기 위해 노력함.

  9) 운영체제는 이런 task들에게 CPU를 할당해줌. (대표적으로 Round-Robin[1] 방식이 유명)

- 임베디드 부팅을 위해선 3개의 파일이 필요하다.

  1) Boot loader: 장치들을 초기화해주고 커널, 파일 시스템을 램에 복사해주는 기능

  2) Kernel: 프로그램에 대한 자원 할당을 담당한다. 인터럽트 처리기와 스케줄러 등을 포함한다. 주기억장치에 저장된다.

     zImage와 uImage는 리눅스 커널 중 하나다. uImage는 zImage의 특정 명령어를 압축한 것이다.

  3) 하나의 압축 파일로, 기본적인 명령어, 라이브러리 파일을 압축하여 파일 시스템으로 만든다.

[1] 일정시간 동안 한 task가 CPU를 사용하고 시간이 지나면 다음 task가 사용하는 방식

- Daemon: 한번만 실행해 놓으면 지속적으로 동작하는 프로그램 또는 명령어
- Shell: 이용자와 시스템 사이의 대화를 가능하게 해주는 명령 해석기. 이용자가 입력한 문장을 읽어, 그 문장이 요청하는 시스템 기능을 수행하게 해준다. Kernel과 달리 보조 기억장치에도 저장이 될 수 있다.
- Idle 상태: 하드웨어 장치에서 다음 수행 명령을 기다리는 동안 별다른 작업을 수행하지 않고 기다리는 상태
- Super Block: 유닉스 시스템에서 파일 시스템의 상태를 설명하는 블록
- metadata: 다른 데이터를 설명해주는 데이터. Contents의 위치와 내용, 작성자 정보, 이용조건, 내력 등이 담겨있다. metadata는 보통 데이터를 표현하기 위한 목적과 데이터를 더 빠르게 찾기 위한 목적이 있다.

CH1.

- Microkernel[2]: 운영체제의 기본적인 기능을 제공하는 kernel만 남기고 소형화한 것.
- Linux는 monolithic structure[3]를 기반으로 설계되었다. (실제로 kernel 자체는 monolithic이지만 파일 시스템, 디바이스 드라이버, 스케줄링 등 여러 부분에서 모듈을 지원하기에 하이브리드라고 볼 수 있다.)

[2] Windows NT, Mac OS에서 사용

[3] 모듈화와는 다르게 독립성이 없는 소프트웨어 블록들의 집합으로 이루어진 소프트웨어.

Monolithic는 최적화에 좋고, 속도가 빠르며, 경로를 단축시킬 수 있지만 유지보수가 힘들다.

CH2.

Day-2

- OS = resource manager (system call을 통해 task가 resource를 활용할 수 있게 해준다.)
  1) Physical resource

     CPU, Memory, Disk, terminal, network 등 시스템 구성요소와 주변 장치
  2) Abstract resource (matched with physical)

     물리적 자원을 관리하기 위한 추상화 객체들

     Ex) task, memory segment, page, protocol, packet
  3) Abstract resource (Not matched with physical)

     Security, ID access control

- Kernel
  1) task manager

     task의 생성, 실행, 상태 전이, 스케줄링, 시그널 처리, 프로세스 간 통신 지원
  2) memory manager

     Segment[4]와 page[5] 관리
  3) filesystem

     파일 생성, 접근 제어, inode 생성, 디렉터리 관리, super block 관리
  4) network manager

     네트워크 장치를 socket으로 제공, TCP/IP 같은 통신 프로토콜
  5) device driver manager.

     주변 장치를 일관되게 접근하도록 해줌

---

[4] 주 기억장치에 load되는 프로그램 분할의 기본 단위

[5] 프로그램을 한번에 처리할 수 있는 단위. Page 단위로 주 기억 장치에 로드하고 언로드한다.

- gcc는 GNU의 Complier다.

  Preprocessor, complier, assembler, linker 과정을 통해, 실행파일을 생성할 수 있다.

  Preprocessor는 전처리 지시자들을 해석하고, complier는 high level language를 .s 형태의 어셈블리어 형태로 전환한다. Assembler는 .s 파일을 .o 형태의 object file로 변환한다. Linker는 생성된 object file들을 .out 형태의 execute file로 만든다.

  - Symbol은 주소를 갖는 최소 단위로, 크게 RO, RW, ZI 부분으로 나뉜다. RO부분은 보통 ROM에 담기는 것으로 code(function), const variable등이 포함된다. RW는 global variable 중 초기화가 된 variable을 포함한다. ZI은 0으로 초기화가 되거나 초기화가 되지 않은 global variable이다. 그리고 static이 아닌 local variable들은 ZI(stack이나 heap)에 자리잡는다.

