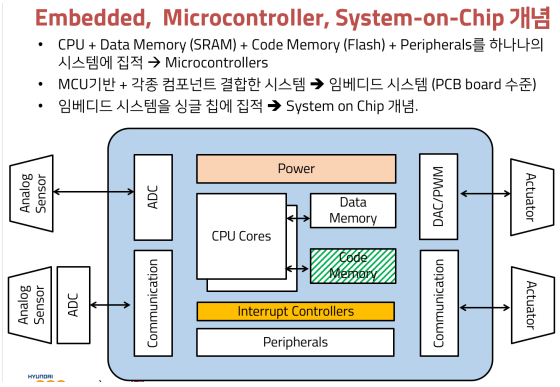
2024년09월02일 (월, 1/4일차)

- **0.** 강의 소개
  - (1) 강의 일정

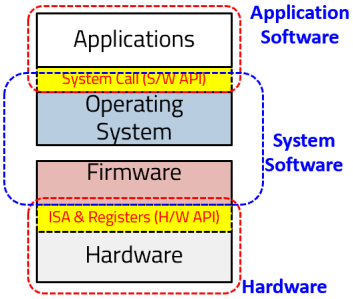
* 교육시간 : 09:00~17:00

구 분	오 전	오 후
1일차	● **임베디드 SW 개발환경 구축 (실습)** - WSL, git, Visual Studio code 설치 - Hello World 빌드 - Github기반 공동 개발 실습 ● **임베디드 C언어 관련 기초 (실습)** - 전역변수,정적변수, extern이해 및 활용 - 변수의 메모리 allocation과 포인터 - 연속된 메모리 할당 및 배열	● **임베디드 C 프로그래밍 (실습)** - Enumeration 이용한 레지스터 flag처리 - Void* 이용한 데이터 복사 속도 향상 - 함수 포인터 이용한 펌웨어 변경 - Volatile 이용한 하드웨어 레지스터 접근 - Endian 처리 방법 (big / little endian) - 구조체를 이용한 하드웨어 레지스터 표현 - 공용체 이용한 하드웨어 영역의 표현 - 하드웨어 비트 슬라이스를 C언어로 표현 - signed/unsigned extension 원리 이해
2일차	● **C Standard Library 기반 시스템 프로그래밍 (실습)** - FILE 구조체 및 파일 포인터 - 표준 입출력, 에러 채널 - fopen, fgets, getc, putc 함수 활용 - 바이너리 포맷 파일 처리 - 블록단위 파일 제어 프로그래밍 - 파일 랜덤 엑세스 및 블록 인코딩 - Buffered I/O의 필요성 및 제어	● **리눅스 시스템콜 기반 시스템 프로그래밍(실습)** - 리눅스 운영체제와 시스템콜 실행원리. - 파일제어 저수준 시스템 콜 함수 활용 - 파일 디스크립터 기반 자원 접근 제어 - 리눅스 파일시스템 구조 이해 (i-노드, 데이터 블록, 동적 i-노드 테이블) - 리눅스 커널에서 파일 시스템관리 원리 - 파일 정보/관리 수정 저수준 함수 - 디렉토리, 하드링크, 심볼릭 링크
3일차	● **리눅스 시스템콜 기반 시스템 프로그래밍(실습)** - 공유파일 영역 제어 (Lock) - 프로세스 관리 - 쉘, 실행, 종료과정과 커널동작 - 프로세스 ID, 사용자 ID, 유효사용자 ID - 프로세스 이미지 및 프로세스 생성 fork() - 부모 프로세스/자식 프로세스 연동 실행흐름 - 프로세스간 동기화 및 파일 디스크립터 연결 - 메모리 관리, 공유메모리 제어, 메모리 매핑 - 시그널 및 시그널 handler - 파이프와 소켓기반 프로세스간/호스트간 통신	● **리눅스 커널 및 하드웨어 동작이해 (실습)** - 프로세스 내부 상태 레지스터 및 실행흐름 - 함수 콜 과정과 인터럽트 처리 과정 - 멀티 프로세서 및 멀티프로세스, 멀티 쓰레드 - 디바이스 드라이버 구현 원리 - 하드웨어 제어를 위한 시스템 프로그래밍 - 가상메모리와 물리 메모리, 장치 주소 영역 - 메모리 레이아웃, 프로세스 메모리 - 하드웨어 영역 접근 제어 프로그래밍
4일차	● **리눅스 기반 하드웨어 제어 (실습)** - 라즈베리 파이 운영체제 설치 - 개발환경구축 - 디바이스 드라이버 접근위한 시스템콜 - 장치를 파이프에 연결하여 표준입출력처리 - 멀티 프로세스 기반 장치 연동처리	● **리눅스 기반 하드웨어 제어 (실습)** - 센서 입력 및 표준 입력 수신 - 엑추에이터 출력 제어 및 파이프 연결 - 소켓 기반 센서 결과 취득 및 가시화 - 하드웨어 스펙 읽고 디바이스드라이버 제어 - I2C, UART등 통신 장치 이용한 연동제어

  - (2) 시간관련.
    - 45분 수업, 15분 휴식
    - 점심시간 11:30~13시
    - 종료시간 16:20~30분 사이
    - => 대신, 9시 정각에 시작, 매시간 정각에 시작.
  - (3) 설문조사
    - 임베디드 **C** 기초 이해도 설문
      - https://forms.gle/cKRfCEsM7JqzixvG8
    - 임베디드 시스템 **SW-HW** 프로그래밍 이해도 설문
      - https://forms.gle/5hS3dR4iY1rKV8A9A

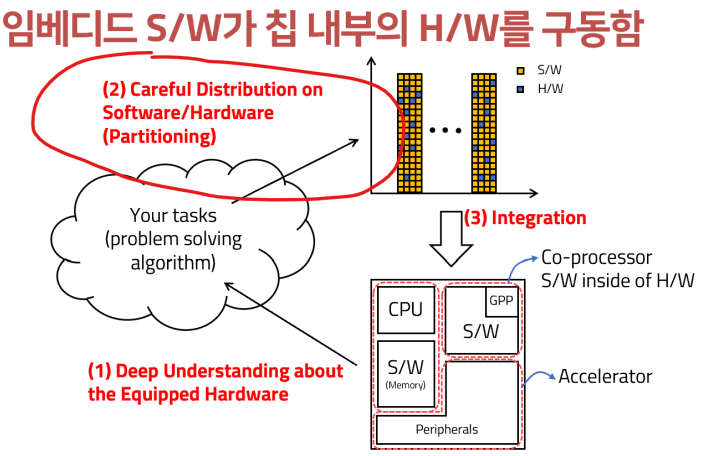
        - 임베디드 리눅스 활용정도
            - https://forms.gle/tjAsUGf2XoNHE92NA
            -
- **1.** 임베디드 시스템/소프트웨어 개론
    - (1) 임베디드란?
        - **Firmware**의 의미
            - Software가 반도체 (flash)에 한번 다운로드 (programming)되고 나면
                - 더이상 지워지지 않고 단단하게 남아 있다는 의미.
            - => 하지만, 지워질수 있음 (retention 특성)
                - 반드시 firmware. erase이슈를 대비하여 self check해주어야 함.
        - **Embedded**의 의미
            - 

            - 1. main cpu (프로세서)
            - **2.** 메모리 **Embedded (**이게 가장 중요한 **block**임)
                - 썼다 지웠다 할 수 있는 Data Memory (SRAM)
                - 한번 쓰면 안지워지는 Code Memory (Flash)
                - **=>** 크기, 스피드, 대역폭, 신뢰성 특성. **->** 시스템 특성을 좌우함.
            - 3. Digital 회로 (HW)
                - timer, gpio
                - spi, uart, i2c, can, lin
            - 4. Analog회로 (CKT)
                - power
                - ADC
                - DAC
            - => 아날로그회로, 디지털회로, 메모리를 하나로 집적 + 소프트웨어
                - => 칩 내부에 flash메모리에 적재되면, firmware가 된다.
                - => System on chip -> Software on chip/systems
        - OS는 사용자 Application과 하드웨어장치를 decoupling함.
            - 사용자 코드 개발할 당시 장치에 대한 명시적인 코드 구현이 필요없고, 코드에서 포함되어선 안된다



- ○
- ○ **OS**는 사용자에게는 **cpu**와 메모리 자원 가상화를 제공 **(**멀티 프로세스**),** 그리고 내장 하드웨어 장치 접근위한 추상화를 제공
  - ■ ***(***디바이스 드라이버를 별도로 설치함으로써 특정 칩을 지원하는 기능을 포함한 커널이 되어 기능이 확장되는 효과***.)***
  - ■ (주변장치 가상화까진 안됨, 이건 하이퍼바이져에서 구현함)

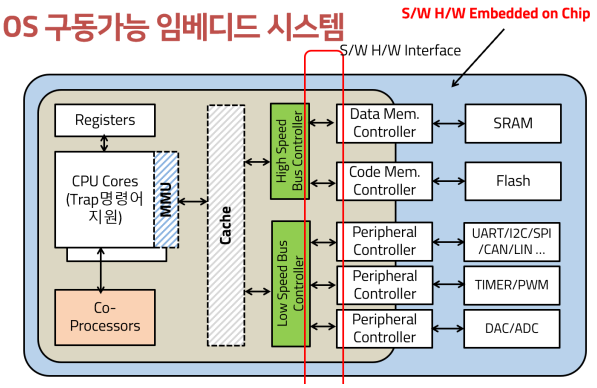
- ● 리눅스 OS로 제어시스템 구현시 커널 동작 이해 필요한 이유.
  - ○ Application이 직접 장치를 제어하는게 아니라
  - ○ ***O/S***가 대신 장치에 접근해서 제어하므로.
  - ○ 내 프로그램의 특성에 의해 전체 동작이 결정되지 않으므로
  - ○ **O/S** 커널의 동작에 대한 이해를 바탕으로 원하는 성능 특성이 나오도록 **Kernel-Aware** 사용자 코드를 잘 작성해야 함. (주로 멀티 프로세스, 멀티 쓰레드 스케쥴링 연동동작시)



System On Chip (Latest Processor Architecture)

- ○

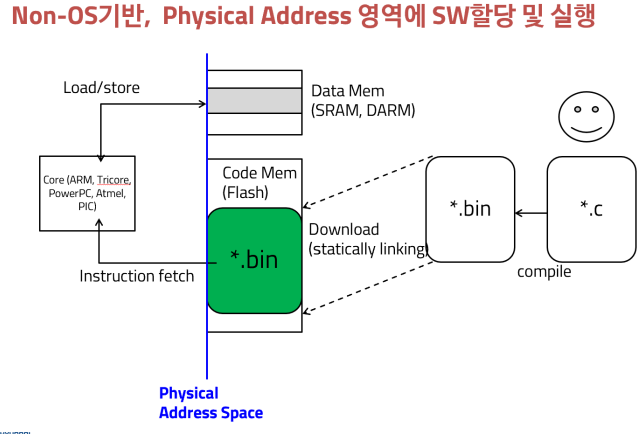
- ○ **(2)** 임베디드 코드 실행을 위한 내장 **HW support**
  - ■ 프로그램에 접근하는 방식 (주소 접근방식의 차이)에 의해 임베디드 MCU기반 제어시스템과 임베디드 OS(리눅스)기반 제어 시스템간 차이가 존재 (MMU와 캐쉬의 필요성)
  - ■ OS기반 시스템에서는.. 아래와 같다.



-
- **1.** 컴파일/링킹 단계에서 물리 영역이 아닌 가상 주소 영역을 대상으로 프로그램 이미지 생성.
  - 런타임에 프로그램을 링킹해서 메모리에 동적배치하고 그 위치를 ***O/S***에 알리면, ***OS***는 다시 ***MMU***내장하드웨어를 제어해줌.
  - 실제 프로그램을 실행할때는 물리 메모리의 주소 영역에 배치될 것이므로 가상 주소로부터 실제 그 주소로 접근하기 위한 주소 변환기 필요 ***(MMU***필요성***)***
- **2. flash**메모리 접근이 아니라, 캐쉬로 먼저 불러들이고 간접 접근 방식
  - 물리 flash memory -> ram 매핑을 캐쉬 컨트롤러가 하드웨어적으로 함. (캐쉬매핑)
  - OS구동이 되려면 flash와 cpu속도 갭을 극복해야 함 (캐쉬필요성.)
  - 캐쉬 **hit/miss**에 의해 제어 특성이 **nondeterministic**한 특성을 보임
  - **=>** 일관된 특성을 보이도록 커널과 잘 결합된 사용자 프로그램을 개발해야 함**.**

○ (3) 임베디드 SW 실행방식 차이

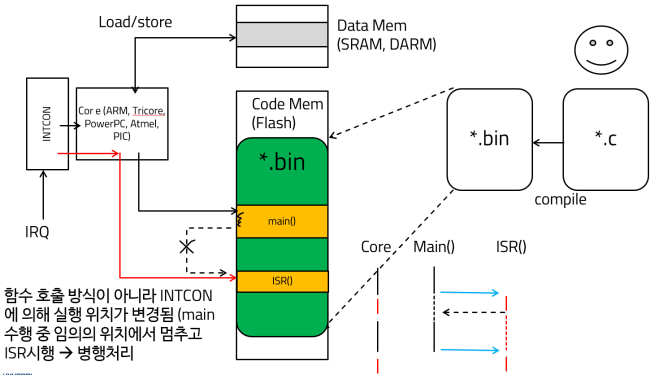
- **1. single process**
  - function call형태로 코드가 보여도 결국은 single flow 실행특성을 보이는 single process동작임.



-

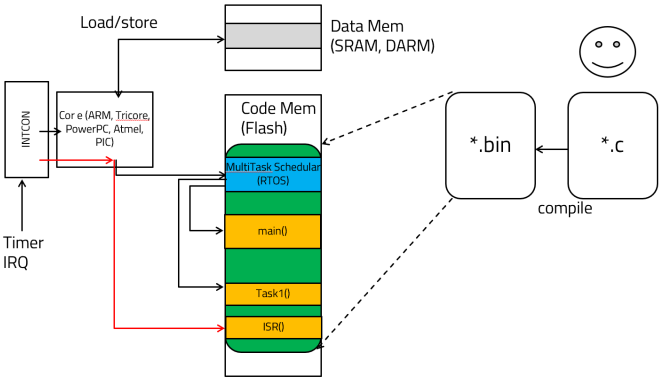
- **2. single process + HW interrupt에 의한 callback 병행처리 (이러한 함수를 ISR이라고 하고 HW Task라고도 한다)**

  Non-OS기반, Physical Address 영역에 SW할당 및 실행
  : Interrupt Controller에 의한 HW Triggered Callback실행

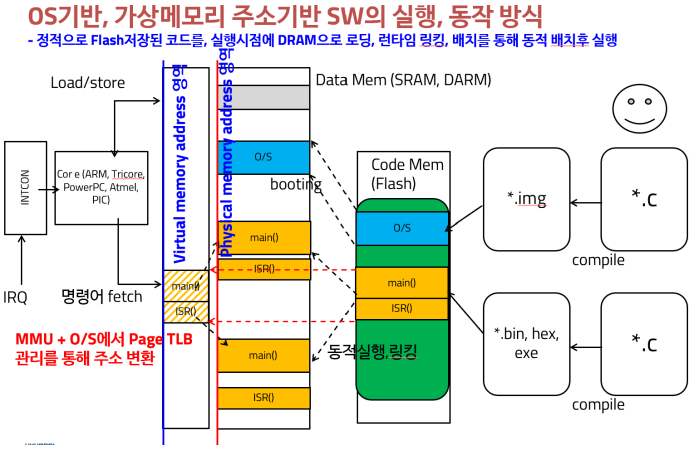
- **3. single process + HW interrupt + SW interrupt 에 의한 callback 병행처리**

  RTOS기반, Physical Address 영역에 SW할당 및 실행
  : RTOS Schedular에 의한 SW Triggered Callback실행

  - SW적으로 RTOS에서 call하는 task를 SW task라고 함.
  - RTOS 쪽에서 이야기하는 RT Task라고도 함.
  - HW ISR및 SW Task모두 병행처리 방식으로 콜백됨
    - 1. HW Task (ISR)은 interrupt controller에 의해 런타임에 분기 (우선순위, 발생시점에 따라)
      - 하드웨어적 context switching
    - 2. SW task (소위 task)는 내장 RTOS같은 소프트웨어 여러 **policy**에 따라 중요한 **task**순을 **async.**한 시점에 병행 호출함.
      - RTOS내부에서 sw기반 context switching함.

