임베디드 리눅스 이해

2024.06

(주)다인시스

DIGNSYS
www.dignsys.com

목 차

운영 체제

- 운영 체제 (OS : Operating System)
  - 컴퓨터 시스템의 전반적인 동작을 제어하고 조정하는 시스템 프로그램들의 집합.
- 운영체제의 역할
  - 하드웨어와 응용 프로그램 간의 인터페이스 역할을 하면서 CPU, 주기억장치, 입출력 장치 등의 컴퓨터 자원을 관리

운영 체제의 필요성 (1)

1. 자원 관리의 효율성

- **제한된 하드웨어 자원 관리**: 제한된 CPU, 메모리, 저장 공간을 효율적으로 관리하여 시스템의 성능을 극대화
- **전력 관리**: 배터리로 구동되는 임베디드 시스템에서는 전력 소비를 최소화하는 것이 중요. OS의 전력 관리 기능을 통해 에너지 효율성을 높임

2. 실시간 성능 보장

- **실시간 스케줄링**: 임베디드 시스템에서는 특정 작업이 정해진 시간 내에 완료되어야 하는 실시간 요구사항이 많음. 운영체제는 실시간 스케줄링 알고리즘을 통해 이러한 요구사항을 충족
- **우선순위 관리**: 중요한 작업이 먼저 수행되도록 우선순위를 설정하고 관리하여 시스템의 응답성을 높임