  - Object file의 크게 .text, .data, .bss section으로 구성되어 있다. .text는 RO, .data는 .RW, .bss는 ZI에 해당한다. Linker를 이용하여 ELF file을 만들 때, 각 object file의 section들끼리 묶여, ELF file의 segment가 된다. 다른 object file에서 참조한 변수나 코드들을 .rel.data나 .rel.text 등으로 linker될 때 relocate되어 연결된다. ELF format은 RO segment의 segment header table을 시작으로 .init, .text, .rodata로 구성되어 있고, RW segment의 .data, .bss 부분, 메모리에 load되지 않는 .symtab(symbol table), .debug, .line, .strtab, object file의 section을 설명하는 section header table로 구성되어 있다.

- Makefile

  파일간의 종속관계를 나타낸 것으로 컴파일러가 순차적으로 실행될 수 있게 해준다.

  Make를 쓰게되면 프로그램의 종속 구조를 파악하기 쉽고, 반복 작업의 최소화가 가능하다.

  Target, dependency, command, macro로 구성되며, 코드를 단순화시키고, 수정을 용이하게 하기 위해 macro를 사용한다.

CH3

DAY-3

- Program과 process

  - Process는 동작중인 program이다. 디스크에 저장되어 있는 program(파일 형태)이 kernel로부터 자원을 할당 받아 동적인 객체가 된 것이 process라고 할 수 있다.

- Process의 구조

  Process의 생김새는 가상 주소[6] 공간에서의 모양을 의미한다. 프로세스는 크게 text, data, heap, stack 부분으로 나눌 수 있다. User space에서 text(함수, instruction 등)는 제일 하위 주소 공간을 차지하고, 그 다음 data(glob var), heap, stack 순으로 차지한다. Local 변수는 stack에, dynamic var는 heap에 동적으로 할당되는데, stack는 kernel, user space의 경계면에서 아래로 커지고 heap은 위로 커진다. 각 영역을 segment 또는 vm_area_struct(가상 메모리 객체)라고 한다.

- Process의 생성

  Process는 fork()와 vfork()로 생성된다.

  - fork()

    fork()는 자식 프로세스를 생성하고, 메모리 영역을 할당하여, 부모 프로세스의 메모리 공간을 모두 복제한다. 하지만 모두 복제하는 것은 비효율적이기 때문에 최근에는 COW 방식으로 기본적으로는 메모리 공간을 참조만 하고 있다가 변경이 확인되면 복제하는 방식으로 변했다. 부모 프로세스와 자식 프로세스는 변수를 공유하지 않는다. fork()는 자식 프로세스가 생성되면, 부모 프로세스에는 자식 프로세스의 PID[7]

---

[6] 한정된 RAM의 한계를 극복하고자 disk같은 보조 저장 장치를 이용해, 디스크 공간을 메모리처럼 활용할 수 있게 해주는 기법이다. Disk에 존재하는 파일을 paging file이라고 하며, process는 가상 주소를 사용하고, 읽고 쓸 때만 MNU를 사용해 물리주소로 변환한다. Process가 RAM의 공간을 초과해, disk를 사용할 때는 물론, 성능이 저하된다. Thread는 각 process에 할당된 주소 공간에만 접근할 수 있다.

가상메모리는 kernel space와 user space로 나뉘는데 kernel space는 kernel, driver 등을 실행시키기 위한 예비 메모리고 user space는 process에 할당되는 공간이다.

[7] Process ID

를, 자식 프로세스에는 0을 반환한다. 에러 시에는 적당한 에러 값을 반환한다.

- vfork()

  fork() 함수는 주로 fork 뒤에 exec()를 실행함으로써 자식과 부모 프로세스를 동시에 실행하는 경우가 많다. 하지만 이럴 경우 불필요한 복제가 일어나기 때문에 오버헤드가 일어날 경우를 위해 vfork()를 사용한다. vfork()는 부모의 메모리 공간을 공유한다. 따라서 부모와 자식이 변수를 공유한다. 따라서 부모와 자식의 race condition을 막기 위해 자식 프로세스가 생기면 부모 프로세스는 sleep된다. vfork()뒤에 바로 exec() 계열 함수를 호출하지 않으면 자식 프로세스에서 변경한 변수가 부모 프로세스에도 적용된다는 위험성이 있다.

- exec()

  exec() 함수는 사용하던 메모리 공간을 해제하고 새로운 실행파일을 로드하고 실행한다. 즉 exec()를 호출한 프로세스는 실행이 중지되고 새로운 프로세스로 교체된다.