- **4. OS기반 동적 프로그램 invoke**

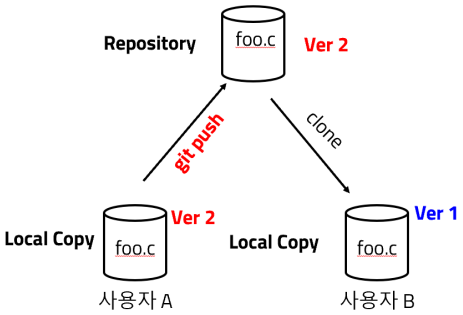


- 컴파일/링킹 할때는 어차피 어느 물리 메모리 영역에 배치될지 모르므로 정적링킹은 의미가없음
- 32비트 주소라면 4G영역을 다 쓸수 있다고 가정하고 그 영역에 컴파일러가 자동으로 배치/컴파일/링킹.
- 이후에 물리 메모리에 로딩될때 빈땅을 찾고 그영역에 배치될수 있도록 프로그램 이미지를 동적 링킹
- 링킹결과로 결정된 주소를 바탕으로 OS및 내장 MMU에 위치정보를 기록.
- cpu는 무작정 가상메모리 대상으로 컴파일된 코드 이미지를 직접 실행. -> 코드 실행, 점프 위치는 컴파일할때 그 주소를 그대로 사용
- 메모리 접근을 할때 **MMU**회로가 그주소를 가로채서, 물리 주소 영역으로 변환함.
- **MMU** 주소 번역 매핑 테이블은 유한하므로 프로세스가 많아지만 **O/S**가 지속적으로 **MMU**변환테이블을 **backup/restore**해야 함 (==> 그 유명한 page fault 현상임)

2. 리눅스 환경설치

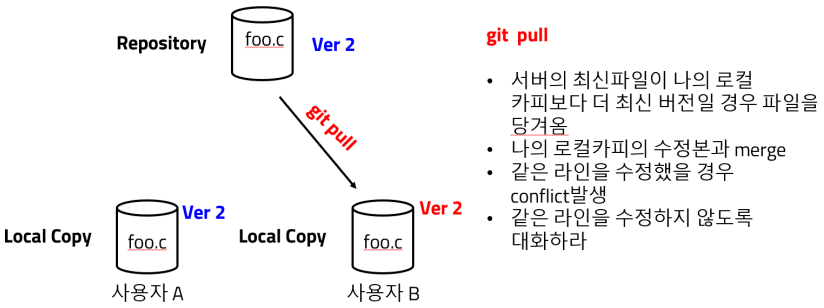
(1) 저장소 개념 (github)

- 저장소 버전 컨트롤 개념에서 가장 중요한 것.



git push

- 스테이징 영역에 있는 파일을 저장소에 최종 저장하고 버전 업데이트
- push하려면 반드시 git pull을 통해 서버의 최신버전 파일을 내 로컬 카피로 받아야 함 (서버와 동기화)
- 이때 생기는 conflic이슈는 각자 책임져야 함.



-
- 반드시, 작업하기 전에, 서버에 있는 내용으로 최신화 해야 함.
  - 다른 동료가 작업후 커밋(푸쉬)를 하면 서버는 버전이 한개 올라가고 (더 최신화 됨)
  - 내가 작업한것을 서버로 커밋 (푸시)하기 전에 다른 동료가 먼저 커밋하면, 그내용을 통합한뒤 나의 코드에 문제가 있는것은 내가 고쳐야 할 책임이 있다.
- 메일이나 알람이 오면, 반드시, 내용을 면밀히 파악하고
  - 그 코드가 현재 내가 작업중인 코드와 conflict나는지 여부를 파악해야 한다.
  - 서버 -> 내 컴퓨터 (로컬)로 **update** (또는 **pull**) 한 뒤 내가 작업중인 코드와 결합해서 동작해보면서 **side effect**가 없는지 파악해야 한다.

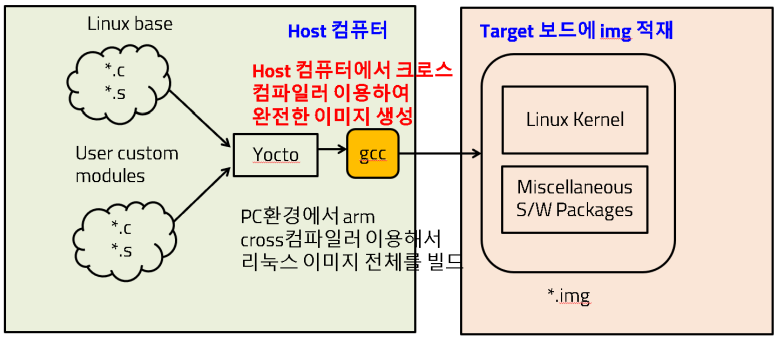
- 차량별, 업체별(협력) 통합 개발시 처리 프로세스 정리
  - **1.** 파일 분리
  - **2.** 파일 내부 함수 분리
  - **3.** 컴파일된 이미지 링킹 단계 분리.
    - **=>** 분리된 로직들이 버젼컨트롤 시스템에서 유연하게 통합되도록 소프트웨어 아키텍처 설계가 필요함.
- 깃허브 가입 및 이메일 기록
  - https://docs.google.com/spreadsheets/d/1ywRwsH2G2KSX2H4MCebp0KUMAasIuj91QCndgxgEPSg/edit?usp=sharing
- 깃허브 저장소 링크
  - https://github.com/AI-SoC/EMLINUX_20240902

(2) 리눅스 설치 및 환경설정

- OS와 사용자 애플리케이션을 포함한 이미지 생성
- **1. Yocto**
  - **OS**와 애플리케이션을 크로스 컴파일러를 이용하여 완전한 빌드, 단독 실행 코드 이미지 생성하기 **(MCU**에서 **F/W**개발과 비슷함**)**
    - 아직 타겟보드에서 실행가능한 네이트브 컴파일러를 포팅하지 못한 단계
    - 필요한 패키지만 선별적으로 골라서 OS 에 탑재하고 싶을때.

커널 및 필요한 패키지 + 사용자 앱을 하나로 빌드

타겟 보드에서 실행가능한 native 컴파일러가 없으면,
이런 방식으로 반드시 구동해야 한다.

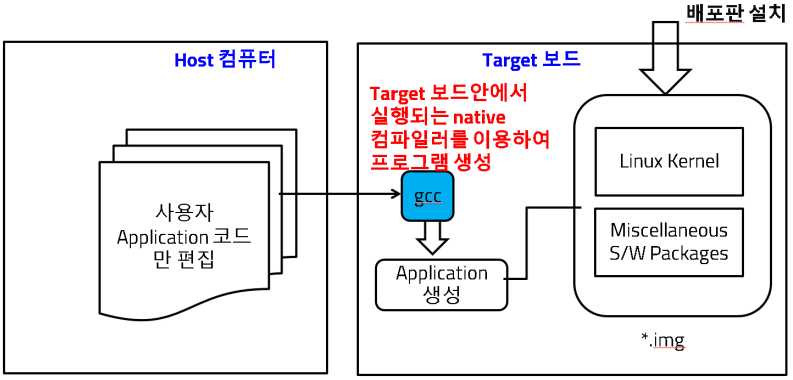


-

■ 2. 배포판

- OS 이미지 생성과, 사용자 애플리케이션 개발을분리
  - 타겟 보드에서 실행가능한 네이티브 컴파일러 포팅이 선행되어야 함.
  - 이미 준비된 **OS** 배포판을 그대로 사용하므로 바로 사용가능하나 불필요한 패키지 많아서 무거움

이미 만들어진 OS배포판 안에서 직접 사용자코드를 빌드

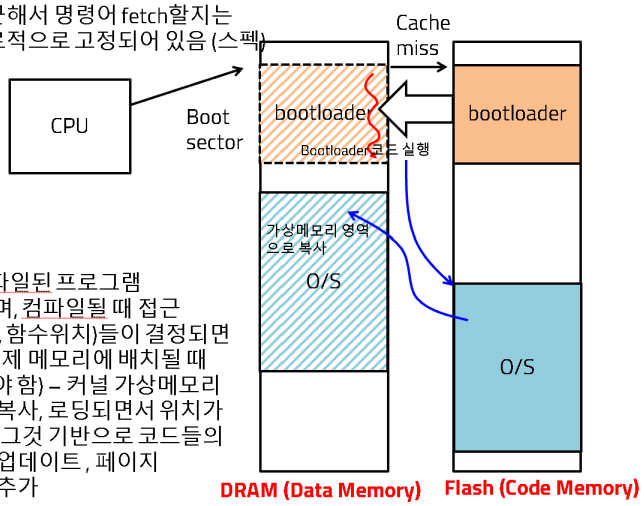


- OS 이미지 생성과 애플리케이션 개발을 분리

○ (3) 리눅스 부팅 및 서버 접속

■ 1. OS를 메모리에 적재 (부팅)

시스템 Reset시 최초 어느 주소에 접근해서 명령어 fetch할지는 회로적으로 고정되어 있음 (스펙)

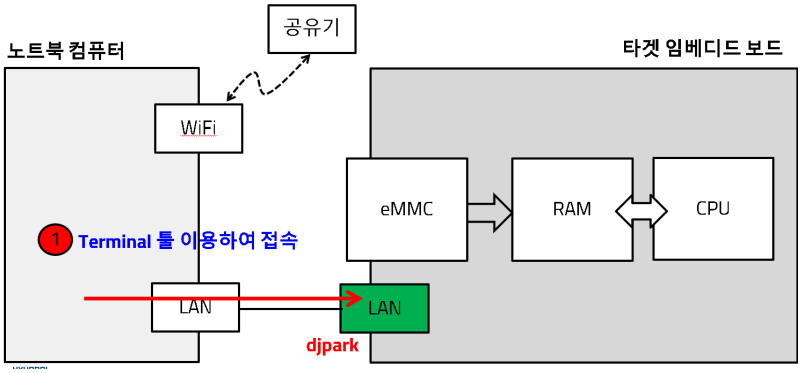


- O/S도 컴파일된 프로그램 이미지이며, 컴파일될 때 접근 주소(변수, 함수위치)들이 결정되면 안된다 (실제 메모리에 배치될 때 결정되어야 함) – 커널 가상메모리
- 메모리로 복사, 로딩되면서 위치가 결정되고, 그것 기반으로 코드들의 주소들이 업데이트, 페이지 테이블도 추가

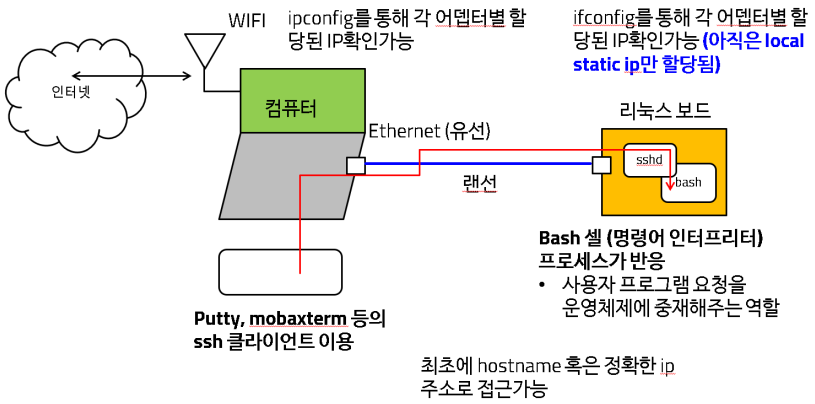
-

■ **2.** 리눅스 보드에 터미널을 이용한 접근

- 이미 eMMC. 메모리에 LinuxOS탑재되어 있음.
  - **192.168.7.2**
  - ID : **debian**
  - Password: **temppwd**



- 타겟보드에 ssh 데몬이 떠 있고 이후에 내부 쉘 프로그램을 구동하여 OS와 인터페이싱 한다.

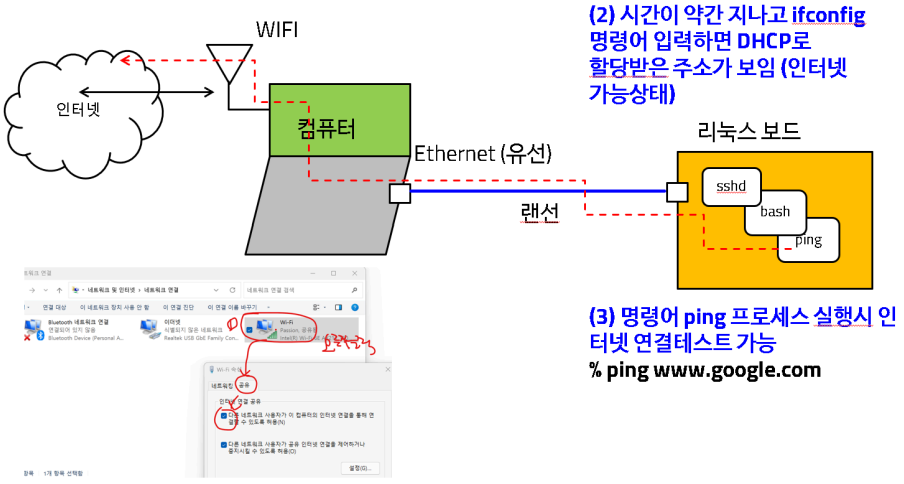


○ (4) 리눅스 보드 인터넷 접속 설정

■ PC의 인터넷 기능을 이용하여 외부 접속하도록 설정

- 즉 PC를 유선 공유기로 만들기.

(1) 인터넷이 되는 어뎁터에 공유기능을 활성화하기.