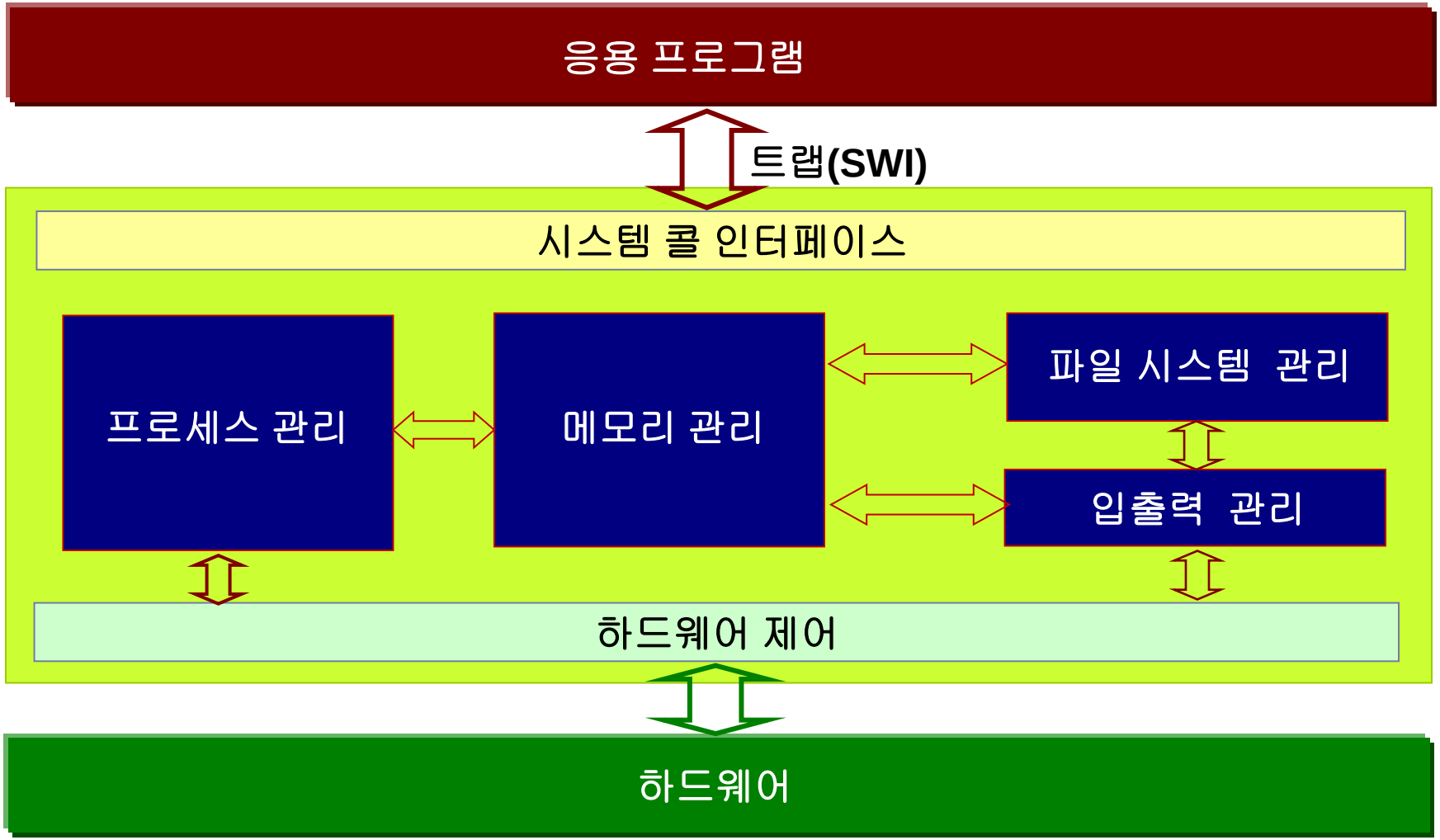
3. 개발 및 유지보수 용이성

- **모듈화 및 재사용성**: OS를 사용하면 SW를 모듈화하여 개발 및 재사용 가능한 코드 작성이 용이. 개발 시간과 비용을 절감
- **디버깅 및 테스트**: 운영체제는 다양한 디버깅 도구와 테스트 환경을 제공하여 소프트웨어의 신뢰성을 향상 시킴

운영 체제의 필요성 (2)

4. 다중 작업 처리
   - **멀티태스킹**: 여러 작업을 동시에 실행할 수 있는 멀티태스킹 기능 제공. 임베디드 시스템이 동시에 여러 센서 데이터를 처리하거나 다양한 입출력 작업을 수행할 수 있게 함.
   - **인터럽트 처리**: 외부 이벤트에 빠르게 반응할 수 있도록 인터럽트 처리 메커니즘을 제공
5. 네트워킹 및 통신
   - **네트워크 스택 제공**: TCP/IP 등 네트워크 프로토콜 스택 제공
   - **데이터 전송 및 통신 관리**: 네트워크를 통한 데이터 송수신 및 통신 관리 효율적 처리
6. 보안 관리
   - **접근 제어**: 임베디드 시스템의 자원에 대한 접근 권한을 제어하여 보안을 강화
   - **데이터 암호화**: 민감한 데이터를 보호하기 위해 암호화 기능 제공
   - **시스템 무결성 검사**: 시스템이 의도하지 않은 변경이나 침입으로부터 보호되도록 무결성 검사를 수행
7. 하드웨어 추상화
   - **하드웨어 독립성**: OS는 HW 추상화 레이어를 제공하여 SW가 특정 하드웨어에 종속되지 않도록 함. 이는 다양한 HW에서 동일한 SW를 실행할 수 있게 함.
   - **드라이버 관리**: 다양한 HW 장치의 드라이버를 관리하여 장치와의 인터페이스 단순화

운영체제의 구성 요소


응용 프로그램
트랩(SWI)
시스템 콜 인터페이스
프로세스 관리
메모리 관리
파일 시스템 관리
입출력 관리
하드웨어 제어
하드웨어

임베디드 리눅스

- 임베디드 리눅스
  - 공개된 리눅스 커널과 각종 응용 프로그램 소스를 임베디드 시스템에서 동작 하도록 최적화 하여 사용하는 운영체제

- 임베디드 리눅스 특징
  - 적은 메모리 사용
  - PC에 비해 느린 CPU에서 동작
  - 특정한 목적의 응용 프로그램 탑재

임베디드 리눅스의 중요성

- 임베디드 리눅스의 중요성
  - 임베디드 리눅스는 자동차 산업, 통신기기, 산업자동화, 의료기기, 에너지 기기, 스마트홈 디바이스 등 다양한 분야에 적용되어 사용되는 운영체제
  - 최근에는 자동차 산업의 전자화로 AGL(Automotive Grade Linux)과 같은 리눅스 기반 오픈소스 프로젝트, 자율주행과 인포테인먼트 시스템에서 많이 사용되고 있음
  - 온디바이스AI와 엣지컴퓨팅 디바이스의 필수 운영체제로 사용되고 있기 때문에 미래 전자산업의 핵심 운영체제
- 임베디드 리눅스 확장
  - 스마트폰 등에서 사용하는 Android OS
  - 삼성전자의 TV, 냉장고 등 모든 가전에 적용하는 Tizen OS
  - LG전자의 가전에 적용하는 webOS