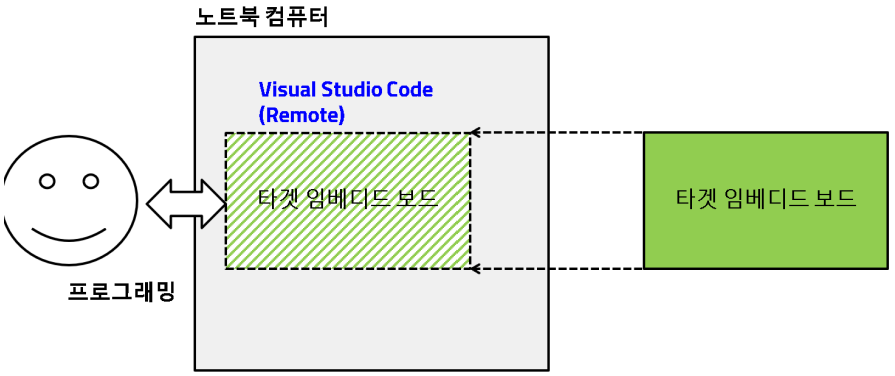


■

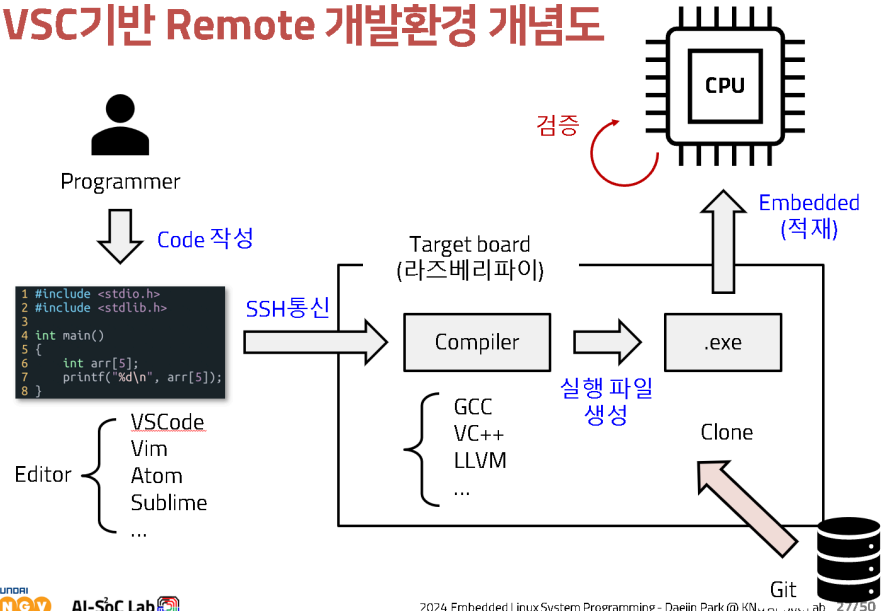
○ (5) 원격 개발환경 구축

■ 편집한 코드를 sftp등으로 타겟보드로 일일이 옮길 필요없게함.

■ 마치 타겟보드의 자원 (파일시스템)이 로컬 컴퓨터에 매핑되는 효과



■

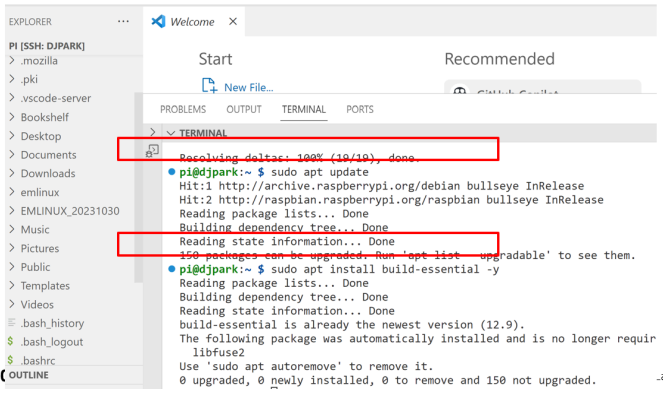


■

○ (6) 저장소 clone (예제 다운)

Git 에 업로드 되어 있는 Embedded 리눅스 C 교육 코드 자료 clone

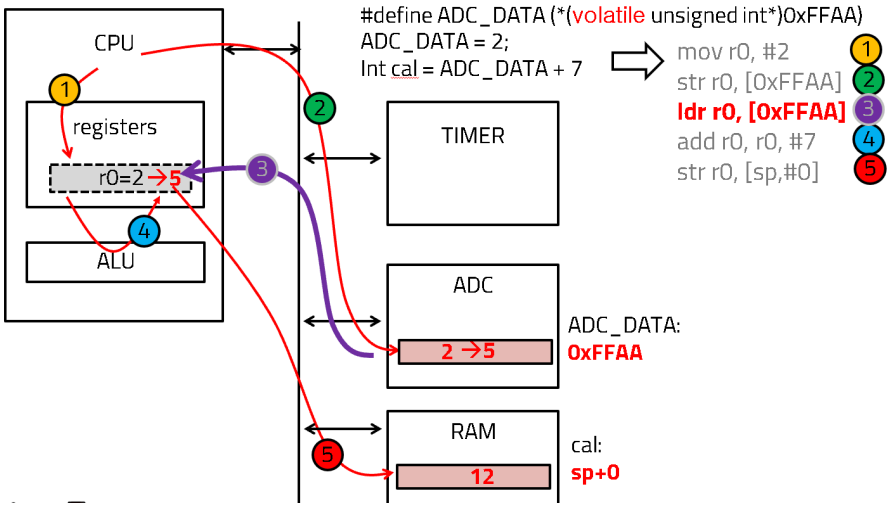
- $ sudo apt update
- $ sudo apt install build-essential -y
- $ git clone https://github.com/AI-SoC/EMLINUX_20240902
  - Visual studio code 내부 shell에서 실행하면 자동으로 브라우저와 연동되어 로그인 처리가 가능함 (mobaxterm에서 진행시 ssh 인증처리 해야 해서 복잡함)



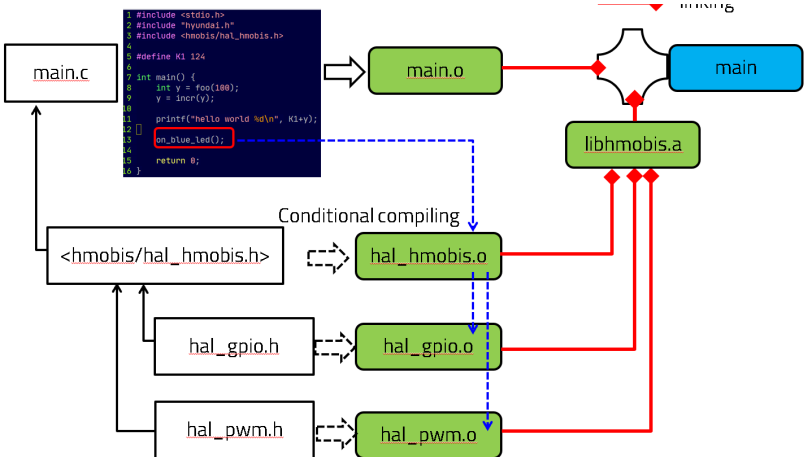
■

○

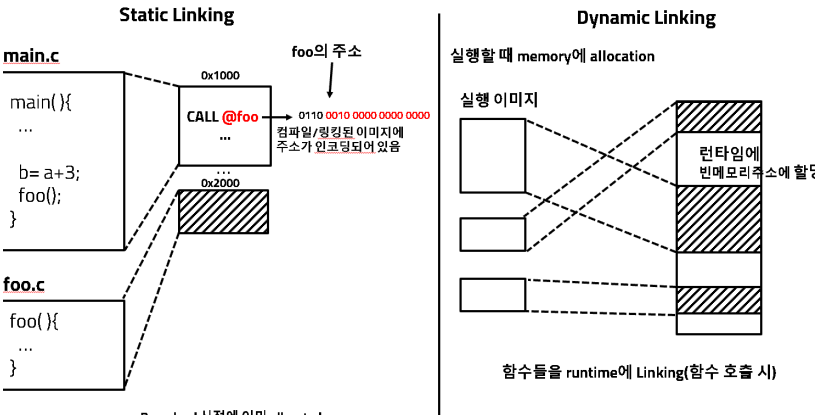
- **3. 프로그램 빌드**
  - **(1) volatile**
    - 특정 메모리 영역에 대해서 명시적으로 memory I/O를 하도록 강제하는것.
    - 어떤 값이 있을 것이라고 예상되더라도, 불필요해보이지만 메모리 주소에 매핑된 하드웨어 장치에 직접 접근해서 읽어오도록 함
    - **=> SW에 의해 어떤 값을 overwrite하지 않더라도 volatile하게 스스로 값이 바뀔 가능성이 있으므로, 직접 읽어보기 전까지는 어떤 값이라고 단정지을 수 없다. 따라서 반드시 해당 메모리 영역에 직접 접근해서 읽는 코드가 생성된다.**



  - **(2) static vs dynamic linking기반 image생성**
    - **1. static 실행 이미지 생성**
      - 모든 서브 루틴 함수를 single standalone 프로그램 이미지에 모두 적재.
      - 컴파일 링킹되고나면, 더이상 하위 코드 object,lib 필요없음
      - 컴파일/링킹단계에서 결정된 주소공간에 프로그램 이미지가 반드시 적재되어야 함



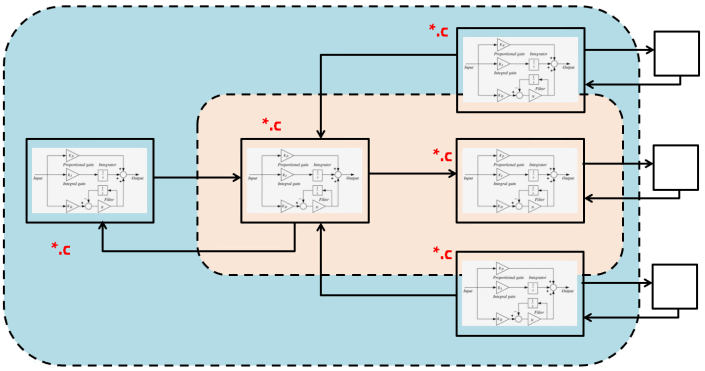
    - **2. dynamic 동적 적재 실행 이미지 생성**
      - 컴파일/링킹할때는 하위함수 라이브러리 불필요함
      - 프로그램이 실행될때, 동적라이브러로 구현된 파일이 런타임 링킹되어 메모리에 적재됨.



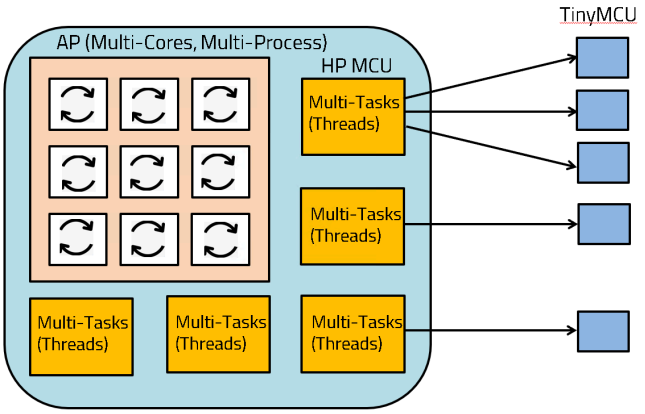
-
- 동적 링킹 원리를 잘 활용하면 런타임에 다양한 기능으로 확장,변경이 용이해짐 (OTA에서)
-

2024년09월03일 (화, 2/4일 차)

- **1. OS 기반 SW실행원리**
  - (1) 분산제어
    - 각 엣지 장치에 독립적인 F/W를 탑재
    - 엣지간 연동을 하는 코드가 각 디바이스에 상호 탑재됨.
      - 하나의 엣지가 추가되면 상호간 고려를 어느정도 해야 함.
    - inner control loop과 계층적으로 중첩되면서, 복잡도가 올라갈수도 있음.

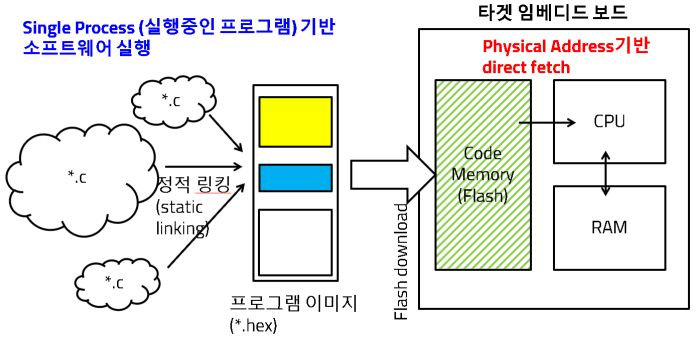


  - (2) 중앙제어
    - big & little구조로 비대칭으로 아키텍처를 구성
    - 제어 알고리즘, 신호처리등, 순수 로직과, 로레벨 제어 장치를 분리함.
      - 업데이트가 필요한 것과, 거의 변화가 필요없는 통신/구동정도만 하는것으로 분리.
    - 멀티 테스크/멀티 프로세스 관리 측면의 코드를 순수한 C알고리즘으로 분리해서 제어가능.



- ○ (3) 가상메모리 기반 SW 이미지 생성 및 실행
  - 엣지 제어 장치의 처리 **task**가 유한하다면 아래처럼 정적 링킹해서 **SW**실행해도 무방함.
  - 해당 서브루틴이 가끔 실행한다면 포함된 코드들은 사용이 거의되지 않고 메모리 땅만 차지함.

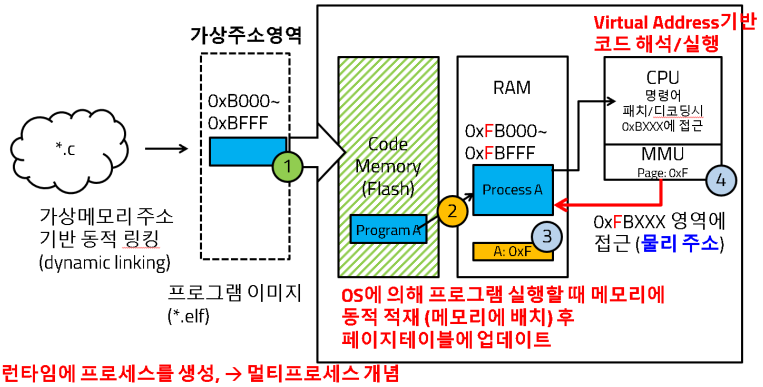
Non-OS (RTOS포함) 기반 프로그램 설치 및 실행



컴파일타임에 함수(테스크 or ISR)를 정적으로 결정하고 코딩, 매핑..
- 처음부터 테스크별 시간을 기준으로 나누어서 cpu를 사용해야 함.
- 처음부터 메모리 세그먼트를 나누어서 memory,를 사용해야 함.

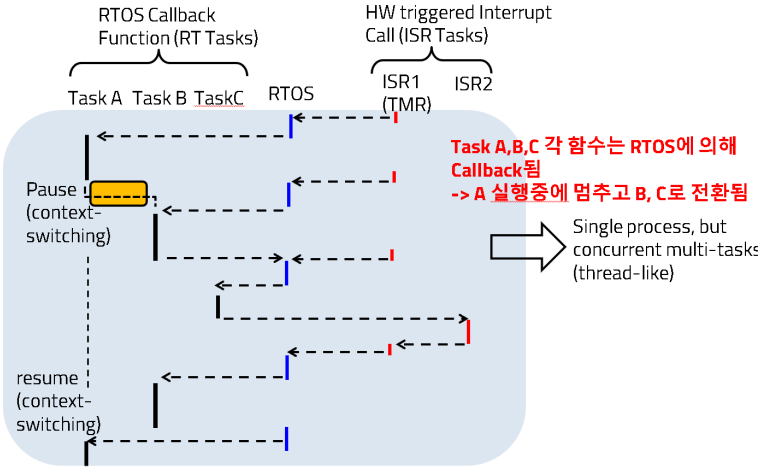
- 런타임에

OS (Non-RTOS) 기반 프로그램 설치 및 실행



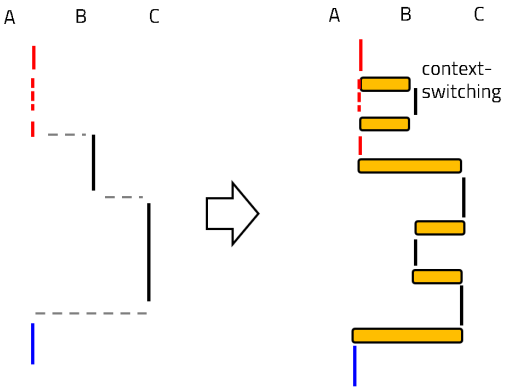
런타임에 프로세스를 생성, → 멀티프로세스 개념
- Cpu 가상화 (멀티프로세스)
  →하나의 프로세스 실행시 cpu를 독점하는것럼 착각하게.
- 메모리 가상화 (가상메모리)
  → 하나의 프로세스를 실행할때 메모리를 독점, 및 무한대 (4GB) 영역을 모두 쓸 수 있도록 함.