임베디드 리눅스의 장점

- GPL(GNU General Public License)를 따르는 공개 소스 소프트웨어
  - 두터운 개발자 층
  - 검증된 운영체제
- 모듈 지원
  - 불필요한 부분을 빼고 필요한 부분 추가가 가능하여 이식성 우수
- 유닉스와 유사한 운영체제 시스템, 독립적인 POSIX 구현
- 다양한 플랫폼 지원
  - x86, alpha, ppc, arm, sparc, 메인프레임에 이르기까지 각종 CPU 지원
- 멀티태스킹, 가상메모리, 공유 라이브러리, 디멘트 페이징, 메모리 관리, TCP/IP 네트워킹 등을 지원

임베디드 리눅스의 단점

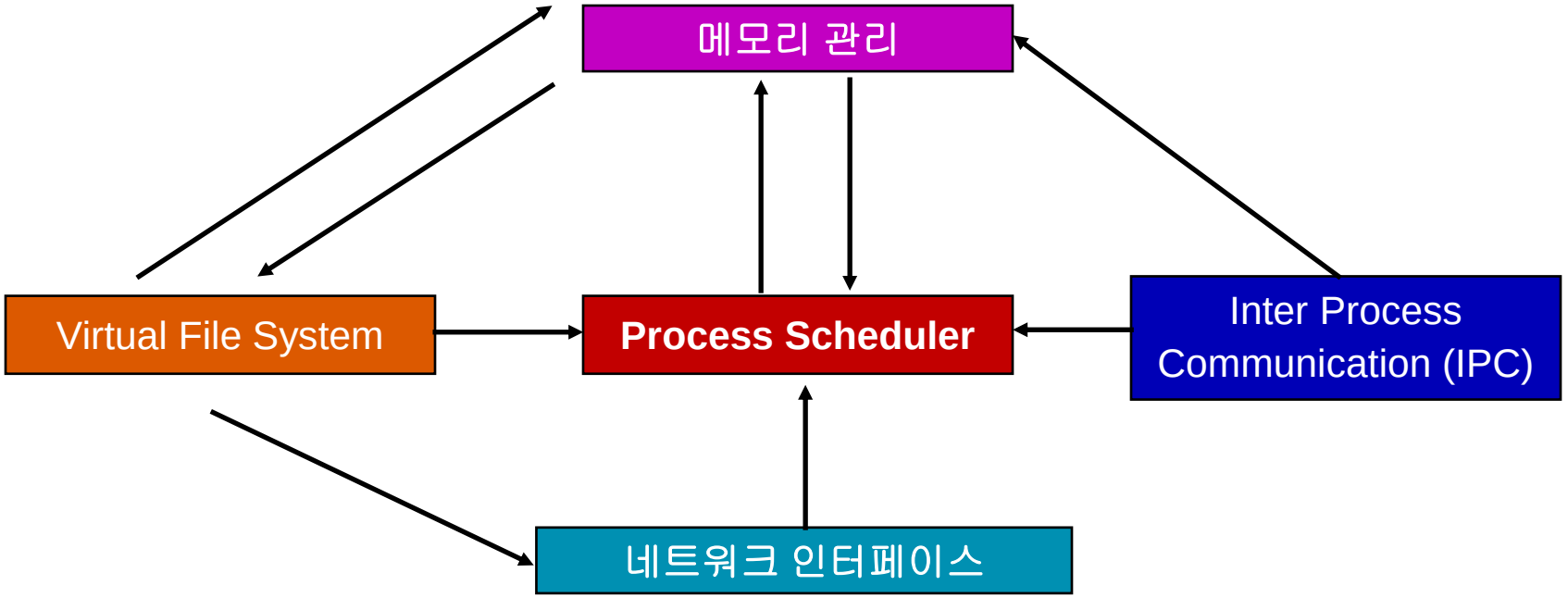
- 개발 환경 설정이 까다롭다
- 실시간성(Real-Time) 기능이 취약하다.
  - 최근 다양한 RT Linux Patch 지원
- 메모리와 전력 소모량이 많다
- 리눅스의 윈도우 시스템
  - 다양한 윈도우 시스템 지원 Qt, Xwindows 등

목 차

운영체제의 커널

- 커널이란 ?
  - 운영체제의 핵심 부분
  - 하드웨어와 운영체제의 다른 부분 사이의 중재자 역할을 한다.
- 마이크로 커널(micro kernel)
  - 커널의 기능을 최소화 하고 가장 핵심 기능만 가진다.
    - 커널은 동기화 프로그램, 매우 간단한 스케쥴러와 프로세스간 통신 방식만 제공
  - QNX, 카네기 멜론(Carnegie Mellon)의 마크 3.0(Mach 3.0)
- 모놀리틱 커널(monolithic kernel)
  - 커널이 운영체제가 관장하는 모든 서비스를 가진다.
    - 프로그램의 실행을 제어하며 데이터와 파일 관리
  - 유닉스, 리눅스 시스템

리눅스 커널의 구조 (1)


메모리 관리
Virtual File System
Process Scheduler
Inter Process Communication (IPC)
네트워크 인터페이스

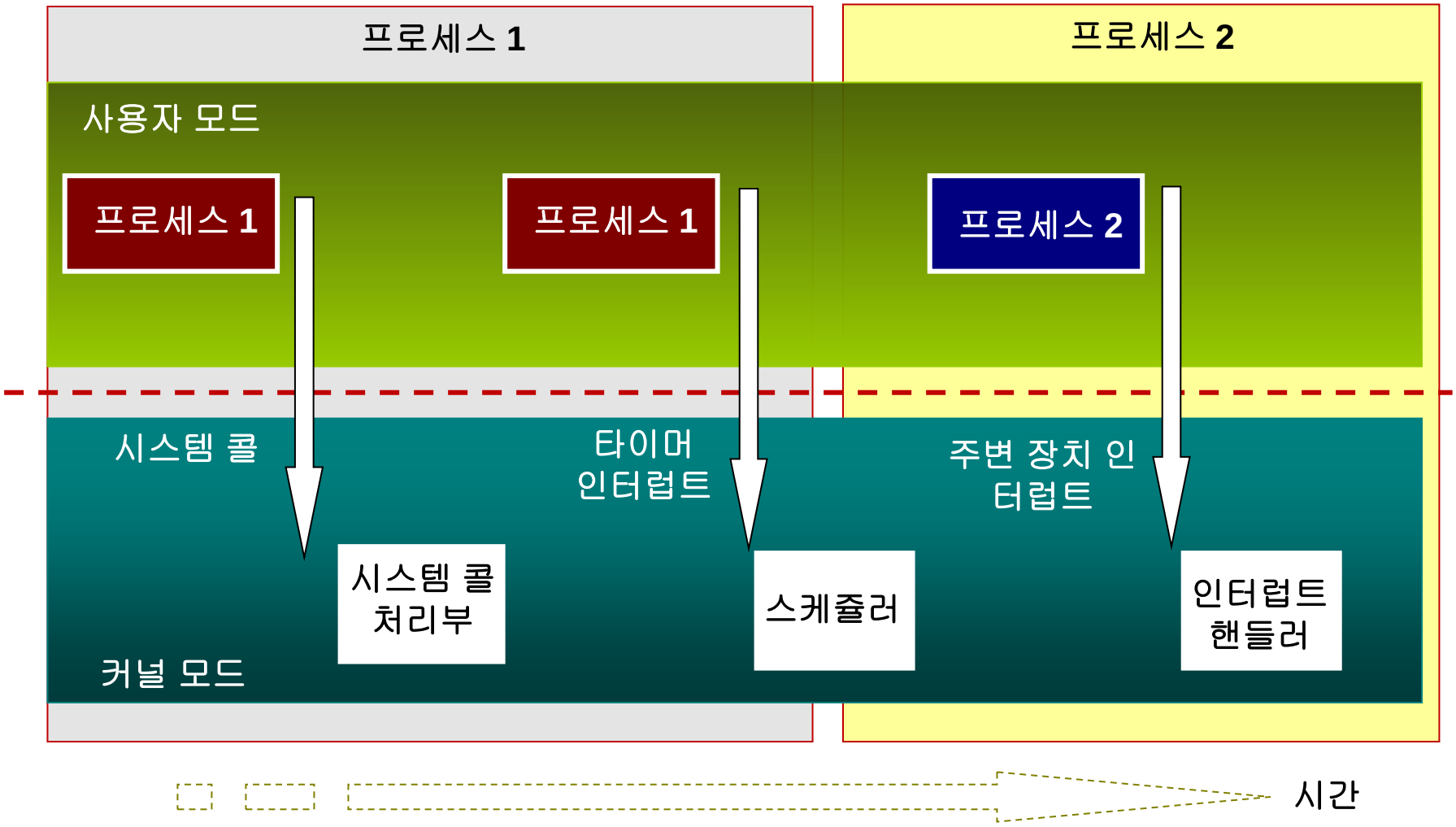
리눅스 커널의 구조 (2)