- (4) single process + multi-tasks



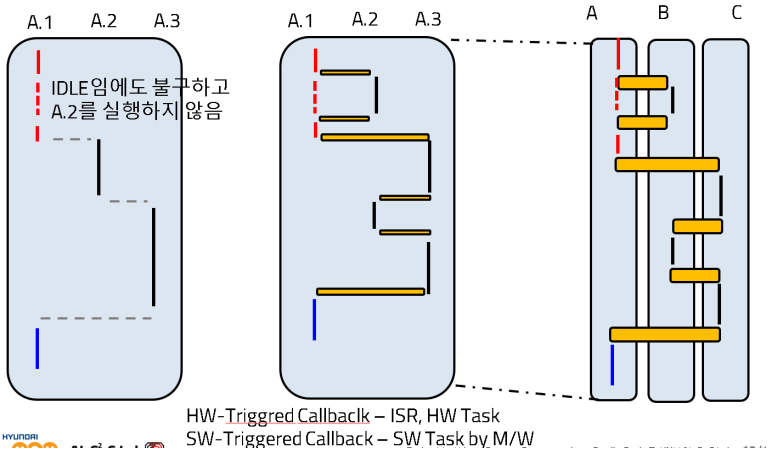
- (5) multi-process가 효과가 있는 이유

싱글 프로세스 vs 멀티프로세스: IDLE 효과



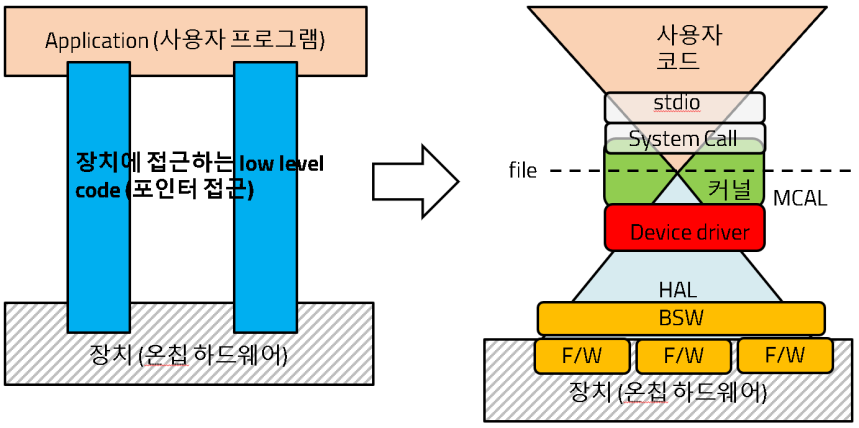
- (6) multi-process + multi-threads(tasks)

싱글 프로세스 (싱글테스크, 멀티테스크/쓰레드) vs 멀티 프로세스



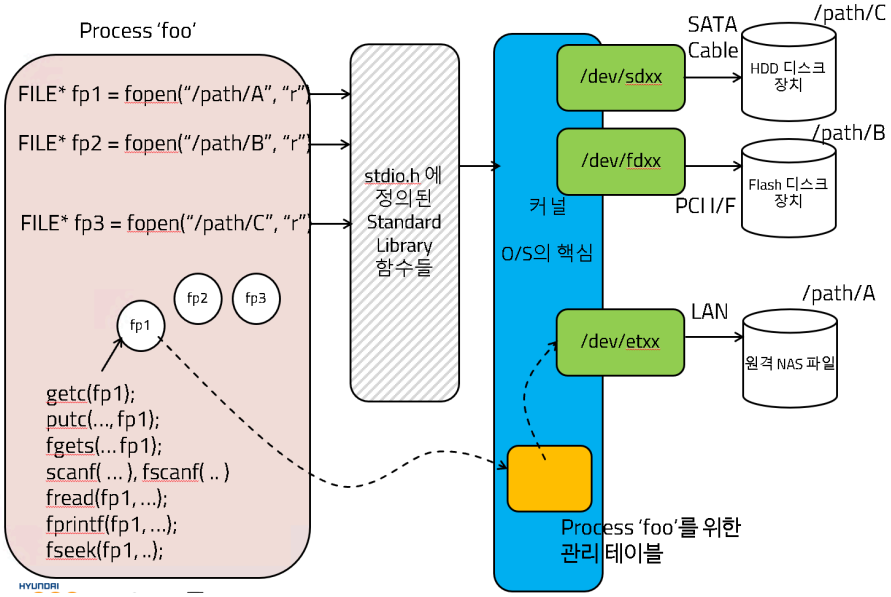
- **2. standard library**
  - **(1)** 장치를 파일로 추상화
    - 파일을 대상으로 모든 알고리즘을 구현하도록 함으로써
    - 장치에 의존적이지 않는 코드를 구현하도록 유도함.

리눅스에서 모든 장치를 file로 추상화하여 접근.



- ○ (2) 각각 실행중인 프로세스와 장치를 파일 인터페이스로 분리
  - 추상적인 파일을 대상으로 각 프로세스는 자기의 일들을 수행
    - 예를 들어, 파일을 읽고 쓰는 행위에 의한 접근 위치를 각 프로세스마다 독립적으로 가지고 있을 수 있다.
  - 처리대상이 되는 장치를 파일로 투명하게 바라볼수 있으므로, 그 장치가 실제어 어디에 매핑되어 있는지는 사용자 코드에서 고려할 필요없다.
    - 예) 파일이 HDD몇번에 있는지 물리 주소를 고려하여 읽는 코드가 사용자 수준 코드에는 없음
    - 예) 읽고 쓰는 대상이 HDD이던지, UART, I2C, CAN이던지 상관없이 읽고 쓰는 행위를 하는 코드는 똑같다.

모든 장치(파일디스크포함)를 파일기반으로 접근하는 원리.



- ○ (3) 볼록 단위로 파일 접근, 바이너리 데이타로 관리
  - txt로 ascii로 관리하지 않고, 바이너리로 블록단위로 관리하면 편리한 이유
    - 각 필드마다 고정된 구획을 나누어서 데이타를 읽고/쓰므로.

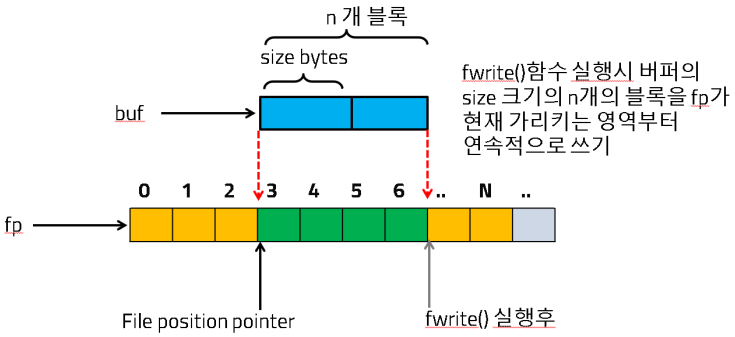
- 특정 위치에 값을 교체 삭제, 추가 등이 용이하다

■ 만약 텍스토로 값을 쓰고 읽는다면

- 데이타간 경계를 파싱해서 찾고 교체할려고 하면 교체 값의 크기에 따라 다른 영역의 데이타까지 다시 rewrite해야한다.

블록 단위로 파일의 특정 영역에 접근 (쓰기)

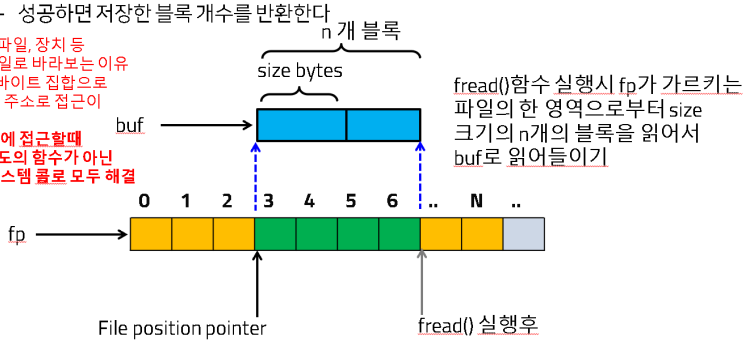
- **int fwrite(const void* buf, int size, int n, FILE* fp);**
  - 파일 포인터 fp가 가르키는 파일의 한 영역에 buf에 저장된 size 크기의 블록을 n개 저장
  - 성공하면 저장한 블록 개수를 반환한다



■

- **int fread(void* buf, int size, int n, FILE* fp);**
  - 파일 포인터 fp가 가르키는 파일의 한 영역에 저장된 size 크기의 블록을 n개 바이너리 데이터를 읽어서 buf.가 가르키는 곳에 쓴다
  - 성공하면 저장한 블록 개수를 반환한다

리눅스에서 파일, 장치 등 모든것을 파일로 바라보는 이유 – 연속적인 바이트 집합으로 볼수 있으며, 주소로 접근이 가능해짐 → 특정 장치에 접근할때 장치마다 별도의 함수가 아닌 파일 접근 시스템 콜로 모두 해결 가능함



■

○ **(4) 장치 접근 (R/W)에서 속도 차이에 대한 고려 불필요**

■ 장치에서 읽을때 프로세스 실행 빈도/주기가 길어서, 자주 cpu타임을 할당받지 못하면, 장치에서 읽어서 버퍼에 쌓이는속도가 빠르다면 ??

- 자동으로 O/S callback에서 지연을 시켜주어서 버퍼 오버런이 방지됨
- MCU F/W코딩할때는 항상 버퍼 오버런/언더런 방지를 위해 flag채크를 해야 하는데, 임베디드 리눅스에선 불필요함.

■ 장치로 데이타를 쓸때도 마찬가지

- 완전하게 버퍼에 채워진 값이 장치로 나가기 전까지는 사용자 영역으로 콜백이 되지 않음.' 자연스럽게 싱크가 맞아짐.

```c
#include <stdio.h>
#define START_ID 1000

struct student {
    int id;
    char name[20];
    int score;
};

int main(int argc, char* argv[])
{
   struct student record;
   FILE *fp;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s [file name]\n", argv[0]);
      return 1;
   }

   if ((fp = fopen(argv[1], "rb")) == NULL ) {
      fprintf(stderr, "File read error\n");
      return 2;
   }

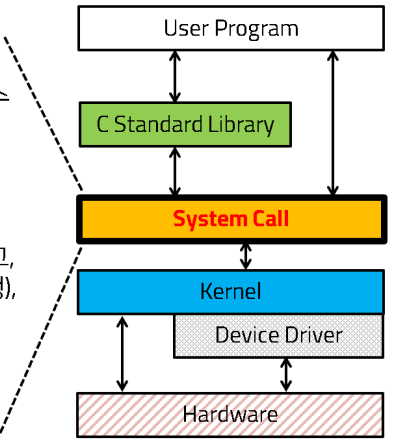
   printf("%-4s %-3s %-4s\n", "ID",  "Name",  "Score");

   while (fread(&record, sizeof(record), 1, fp) > 0)
      if (record.id != 0)
         printf("%10d %6s %6d\n", record.id, record.name, record.score);

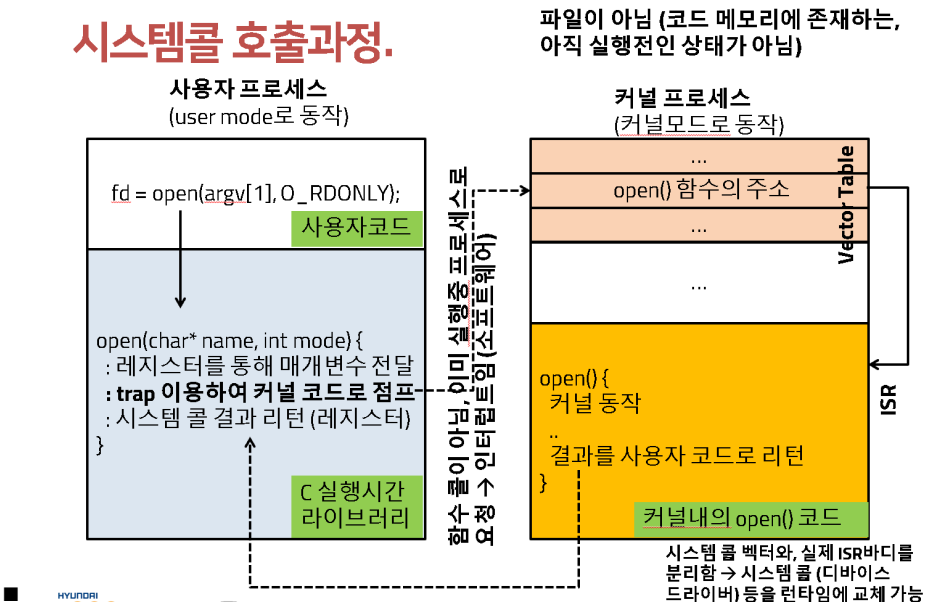
   fclose(fp);
   return 0;
}
```

- **3. system call을 이용한 OS를 통한 HW접근**
  - **(1) 반드시 system call을 통한, OS내부에 정의된 콜백을 통해서만 내장 하드웨어제 접근 가능함**

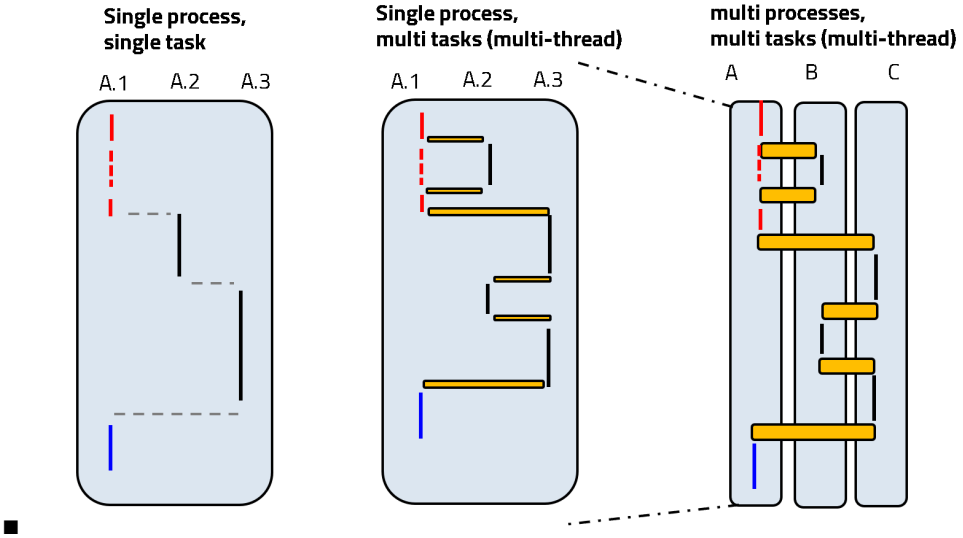
- 시스템호출 (System Call)은 운영체제가 제공하는 프로그래밍 인터페이스
- 시스템 콜을 통해 커널에 서비스를 요청할 수 있음 → 응용프로그램과 커널 사이의 인터페이스
- 사용자 프로그램이 하드웨어에 직접 접근할 수 없고, 시스템 콜을 통해 커널에 요청하여 각종 서비스 (열기, 읽기, 쓰기 등)을 요청하고, 수행은 커널이 하게 되며(커널모드에서 실행), 결과를 돌려 받아 나머지 수행 (유저모드).
- C 표준 라이브러리를 이용하더라도 결국 내부적으로 시스템 콜을 통해서 하드웨어에 접근한다.



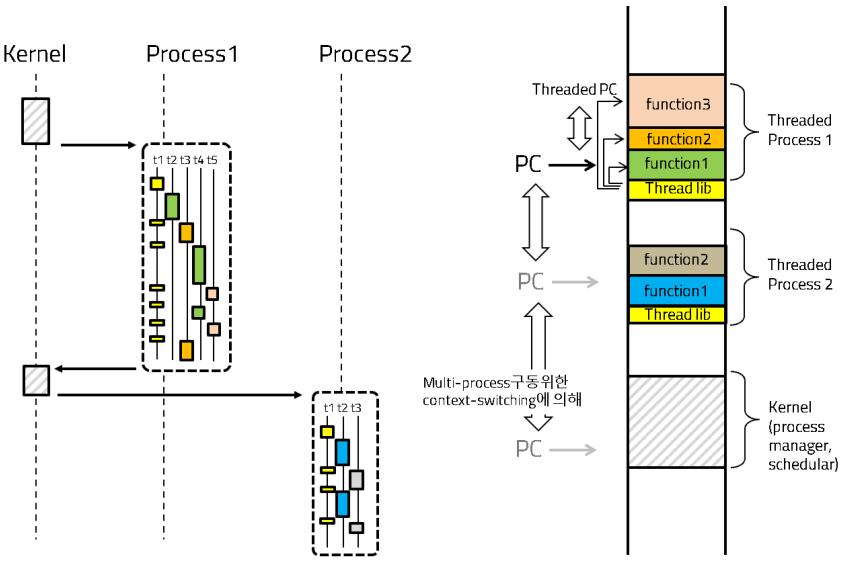
  - **(2) 실행중인 커널에 어떠한 요청을 하는 핵심원리**
    - **1. 온칩 레지스터/스택을 경유한 argument passing**
    - **2. trap을 통한 SW인터럽트 발생**
    - **3. sw 인터럽트 백터 개념의 콜백.**



- **4. concurrent SW 실행원리 (멀티 프로세스, 멀티 쓰레드)**
  - **(1) SW 실행흐름 차이**

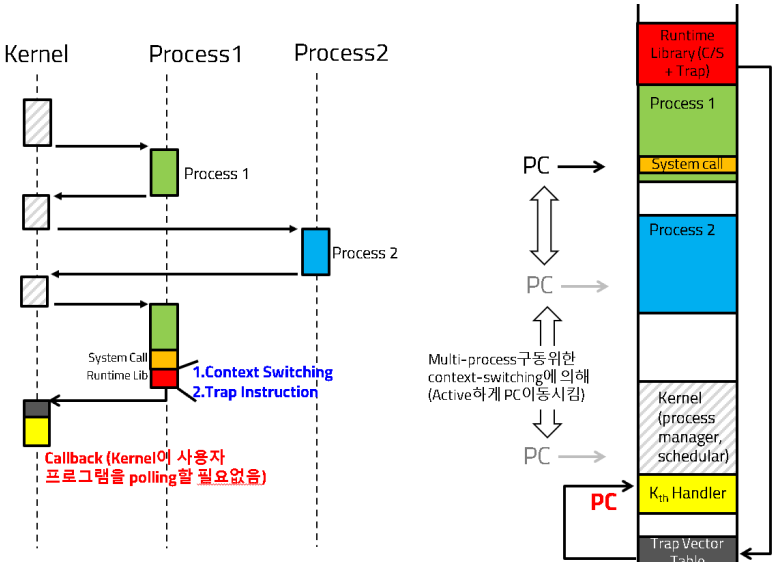


  - **(2) PC (program counter)의 이동을 통한 병행 SW실행**
    - **macro**하게 보면 **process**간 실행 위치가 이동
    - **micro**하게 보면 **thread**간 실행 위치 이동함



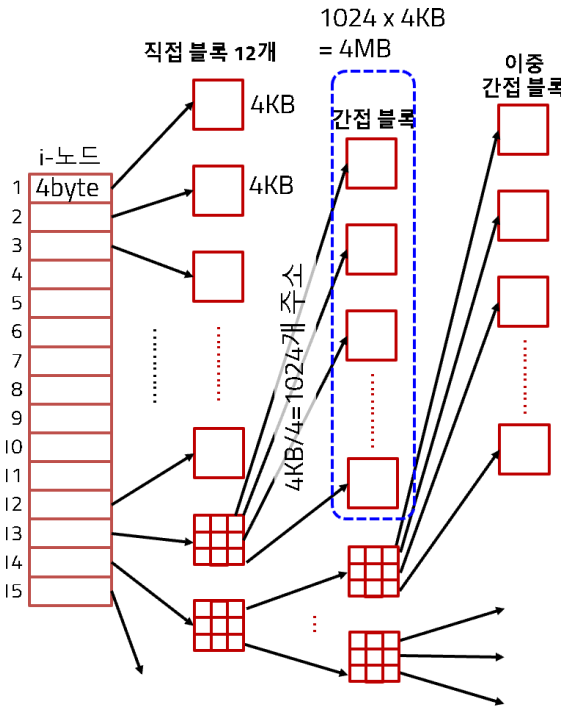
- ■

- ○ **(3)** 어떻게 실행중인 프로그램 내부에 특정함수를 호출할 수 있을까 **??**
  - ■ **SW** 인터럽트 발생시
  - ■ **CPU**는 약속된 위치에 있는 값 **(**콜백될 함수의 주소가 부팅할때 저장되어 있음**)**을 읽어서
  - ■ **PC**에 **overwrite**함으로써 자연스럽게 커널내부의 특정함수로 점프함 (시스템 콜이 되는 원리)

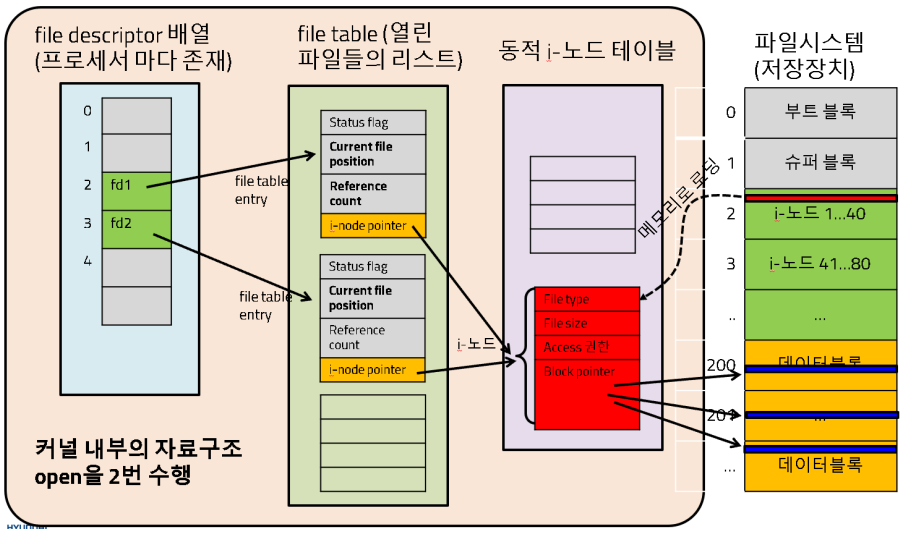


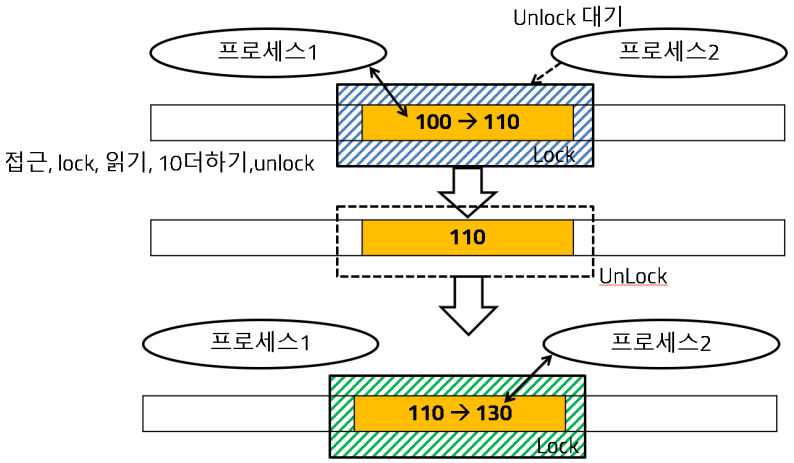
- ■

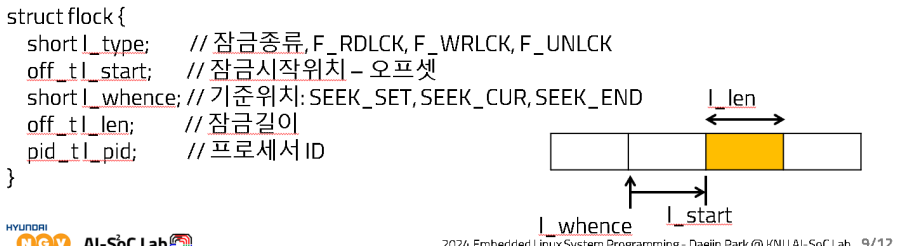
- **5.** 파일시스템
  - ○ **(1)** 파일시스템의 필요성
    - ■ 파일 크기는 서로 다르고, 삭제, 삽입, 심지어 크기가 변하는 환경에서
    - ■ 주어진 연속적인 저장공간의 집합을 효율적으로 사용할려면,
    - ■ 포인터의 자료구조 집합으로 구현해야 함.



- **(2)** 저장된 파일과 그것을 사용하는 **process**를 독립적으로 관리
  - 여러개의 **process**도 동시에 특정 파일에 접근할 수 있도록. 파일을 **open**하면 그정보를 커널 내부에서 로딩.
  - 접근 위치도 각 프로세스마다 따로따로 관리함.



- **6.** 파일(모든 장치)에 대한 공유영역 관리.
  - **(1) race condition** 및 **lock**의 필요성.
    - 

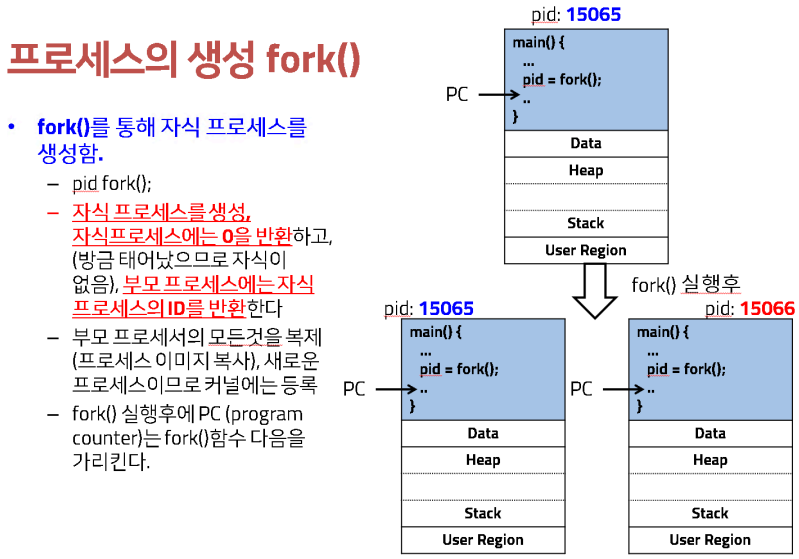
  - **(2)** 파일뿐만 아니라 모든 장치에 대해 일관적인 공유문제 처리가능
    - 앞으로 내장 하드웨어 장치도 모두 디바이스 드라이버를 구현해서 파일로 추상화하므로
    - 장치에 접근하는 다양한 프로세스들의 동시 접근에 의한 공유문제도 동일한 **lock** 시스템 콜을 통해 관리가 가능해짐.
    - 파일 전체(장치전체) 또는 장치의 일부영역만 **lock**할수 있음
    - 

  - **(3) lock.**영역에 접근시 **wait polling** 하지 않음.
    - ```
8 int main(int argc, char **argv)
9 {
10   int fd;
11
12   fd = open(argv[1], O_WRONLY | O_CREAT, 0600);
13   if (flock(fd, LOCK_EX) != 0) {
14   printf("flock error\n");
15   exit(0);
16   }
17
```
    - **1.** 어떤 프로세스가 **open** 시스템 콜을 통해 장치에 접근했는데 **lock**이 되어 있다면,
    - **2.** 커널은 해당 프로세스를 별도 테이블에 기록함. (먼저 온 순서대로)
      - 프로세스는, lock풀릴때까지 polling하지 않음,sleep상태로 감.
      - 즉 커널은 **lock** 대기중인 프로세스를 라운드로빈 방식으로 번갈아가면서 **cpu**타임을 할당할 대상에서 빼버림.
```

- **3.** 해당영역이 **lock**이 풀리면 그때 커널이 확인후, 가장 먼저 이 영역에 접근하기로 예약한 프로세스를 **wake up**시켜줌
  - 깨어난 프로세스는. 즉각 해당 파일을 lock을 해서 소유권 취득함.