- Process Scheduler
  - CPU를 여러 프로세스가 공평하게 사용할 수 있도록 한다.
- 메모리 관리(Memory Manager)
  - 여러 개의 프로세스 메인 메모리를 안전하게 공유 할 수 있도록 한다.
- Virtual File System
  - 리눅스에서 제공하는 모든 파일시스템들에 대한 공통적인 인터페이스 제공
  - 하드웨어 장치(문자 디바이스,블록디바이스)도 파일로 관리 된다.
- 네트워크 인터페이스(Network Interface)
  - 표준 네트웍 프로토콜과 드라이버를 제공한다.
- Inter Process Communication (IPC)
  - 프로세스 간에 서로 정보를 교환 할 수 있는 방법을 제공한다.

프로세스

- 프로세스(Process) 또는 태스크(Task)
  - 실행 중인 프로그램
  - 프로세스의 구성
    - 프로그램, 데이터, 스택
    - 프로세서 내부의 PC를 비롯한 레지스터 와 상태 정보
- 멀티 프로세싱
  - 동시에 여러 프로세스가 활동
- 프로세스는 프로그램이 아니다
  - 여러 개의 프로세스가 똑같은 프로그램을 동시에 실행 가능
  - 프로세스 하나가 여러 프로그램을 차례로 실행 가능
- 리눅스에서는 프로세스가 실행 중일 때 새로운 프로세스가 끼어드는 현상을 허용하지 않는다.
  - Task preemption이 지원되지 않는다.
    - 실행중인 프로세스는 선점하지 않는다
  - 새로운 프로세스를 실행하기 위해서는 시스템 이벤트를 기다린다.

커널/프로세스 모델


프로세스 1
프로세스 2
사용자 모드
프로세스 1
프로세스 1
프로세스 2
시스템 콜
타이머
인터럽트
주변 장치 인
터럽트
시스템 콜
처리부
스케쥴러
인터럽트
핸들러
커널 모드
시간

리눅스 프로세스의 Preemptive

- 최근 리눅스 커널에서 소프트 리얼타임 지원
  - 커널 옵션 CONFIG_PREEMPT 설정
  - 우선 순위가 높은 프로세스 실행 시 Preemption 가능

- 리눅스 프로세스의 Preemption 동작
  - 프로세스가 TASK_RUNNING 상태로 진입할 때 커널에서 우선 순위 체크
  - 현재 실행중인 프로세스보다 높은 우선 순위를 가지는 프로세스가 시작되면 현재 프로세스를 중지하고, 새로운 프로세스를 실행
  - 실행 중인 프로세스는 need_resched 필드가 설정되고, 펜딩된 프로세스의 처리가 끝나면 멈췄던 프로세스를 실행한다.

스케쥴러(Scheduler)

- 한 순간에 오직 하나의 프로세스 만이 CPU를 점유 가능
  - 스케쥴러는 여러 프로세스가 CPU를 공유하여 사용 가능하도록 해준다.
  - 즉, 다음에 실행할 태스크를 결정
- 선점형(preemptive)과 비선점형(non-preemptive) 스케쥴러
  - 비선점형
    - 프로세스가 자발 적으로 CPU 사용권 반납했을 때만 스케쥴링
  - 선점형
    - 프로세스 별 사용 시간을 지켜보면서 정기적으로 스케쥴링
- 스케줄링 정책(Scheduling Policy)
  - 리눅스의 스케쥴링 정책
    - SCHED_FIFO, SCHED_RR 등

```
#define SCHED_NORMAL    0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH     3
#define SCHED_IDLE      5
```

리눅스의 스케쥴링 정책

- 리눅스의 스케쥴링 정책
  - 각 프로세스는 우선순위를 가진다.
    - 높은 우선순위의 프로세스가 낮은 우선순위 보다 먼저 실행
  - 동일 우선 순위는 라운드 로빈(round robin)으로 실행
  - 리눅스는 동적 우선 순위를 지원
  - 각 프로세스는 time-slice 동안 처리, time-slice 가 지나면 다른 프로세스 실행
    - Preemptive scheduling
  - Time-slice
    - 각 프로세스가 실행되는 최소한의 시간 단위
    - 기본적으로 100ms 사용
- 리눅스의 우선순의
  - nice 값에 따른 우선순위 : 100~140까지
    - nice는 -20 ~ +19 값을 가지며 큰 nice 값이 낮은 우선 순위
  - 실시간 우선 순위(SCHED_FIFO, SCHED_RR만 지원)
    - 0~99까지의 값을 가진다
    - POSIX 실시간 표준에 따른다

리눅스의 타임 슬라이스와 nice 값

- “nice” 값에 의해 동적으로 할당

태스크 종류	Nice 값	타임슬라이스 길이
최저 우선순위	+19	5ms
기본 우선순위	0	100 ms
최고 우선순위	−20	800 ms

- 우선순위 변경
  - ‘nice’ 또는 ‘renice’ 명령어 사용

```
nice  -n  [nice value]  [명령]
renice  [nice value]  -p  [프로세스 ID]
```

스케쥴링 관련 시스템 콜

시스템 콜	설명
nice( )	일반 프로세스의 우선순위를 변경
getpriority( )	일반 프로세스의 최대 우선순위 값을 가져온다
setpriority( )	일반 프로세스의 우선순위를 설정
sched_getscheduler( )	프로세스의 스케쥴링 정책을 얻는다
sched_setscheduler( )	프로세스의 스케쥴링 정책을 설정
sched_getparam( )	프로세스의 우선순위를 얻는다
sched_setparam( )	프로세스의 우선순위를 지정
sched_yield( )	일시적으로 프로세스를 양도
sched_get_priority_min( )	스케쥴링 정책에서 사용가능한 최소 우선 순위 값을 얻는다
sched_get_priority_max( )	스케쥴링 정책에서 사용가능한 최대 우선 순위 값을 얻는다
sched_rr_get_interval( )	Round Robin 정책에서의 타임슬라이스 값을 얻는다

스케쥴링 알고리즘

- O(n) 스케쥴링
  - 과거 2.4 커널까지 사용
  - 스케쥴링 동안에 모든 프로세스의 우선 순위 및 타임 슬라이스 값을 계산
  - 태스크가 많아지면 스케쥴링에 많은 시간 소요
    - n개의 프로세스가 사용되면 O(n) 시간이 소요
- O(1) 스케쥴링
  - 2.6 커널 에서 새롭게 도입, 커널 2.6.22까지 사용
  - 프로세스 개수에 무관하게 일정한 시간 내에 스케쥴 가능하도록 수정
    - 항상 일정한 시간의 스케쥴링 시간 O(1)이 소요
  - SMP를 지원하는 구조로 설계되어 멀티코어 시스템에서 좋은 확장성 제공
- CFS 스케쥴링(Completely Fair Scheduling)
  - 커널 2.6.23 이후부터 사용
  - nice 값에 따라서 타임 슬라이스가 정해진 것이 아니고, nice 값의 가중치에 따라 타임 슬라이스를 동적으로 할당하여 공정하게 스케쥴링이 되어 프로세스 전환에 따르는 오버헤드를 줄이는 방법

O(n) 스케쥴링 과 O(1) 스케쥴링

- O(n) 스케쥴링
  - N 개의 태스크(프로세스)가 있을 때 처리하는 시점에서 모든 태스크의 우선순위를 계산하여 한 개를 선택하여 처리
  - 프로세스의 개수가 늘어나면 스케쥴링을 위한 시간도 증가, O(n) 시간 필요