2024년09월04일 (수, 3/4일차)

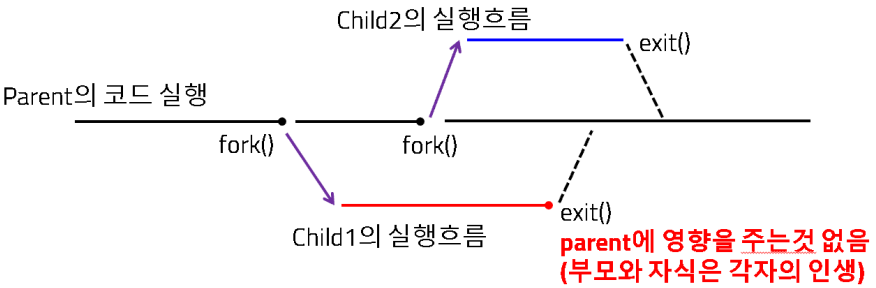
- **1.** 프로세스(자식) 생성 .
  - **(1) fork()** 시스템콜을 통한 프로세스 생성
    - 부모 프로세스의 메모리 공간의 값(상태값)을 모두 자식으로 복제함.
    - 따라서 자식 프로세스의 실행위치도 fork()다음이 됨, fork()다음 줄부터 프로그램이 시작됨 (fork이전까지 흘러오면서 생긴 메모리 값도 그대로 유지)



    - fork() 함수는 자식 프로세스의 ID를 반환함.
      - 따라서 부모 프로세스는 자식 프로세스 ID를 리턴받는 반면
      - 자식프로세스는 태어나자마자 내 자식이 없으므로 당연히 ID값은 0이 됨.
    - 그래서 아래처럼 코딩하면. 내가 자식인지 부모인지 파악할 수 있음.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    int pid1, pid2;

    pid1 = fork();
    if (pid1 == 0) {
        printf("[Child 1] : Hello, world ! pid=%d\n",getpid());
        exit(0);
    }

    pid2 = fork();
    if (pid2 == 0) {
        printf("[Child 2] : Hello, world ! pid=%d\n",getpid());
        exit(0);
    }
    printf("[PARENT] : Hello, world ! pid=%d\n",getpid());
}
```

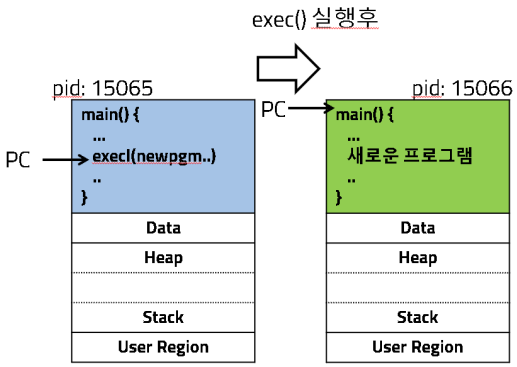
- (2) 부모 프로세스와 자식프로세스는 **concurrent**하게 코드 진행됨



parent도 Child와 상관없이 자신의 코드를 실행함
child도 부모와 상관없이 자신의 코드를 실행함

- **2.** 프로세스(새로운) 생성 .
  - **(1)** 자신을 새로운 프로세스로 대체함.

exec()로 새로운 프로세스 생성시



부모 프로세스는 종료하고 새로운 명령어가 새롭게 시작된다.

부모 프로세스를 유지할려면, fork()를 통해 자식 프로세스를 만들고, 그 안에서 exec를 호출하면된다

```
if( (pid=fork() == 0 ) {
  exec( arguments) ;
  exit(1); // 여기 수행되면 에러
}
```

- **(2) fork()와 exec()를 결합해서 프로세스 생성.**
  - **fork()**를 통해 자식 프로세스를 위한 메모리 공간, 프로세스 리스트를 생성하고
  - 부모의 인생이 아닌 자신만의 인생을 살아가게 해줌.
  - 부모는 여전히 자신의 인생을 살아감.
  - ```c
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

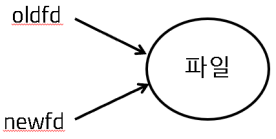
    int main( )
    {
        printf("Parent process start\n");
        if (fork( ) == 0) {
            execl("/bin/echo", "echo", "hello", NULL);
            fprintf(stderr,"first exec failed");
            exit(1);
        }

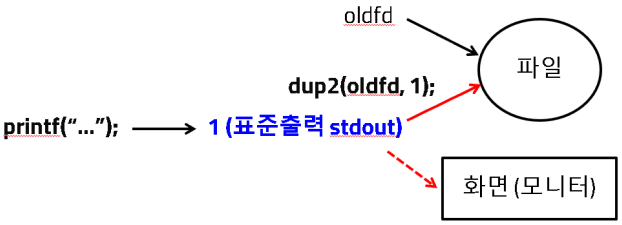
        if (fork( ) == 0) {
            execl("/bin/date", "date", NULL);
            fprintf(stderr,"second exec failed");
            exit(2);
        }

        if (fork( ) == 0) {
            execl("/bin/ls","ls", "-l", NULL);
            fprintf(stderr,"third exec failed");
            exit(3);
        }
        printf("Parent process end\n");
    }
    ```
  -
- **(3)** 프로세스의 실행결과를 다른 프로세스로 전달.
  - 장치를 파일로 바라보는 것처럼
  - 입출력(데이타)도 파일 스트림으로 추상화한다.
  - 따라서 프로그램의 실행결과를 **printf**등으로 출력하고
    - 이때 출력이 어떤 대상으로 전달할지를 고려해서 프로그램에 녹이지 않음
  - 실제 printf로 출력된 데이타가 어디로 전달될지를 바깥에서 파일 스트림 **redirect**로 조정.
    ```

파일 디스크립터 복제 함수

- **int dup(int oldfd);**
  - oldfd의 복제본인 새로운 디스크립터를 생성하여 반환
  - **newfd = dup(oldfd)**
- **int dup2(int oldfd, int newfd);**
  - 기존의 파일 디스크립터가 **oldfd**를 가리키도록 복제함.
  - **dup2(oldfd, 1);** 이제 표준출력도 oldfd를 가리키게 됨
  - 이제 **printf**등을 통해 표준출력으로 나가는 모든 데이터가 파일에 전달됨



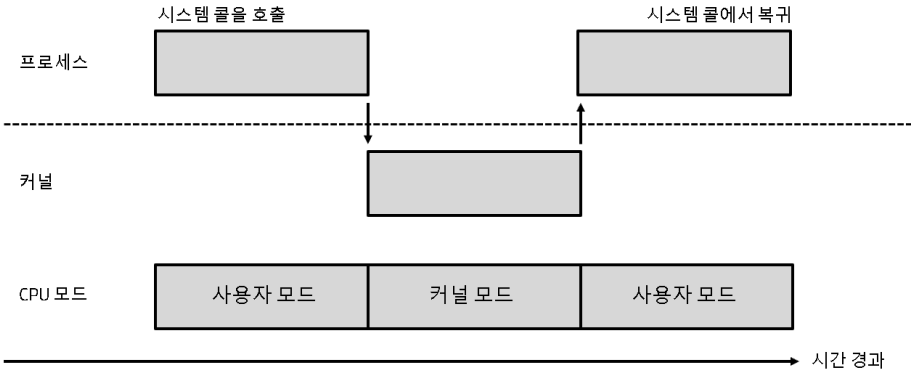


- fd 대상은 화면, uart, 장치등 다양하게 변경가능.

● 3. 멀티 프로세스를 동시에 실행시키는 원리.

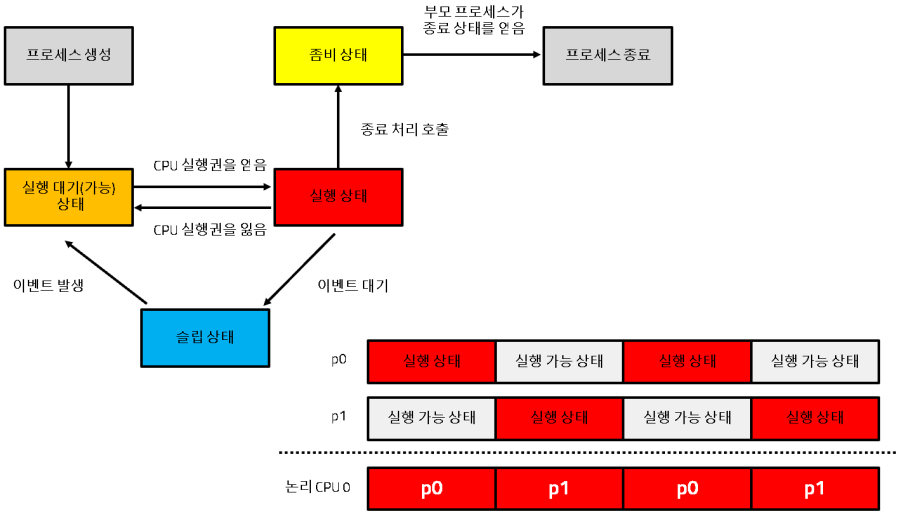
○ (1) 사용자 프로세스와 커널프로세스

- 사용자 프로세스간 번갈아가면서 cpu 시간을 할당받는다.
- 커널도 주기적으로 **cpu**타임을 할당받아서, 사용자 프로세스를 **cpu**에 배정함 **(스케쥴링)**
- 하나의 사용자 프로세스 내부에서도 커널 시스템 콜을 통해 커널모드로 동작한다.



○ (2) 프로세스의 상태변환.

- 다음상태로, 변화하며, 실행상태일때 프로세스는 cpu에 배정된다



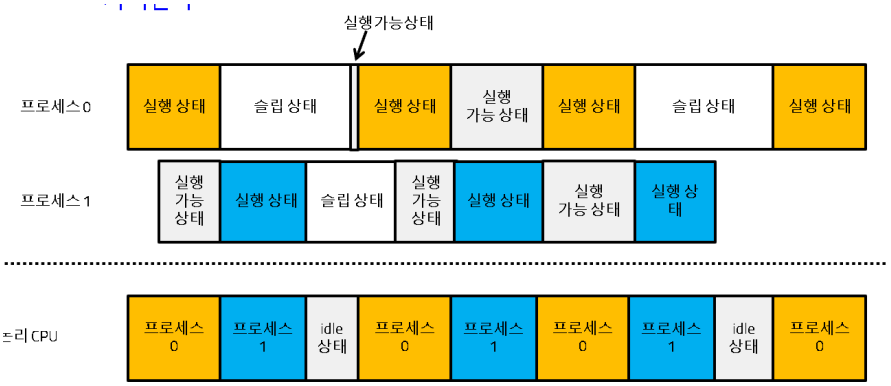
■

■ 실행대기(가능)상태

- 프로세스가 여전히 할일이 남았고,
- 멈추지 않고 계속해서 **cpu**타임을 할당받는게 유리한데,
- 각 프로세스간 균등한 서비스 타임을 할당해주기 위해,
- 강제로, 멈추고 다른 프로세스를 **cpu**에 할당하기 위해 잠시 실행대기상태로 전환시킴 **(**강제로 밀려나는것**)**

■ 슬립상태

- 외부 장치 **I/O**등을 기다리기때문에 더이상 **cpu**가 할일은 없기 때문에
- 스스로 슬립상태로 전환되는것이 맞음.
- 그래서 그래서 커널이 관리할 프로세스, 즉 cpu 타임을 할당할 프로세스의 숫자를 줄임.
- 슬립상태에 있다가 **I/O**준비를 완료하면, 이벤트가 발생하고 이때 다시 실행대기상태 큐에 들어감. **(**반대 방향으로 직접 실행상태로 가지 못함**)**

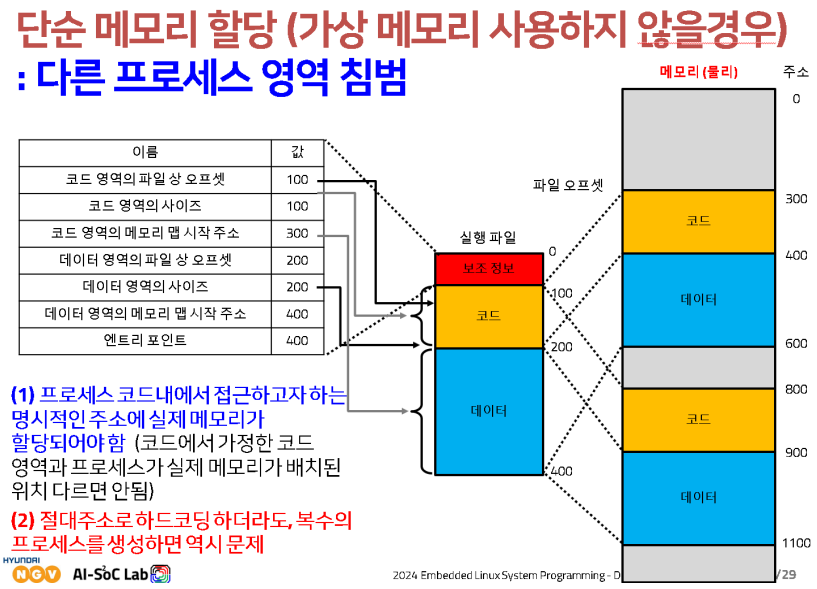


■

● 4. 멀티프로세스를 위한 가상메모리

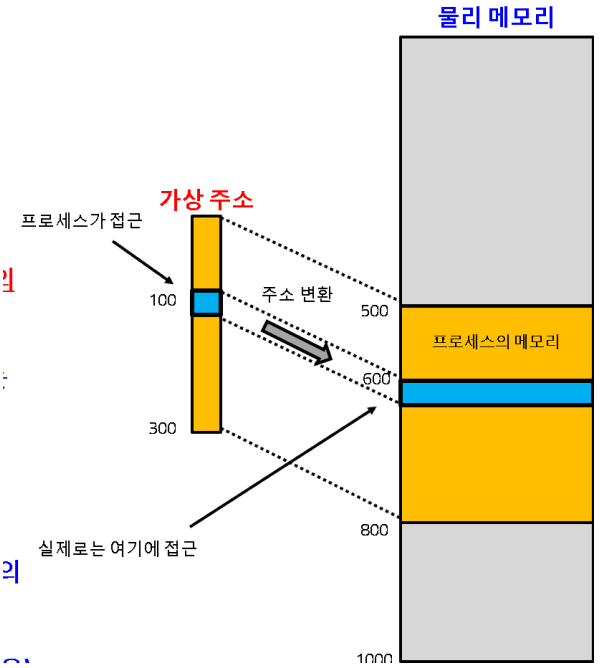
- 멀티프로세스가 실현되기 위해서는 가상메모리 개념이 지원되어야 한다

○ **(1)** 가상메모리가 지원되지 않으면

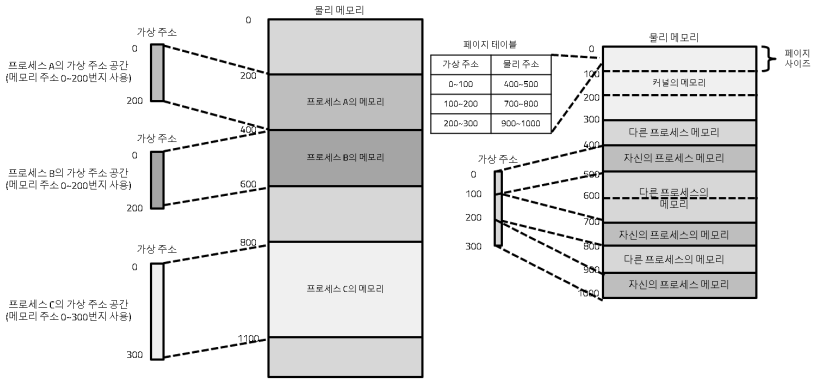


○ **(2)** 가상메모리 영역에 컴파일/링킹된 코드 이미지가 그대로 실행되는 원리

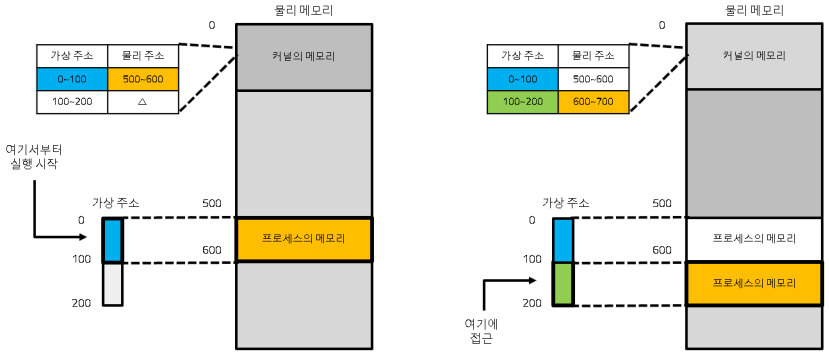
- 모든 프로그램은 가상메모리영역을 대상으로 컴파일/링킹되어 파일에 저장한 상태임.
- 따라서 **cpu**가 이 프로그램을 읽어서 디코딩해보아도 접근할 메모리(주소) 영역 (변수가 배치된 주소 및 함수 시작주소)도 여전히 원래 가상메모리 영역 그대로 이다.
- 하지만 이 프로그램을 실제 물리메모리(**DRAM**)에 동적 배치되었으므로 프로그램이 물리 메모리에 배치된 위치를 별도의 테이블에 기록해두고 이 테이블을 이용해서 가상메모리 접근시 가상주소를 실제 물리 메모리 주소로 변환해주면된다.:



- **(3)** 가상메모리의 효과
  - 이 개념을 통해 모든 프로그램은 각자 가상메모리 영역 **(32**비트 주소영역이면, **0~4G** 전체**)**을 모두 사용할수 있고 컴파일/링킹되어 프로그램 이미지가 생성됨.
  - 프로그램이 실행되어 DRAM에 동적배치되어 프로세스가 생성될때 각 프로세스마다 주소영역 변환 테이블 **(**페이지 테이블**)**을 생성/업데이트 한다.
  - 페이지 크기를 잘게 쪼개면, 아래 오른쪽 처럼 메모리의 작은 가용영역 조차 매핑해서 재사용할 수 있다.