- O(1) 스케쥴링
  - 구동중인 프로세스 수에 무관하게 O(1)으로 동작 : constant-time scheduler
  - Process (thread)의 개수가 적을 경우, 큰 효과 없음
    - The numbers of Processes ≤ 5

O(1) 스케쥴링

- 미리 우선순위를 정해놓고 항상 첫번째 우선순위의 태스크를 처리하는 것을 O(1) 스케쥴링
  - 태스크의 수가 증가하더라도 일정한 스케쥴링 시간을 가진다.
- Priority 기반 Multi-level feedback queue 사용
- 런 큐 (Run Queue)
  - 두개의 우선순위 배열을 가진다
    - Active 우선순위 배열 과 Expired 우선순위 배열

```
struct   runqueue {
             ~~~~~
             struct   prio_array   *active;      // this is scheduled
             struct   prio_array   *expired;
             struct   prio_array   *arrays[2];
}
```

우선순위 배열 (Priority Array)

- 우선순위 배열의 관리
  - Active 우선순위 배열
    - 타임 슬라이스가 남아있는 태스크
    - 실질적으로 스케쥴링 대상이 되는 태스크
  - Expired 우선순위 배열
    - 타임 슬리이스가 소진된 태스크
- 우선순위 배열 동작
  - Active 배열에서 타임 슬라이스를 모두 소모한 태스크는 타임 슬라이스와 우선순위를 재 계산하여 Expired 배열로 이동
  - Active 배열의 태스크들이 모두 Expired 배열로 옮겨진 경우 Active와 Expired의 포인터 스왑

```
struct   prio_array {
            int   nr_active;
            unsigned   long  bitmap[BITMAP_SIZE];
            struct   list_head_queue[MAX_PRIO];
}
```

O(1) 스케쥴러의 문제점

- 응답 시간(Response time) 저하
  - 과도한 타임 슬라이스 할당에 의한 응답 시간 저하
  - 예) 태스크들의 nice 값 할당이 0인 경우 100ms마다 스위칭 발생
- 성능(Throughput) 저하
  - 과도한 스위칭 오버헤드로 인한 CPU 시간 낭비
  - 예) 태스크들의 nice 값 할당이 19인 경우 5ms마다 스위칭 발생
- 우선순위 1단위 차이에 따른 CPU 할당량이 공평하지 않음
  - 예)
    - nice값이 0인 경우 100ms,1인 경우 95ms: 5%감소
    - nice값이 18인 경우10ms,19인 경우 5ms: 50%감소
- 공평하지 않은 time slice할당
- Time slice가 timer tick의 배수이어야 한다는 제약이 있음

nice 값에 따른 타임 슬라이스 계산 (O(1))

- Nice 값에 따른 정적 우선순위와 Time slice
  - 정적 우선 순위가 120보다 작은 경우 :
    Time slice = (140 – 정적 우선순위) * 20
  - 정적 우선 순위가 120보다 크거나 같은 경우 :
    Time slice = (140 – 정적 우선순위) * 5

설명	Nice 값	정적 우선순위	타임슬라이스 길이
최저 우선순위	+19	139	5 ms
낮은 수선순위	+10	130	50 ms
기본 우선순위	0	120	100 ms
높은 우선순위	−10	110	600 ms
최고 우선순위	−20	100	800 ms

nice 값에 따른 타임 슬라이스 값 (O(1))

nice 값	정적 우선순위	타임 슬라이스	Nice 값	정적 우선순위	타임 슬라이스
-20	100	800 ms	0	120	100 ms
-19	101	780 ms	1	121	95 ms
-18	102	760 ms	2	122	90 ms
-17	103	740 ms	3	123	85 ms
-16	104	720 ms	4	124	80 ms
-15	105	700 ms	5	124	75 ms
-14	106	680 ms	6	126	70 ms
-13	107	660 ms	7	127	65 ms
-12	108	640 ms	8	128	60 ms
-11	109	620 ms	9	129	55 ms
-10	110	600 ms	10	130	50 ms
-9	111	580 ms	11	131	45 ms
-8	112	560 ms	12	132	40 ms
-7	113	640 ms	13	133	35 ms
-6	114	520 ms	14	134	30 ms
-5	115	500 ms	15	135	25 ms
-4	116	480 ms	16	136	20 ms
-3	117	460 ms	17	137	15 ms
-2	118	440 ms	18	139	10 ms
-1	119	420 ms	19	140	5 ms