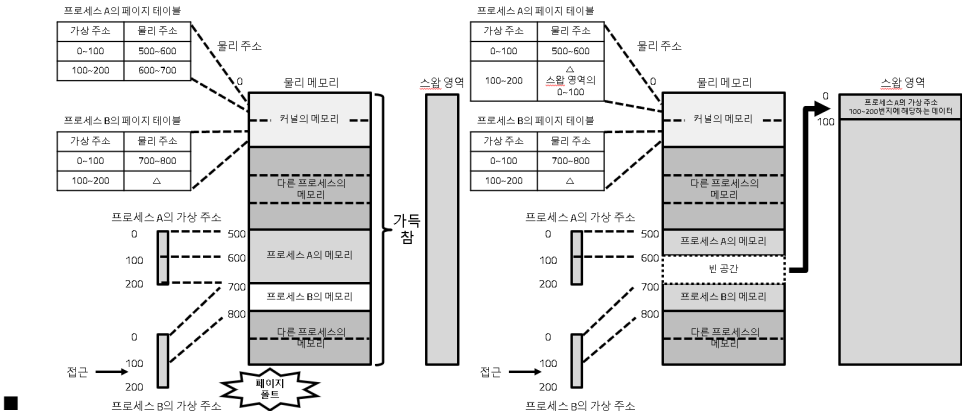


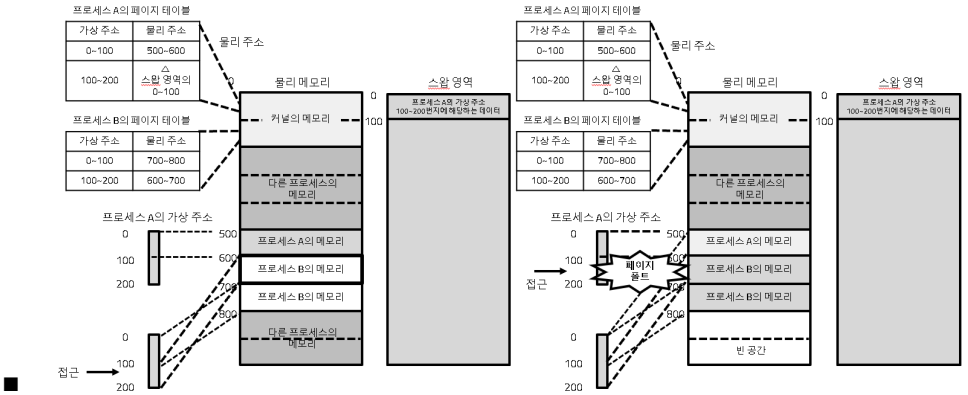
- **(4)** 온디멘드 페이징



  - mmap() 함수 이용하여 메모리를 확보하면 가상메모리 공간을 확보한 것, 여기에 접근하면, 물리 메모리를 동적으로 확보하고 페이지 변환테이블에 등록함

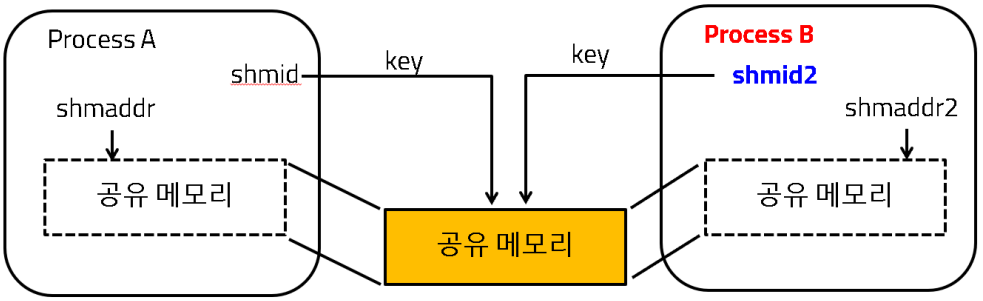
- **(5) 스왑**
  - 물리 메모리가 부족해도, 프로세스가 동작하는 이유 – 스왑
    - 저장 장치이 일부를 메모리 대신 사용하는 것.
    - 기존의 물리 메모리의 일부분을 저장장치에 저장하여 빈공간을 확보





5. 공유 메모리를 통한 프로세스간 통신

- 프로세스가 다른 프로세스에 직접 접근하는 것은 리눅스 철학에 위배됨.
- 공용 공간 (탕비실 같은곳) 만들고 약속된 이 곳에 값을 쓰고 상대방은 여기서 값을 읽어가도록 함



- Process 1안에서 shmaddr 포인터 변수는 위의 공유메모리를 가리킨다 (연결됨)
- Process 2안에서 shmaddr2 포인터 변수는 위의 공유메모리를 가리킨다 (연결됨)

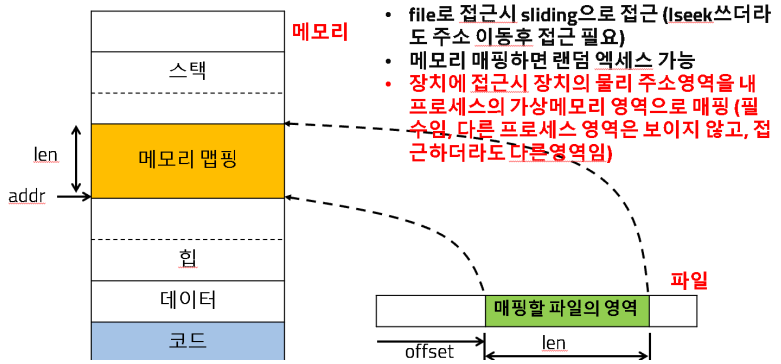
```c
int main()
{
    int shmid;
    char *shmptr1, *shmptr2;

    shmid = shmget(IPC_PRIVATE,10*sizeof(char),IPC_CREAT|0666);
    if (shmid == -1) {
        printf("shmget failed\n");
        exit(0);
    }

    if (fork() == 0) {
        shmptr1 = (char *) shmat(shmid, NULL, 0);
        for (int i=0; i<10; i++)
            shmptr1[i] = i*10;
        shmdt(shmptr1);
        exit(0);
    } else {
        wait(NULL);
        shmptr2 = (char *) shmat(shmid, NULL, 0);
        for (int i=0; i<10; i++)
            printf("%d ", shmptr2[i]);
        shmdt(shmptr2);
        if (shmctl(shmid,IPC_RMID,NULL)==-1)
            printf("shmctl failed\n");
    }
    return 0;
}
```

- **6. 장치(파일)을 내부 메모리 영역에 매핑**
  - 파일(장치)내부의 물리 주소 영역을 내 프로세스 내부 가상메모리 공간에 매핑
    - 다른 프로세스의 주소공간은 내게 보이지 않지만, **mmap**을 이용하여 파일(장치 드라이버)로 추상화된 하드웨어 영역을 현재 내 프로세스 내부의 가상메모리 주소 공간에 매핑할 수 있음

메모리 매핑의 개념

모든 것을 파일로 매핑 → 그 다음에 메모리로 매핑

- 파일의 일부 영역을 메모리에 매핑할 수 있다 (가상적으로 메모리에 배치한 효과)
  - 따라서 메모리 주소에 접근하듯이 파일에 접근이 가능하다

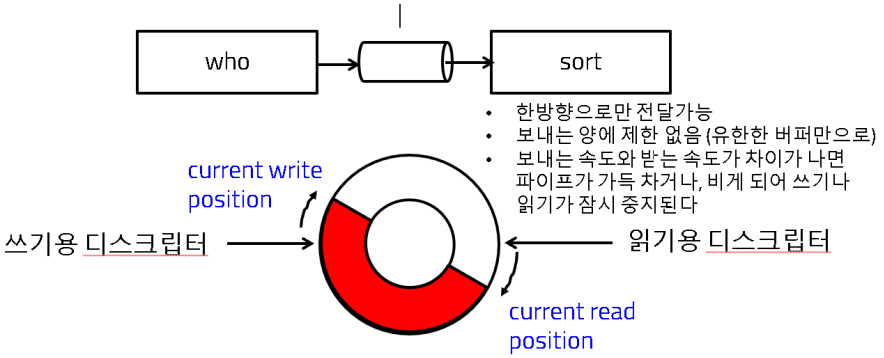


  - 파일 (장치)를 **mmap**을 통해 프로세스 내부 가상메모리영역에 배치하면
    - 이제부터 장치에 접근 하기 위해 가상메모리에 배치된 변수에 접근하면됨.

```c
int main(int argc, char *argv[])
{
    struct stat sbuf;
    char *p;
    int fd;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s file\n", argv[0]);
        exit(1);
    }

    fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(1);
    }

    if (fstat(fd, &sbuf) == -1) {
        perror("fstat");
        exit(1);
    }

    p = mmap(0, sbuf.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if (p == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    for (long l = 0; l < sbuf.st_size; l++)
        putchar(p[l]);

    close(fd);
    munmap(p, sbuf.st_size);
    return 0;
}
```

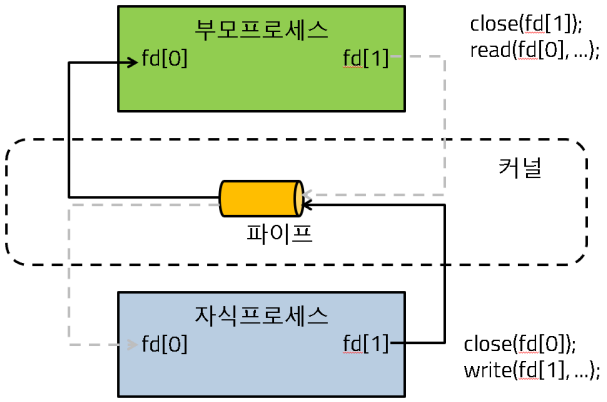
- ■

- **7. 파이프**
  - **(1) 파이프의 필요성**
    - 프로그램의 출력이 어디로 갈지 고려하지 않고 무작정 출력만 내도록 구현
    - 입력을 받는 프로그램도 어디서오는지 고려할 필요없이 입력 스트림으로부터 데이타 들어온다고 가정하고 구현
    - 출력 스트림이 어디로 흘러갈지 제어하는것은 파이프를 통해 실현함
    - 즉 파이프를 출력을 내는 프로세스와 입력을 받는 프로세스에 연결하면됨.
    - 출력 스트림이 흘러가서 입력을 받아들이는 프로세스까지의 흐름을 파이프가 제어하게 됨



- ■

○ **(2)** 부모프로세스와 자식프로세스를 파이프로 연결



➔ fork를 통해 복제하므로, 파이프와 fd[0] fd[1] 연결정보도 그래도 복제됨
➔ 이름없는 파이프는 검색이 안되므로, 연결정보를 그대로 복제한 부모와 자식 프로세스간에 통신에만 사용가능

■ **자식→부모프로세서로 데이타 전달**

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define MAXLINE 100

int main( )
{
    int n, length, fd[2], pid;
    char message[MAXLINE], line[MAXLINE];

    pipe(fd);

    if ((pid = fork()) == 0) {
        close(fd[0]);
        sprintf(message, "Hello from PID %d\n", getpid());
        length = strlen(message)+1;
        write(fd[1], message, length);
    } else {
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        printf("[%d] %s", getpid(), line);
    }

    exit(0);
```

■

○ **(3)** 이름있는 파이프를 이용하여 독립적인 파이프를 생성

■ 파일 형태로 존재하는 파이프임 **(**사용자가 자유롭게 생성할 수 있음**)**

- 이름을 가지는 파이프를 하나 생성
- **1.** 프로세스는 이 특정 파이프에 연결해서 값을 보내고
- **2.** 또다른 프로세스는 특정 파이프에 꼽아서 값을 받을수 있게 됨**.**
- **=>** 프로세스는 단순히 출력을 내보내기만하고 어디로 갈지는 파이프에 의해 결정됨**.**
- **=>** 프로세는 누군가 보내는 데이타를 읽기하만 하도록 코딩하고**,** 어느 파이프에 꼽아서 읽을지만 결정하면됨**.**

■ 특정 파이프에 값을 **write** 프로세스

- 특정 파이프를 열고 그 파이프를 대상으로 값을 전달함**.**

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#define MAXLINE 100

int main( )
{
   int fd, length;
   char message[MAXLINE];

   sprintf(message, "Hello from PID %d", getpid());
   length = strlen(message)+1;

   do {
      fd = open("myPipe", O_WRONLY);
      if (fd == -1) sleep(1);
   } while (fd == -1);

   for (int i = 0; i <= 3; i++) {
      write(fd, message, length);
      sleep(3);
   }
   close(fd);
   return 0;
}
```

- 특정 파이프에 연결하여, 값을 읽어들임

```c
int readLine(int fd, char *str);

int main( )
{
   int fd;
   char str[MAXLINE];

   unlink("myPipe");
   mkfifo("myPipe", 0660);
   fd = open("myPipe", O_RDONLY);

   while (readLine(fd, str))
      printf("%s \n", str);

   close(fd);
   return 0;
}

int readLine(int fd, char *str)
{
   int n;
   do {
      n = read(fd, str, 1);
   } while (n > 0 && *str++ != '\0');
   return (n > 0);
}
```

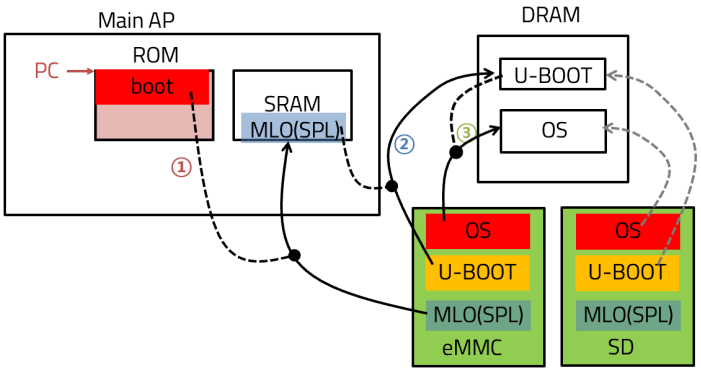
2024년09월05일 (목, 4/4일차)

- **1. 부트로더, OS부팅과정.**
  - **(1) 3단계 부팅의 필요성**
    - https://drive.google.com/file/d/1L5bPXVnAr-0v8DEPSXJFtb9abm_ZJw5u/view?usp=sharing

OS가 부팅되는 과정

3단계의 코드적재, 이미지이동, 실행과정을 거침

- ① 최초 온칩 flash에 있는 부트 코드 (bootstrap, 칩벤더가 구워둔 이미지) 실행
  - eMMC 혹은 SD카드 메모리중에 선택해서 MLO영역 코드를 SRAM으로 적재
- ② cpu의 실행위치 즉 PC가 이제 SRAM으로 이동함
  - SRAM에 적재된 MLO 코드 실행시 U-BOOT 이미지를 DRAM에 적재
- ③ **DRAM에 적재된 U-BOOT코드가 실행되어 S이미지를 DRAM으로 이동시킴 (부팅)**



■

○ **(2) Custom OS**로 부팅하는 방법

■ 1. 이미 컴파일된 바이너리 이미지 다운 (첫번째 방법, 업체에서 제공해줄경우)

- **1) U-Boot**, 커널, 파일시스템, 각종 프로그램을 하나로 빌드해둔 이미지 다운로드
- **2)** 다운로드한 파일을 **SD**카드나 **eMMC**에 **flash write**하기
  - 전용 툴을 사용해야 함,
  - 특정 약속된 메모리 영역에 써야 하기때문,
  - AP칩 벤더의 부트롬이 약속된 영역에 접근해서 코드를 읽어서 실행하기 때문임

■ 2. 소스를 받아서 빌드 (두번째 방법)

- **1)** 호스트 컴퓨터에서 실행가능한 크로스컴파일러 설치
- **2) u-boot**소스, 커널소스, 디바이스 드라이버 소스를 직접 받아서 빌드
  - 빌드하는 것은 호스트 컴퓨터에서 크로스 컴파일 해야 함.
  - 선택적으로 패키지 정해서 컴파일/빌드해서 OS이미지 만드는 과정을 편라하게 자동화 해주는것이 yocto임

Linux 커널 이미지 및 device driver 빌드

- **git clone https://github.com/beagleboard/linux.git**
- **cd linux**
- **make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- omap2plus_defconfig**
- **make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j$(nproc)**
- **ls arch/arm/boot/**

```
ghjeon@ghjeon-ThinkBook-16-G7-IML:~/linux$ ls arch/arm/boot/
bootp       deflate_xip_data.sh  Image       Makefile
compressed  dts                  install.sh  zImage
```

- **ls arch/arm/boot/dts/ti/omap/am335x-boneblack.dtb**

```
ghjeon@ghjeon-ThinkBook-16-G7-IML:~/linux$ ls arch/arm/boot/dts/ti/omap/am335x-boneblack.dtb
arch/arm/boot/dts/ti/omap/am335x-boneblack.dtb
```

■ 3. flash 메모리 파티션 잡고, rom write하기

SD card 에 uboot, kernel, root file system복사

- sudo mount /dev/sdx1 /media/${USER}/BOOT
- sudo mount /dev/sdx2 /media/${USER}/ROOT

mount

- sudo cp u-boot/MLO /media/${USER}/BOOT
- sudo cp u-boot/u-boot.img /media/${USER}/BOOT
- sudo cp linux/arch/arm/boot/zImage /media/${USER}/BOOT
- sudo cp linux/arch/arm/boot/dts/ti/omap/am335x-boneblack.dtb /media/${USER}/BOOT

BOOT partition (커널이미지 디바이스드라이버 설치)

- sync
- sudo tar -xf debian-10.3-minimal-armhf-2020-02-10/armhf-rootfs-debian-buster.tar –C /media/${USER}/ROOT
- sync

ROOT partition (file systems)

- sudo umount /media/${USER}/BOOT
- sudo umount /media/${USER}/ROOT

unmount

-

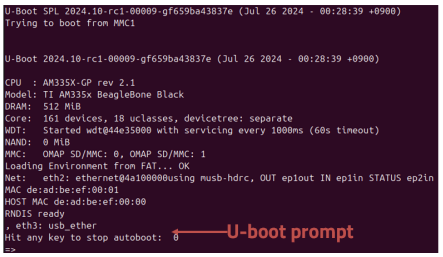
■ 4. 특정 OS를 로딩하도록 uboot 환경설정

- 어느 영역에 배치된 OS를 실행할지 지정
- 부팅중에 매번 입력해도 되지만, 환경변수를 잡고 저장해두면 편하다

부팅시 최초 eMMC,혹은 SD카드 선택

(On-Chip Flash ROM 코드에 의해 – 마치 컴퓨터의 BIOS임)

- GPIO특정 버튼 누르고 부팅하면 eMMU가 아닌 SD에 저장된 U-BOOT가 실행됨
- Space bar 빠르게 연타하면 uboot 설정 모드로 바뀜

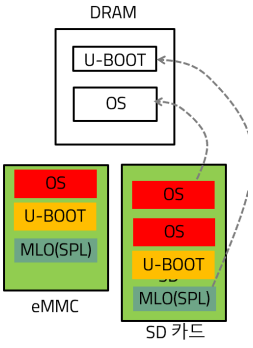


-

■ 5. 선택적 부팅

- Then there are two zImages named zImage, zImage2.
  So, we have to inform u-boot to boot the zImage2, not zImage.
  - **setenv bootfile zImage2**
  - **setenv loadimage fatload mmc ${mmcdev}:${bootpart} ${loadaddr} ${bootdir}${bootfile}**
  - **setenv bootcmd "run mmcargs; run loadimage; run loadfdt; bootz ${loadaddr} - ${fdtaddr}"**
  - **saveenv**
  - **reset(reboot)**

```
root@arm:~# uname -r
6.7.7
```



-

■

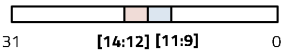
● 2. 하드웨어 장치에 직접 접근 (memory mapped I/O)

○ (1) 장치 Address map을 통한 접근 (MCU F/W 코딩 스타일)

■ 스펙 문서 읽고 주소확인
■ 포인터를 통해 해당 영역에 접근
■ 개별 비트를 비트연산을 수행해서, 값을 변경/읽기.

GPIO 레지스터 설정 - GPFSEL

32bit GPFSEL 레지스터



BCM2711_ARM_peripherals.pdf p.68

GPFSEL2 Register

Bits	Name	Description	Type	Reset
31:30	Reserved.	-	-	-
29:27	FSEL29	FSEL29 - Function Select 29 000 = GPIO Pin 29 is an input 001 = GPIO Pin 29 is an output 100 = GPIO Pin 29 takes alternate function 0 101 = GPIO Pin 29 takes alternate function 1 110 = GPIO Pin 29 takes alternate function 2 111 = GPIO Pin 29 takes alternate function 3 011 = GPIO Pin 29 takes alternate function 4 010 = GPIO Pin 29 takes alternate function 5	RW	0x0
26:24	FSEL28	FSEL28 - Function Select 28	RW	0x0
23:21	FSEL27	FSEL27 - Function Select 27	RW	0x0
20:18	FSEL26	FSEL26 - Function Select 26	RW	0x0
17:15	FSEL25	FSEL25 - Function Select 25	RW	0x0
14:12	FSEL24	FSEL24 - Function Select 24	RW	0x0
11:9	FSEL23	FSEL23 - Function Select 23	RW	0x0
8:6	FSEL22	FSEL22 - Function Select 22	RW	0x0
5:3	FSEL21	FSEL21 - Function Select 21	RW	0x0
2:0	FSEL20	FSEL20 - Function Select 20	RW	0x0

- GPFSEL 레지스터에 어떤 값을 쓰는지에 따라 GPIO pin의 동작을 결정할 수 있다
→ 3자리 bit의 값에 따라 pin이 input인지 / output인지 결정

- 제어할 23,24번 pin은 GPFSEL2의 [11:9], [14:12]번째 bit이다.

- **Input에 사용할 23pin의 11:9번째 bit를 '000'으로**
- **output으로 사용할 24pin의 14:12번째 bit를 '001'으로 설정한다.**

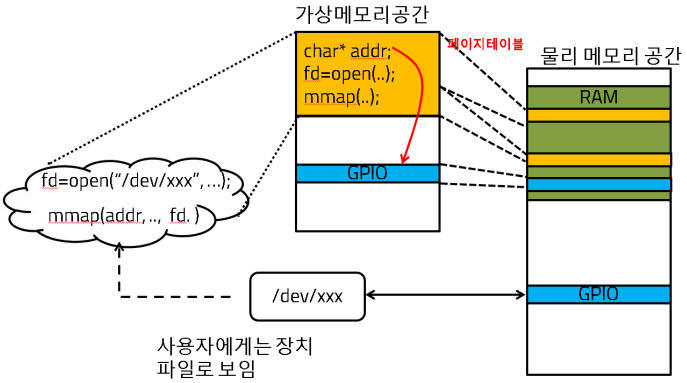
■

○ (2) 메모리 매핑의 필요성

■ 프로세스 내부에서 접근하는 주소영역은 모두 가상메모리 영역임

■ 즉 칩 스펙 문서를 읽어보니 **GPIO**의 물리 주소가 **0xFB000000**일 경우 프로세스 내부에서 ***(0xFB000000)**를 통해 접근하더라도 **GPIO**에 접근할 수 없음.

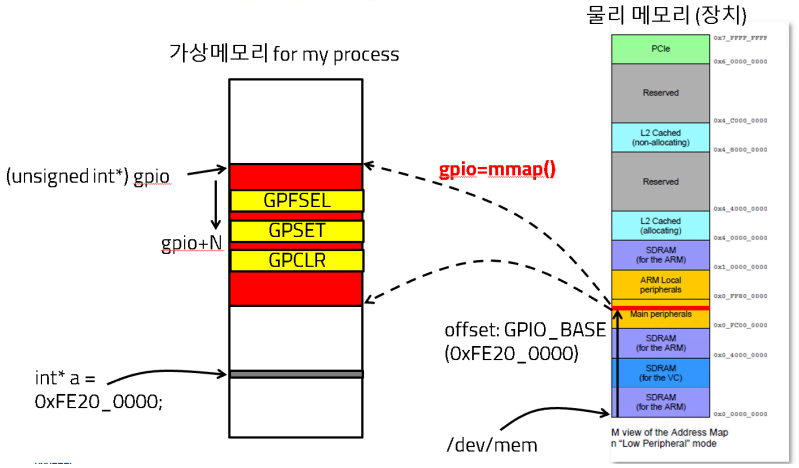
■ 따라서, 하드웨어 메모리 맵 영역을 사용자 프로세스 내부로 매핑한 뒤에 접근한뒤 접근가능해짐



■

- **(3)** 하드웨어 주소공간을 나의 가상메모리 내부 주소공간에서 접근

물리적 메모리(하드웨어)영역을 나의 변수공간으로 매핑한 결과
(선택이 아니고 필수임, 가상메모리 정책상)

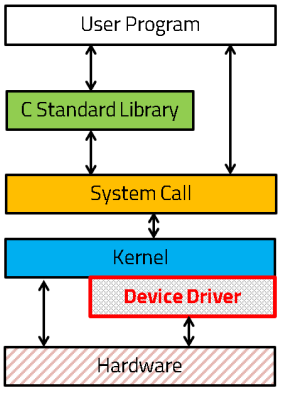


-
-

3. 디바이스 드라이버를 이용한 장치 접근

(1) 커널 모듈의 개념, 필요성;

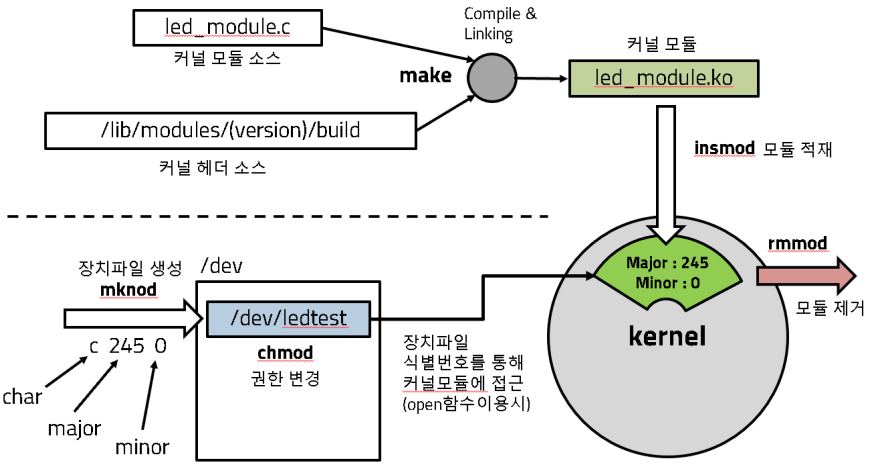
- 새로운 장치를 인식하여 동작시키려면 커널을 확장해야 한다.
  - 커널을 리빌드 하는것은 아무 복잡하다.
  - 장치에 접근하여 제어하는 코드를 동적 라이브러리로 만들어서 커널에 동적 삽입을 할수 있다면 마치 커널이 확장되는 효과를 누릴수 있다.



-

- 커널 모듈은 커널에 로드, 언로드할 수 있는 코드 조각이다.
- 런타임 중에 로드되는 커널 모듈은 런타임에 커널에 코드를 추가하거나 제거하는 매커니즘이다.
  - 즉, 런타임에 커널의 기능이 확장된다는 의미이다.
- 커널 모듈의 조작을 통해 사용자는 커널 레벨에서 하드웨어에 접근할 수 있다.
- 디바이스 드라이버는 커널이 시스템에 연결된 하드웨어에 엑세스할 수 있도록 하는 커널 모듈의 일종이다.

- **(2)** 커널 모듈 빌드 및 실행.



  -
  - 빌드
    - **make all**
    - ***.c**로부터 ***.ko**를 생성.
  - 커널에 삽입
    - **sudo rmmod led_test_kernel**
    - **sudo insmod led_test_kernel.ko**
  - 장치드라이버 생성
    - **sudo rm -rf /dev/led_test**
    - **sudo mknod /dev/led_test c 246 0**
    - **sudo chmod 666 /dev/led_test**
- **(3)** 디바이스 드라이버 이용한 장치 접근.
  - ./main 1
  - ./main 0
  - LED 온오프 확인.

- 시험문제
  - 1 프로그램 생성과 동시에 열리는 표준 입출력이 아닌것은
    - stdin stdout, stderr 외에 먼가가 정답이 된다.
  - 2 파일에 접근하기 위해 최초로 파일을 열어서, 파일 구조체로 포인터를 얻기 위한 표준함수는
    - fopen
  - 3 하드웨어 장치에 접근하기위한 저수준의 함수... 커널이 제공....
    - system call
  - 4 프로그램 실행시, 점프할 함수를 내 프로그램에 동적으로 결합
    - 동적 링킹
  - 5. 리눅스에 다룰수 잇는 모든 장치를(하드웨어를) 열기 위한 함수로, 파일 디스크립터를 반환하는 함수
    - open()
  - 6. 하나의 파일의 특정영역을 복수의 프로세스가 접근할때 I/O 잠금

        - flock()
    - 7. 운영체제와 나 사이에 인터페이싱해주는... 프로그램을 커널위에 올려주는. 프로세스 생성.. (fork(), exec())
        - 쉘. shell
    - 8. 파일의 소유자는 아니지만, 실행할때 잠시 소유자의 권한을 획득
        - set user id (effective id)
    - 9. fork()를 이용해서 자식을 생성하는 과정에 대해 틀린것.
        - 자식프로세느는 부모프로세스를 완전히 복제
        - 자식프로세스는 프로그램의 처음부터 시작. **(X)**
        - 부모프로세스의 fork()의 리턴값은 0이 아니다.
        - 부모프로세스는, 자식프로세스가 끝날때까지 기다리지 않고 바로 실행을 이어감.
    - 10. 하드웨어 자원의 특정 영역을 메모리의 한 영역으로 매핑해서, 마치 메모리에 읽고 쓰는 방식으로 자원에 접근하게 해주는 함수
        - lseek
        - dup
        - **mmap**
        - read
        -
- 책 추천
    - 임베디드 프로그래밍 **C**코드 최적화
    - 깐깐하게 배우는 **C**
    - 전공자를 위한 **C** 언어 프로그래밍
    - 자료구조와 함께 배우는 알고리즘 입문
    - 다시 시작하는 프로그래밍
    - **Embedded Recipes**
    - **Debug Hacks** 디버그를 극대화하는 테크닉 **&** 툴
    - 리눅스 바이너리 마스터
    - 원리부터 실무까지 **ARM** 프로그래밍
    - 임베디드 엔지니어 교과서
    - 임베디드 메모리 최적화 기법
    - 프로그래머가 몰랐던 멀티코어 **cpu** 이야기
-