CFS(Completely Fair) 스케쥴링

- 모든 프로세스를 공평하게 실행하는 것이 목표
- 우선 순위 별로 서로 다른 가중치 사용
  - 고정 크기의 타임 슬라이스 대신 동적 크기의 타임 슬라이스 할당
- 태스크뿐만 아니라 사용자나 그룹에 공평하게 스케줄러를 조율 가능
- 태스크를 함께 그룹으로 묶어 놓으면, 스케줄러는 이런 엔티티와 엔티티에 속하는 태스크에 공평하게 동작
- CONFIG_FAIR_GROUP_SCHED를 선택 필요
  - 현재까지는 SCHED_NORMAL과 SCHED_BATCH 태스크만 그룹으로 묶을 수 있다.
- 3D 게임을 비롯해 스레드를 많이 사용하는 응용 프로그램 테스트 과정에서 좋은 평가를 받고 있다

CFS 스케쥴링과 타임 스라이스 할당(예)

- CPU time slice 할당예제
  - sched_period = 10ms

5개의 Processes 가 아래와 같은 경우

nice	weight	Wp/Wt	time slice(ms)
−10	9548	0.675	6.75
−5	3121	0.220	2.20
0	1024	0.072	0.72
5	335	0.024	0.24
10	110	0.008	0.08
total	14138	1.000	10.00

3개의 Processes 가 아래와 같은 경우

nice	weight	Wp/Wt	time slice(ms)
−10	9548	0.697	6.97
−5	3121	0.228	2.28
0	1024	0.075	0.75
total	13693	1.000	10.00

Wp/Wt = weight/total
Time slice = 10ms * Wp/Wt

리눅스의 Preemptive RT(Real-Time)

- 리눅스 커널의 **Preemptive Real-Time (RT)** 기능
  - 실시간 시스템에서 요구하는 엄격한 응답 시간을 보장하기 위해 도입된 기능
  - 커널이 특정 작업을 중단하고 더 높은 우선순위의 작업을 즉시 수행할 수 있도록 함
- **Preemptive RT의 필요성**
  - **실시간 응답성**: 실시간 시스템에서는 정해진 시간 내에 작업을 완료해야 하는 엄격한 요구사항 필수. Preemptive RT는 이러한 요구사항을 충족시키기 위해 사용
  - **우선순위 기반 스케줄링**: 높은 우선순위의 작업이 낮은 우선순위의 작업에 의해 방해받지 않고 즉시 실행될 수 있도록 지원
  - 작업이 낮은 우선순위의 작업에 의해 방해받지 않고 즉시 실행될 수 있도록 합니다.
- **Preemptive RT 주요 개념**
  - **Preemption (선점)**: 현재 실행 중인 작업을 중단하고 더 높은 우선순위의 작업을 실행, 시스템은 더 긴급한 작업을 신속하게 처리 가능
  - **Real-Time (RT) 커널 패치**: 리눅스 커널의 기본 스케줄러는 완전한 실시간 성능을 제공하지 못함. 이를 보완하기 위해 RT 패치가 적용 됨. 이 패치는 커널의 선점성을 강화하고, 실시간 스케줄링 알고리즘을 추가

리눅스 Preemptive RT 주요 기능

- **커널 선점성 강화**
  - **완전한 선점성**: RT 커널 패치는 커널의 모든 코드 경로에서 선점을 허용하여, 커널 모드에서 실행 중인 작업도 중단 가능. 이는 긴 커널 경로에서도 실시간 응답성 보장
  - **락(lock) 기법 개선**: 실시간 성능을 저해할 수 있는 스핀락(spinlock)을 대체하기 위해 잠금 기법이 개선. 주로 선점 가능한 락과 우선순위 상속 프로토콜(priority inheritance protocol) 사용
- **스케줄링 알고리즘**
  - **SCHED_FIFO**: 고정 우선순위 스케줄링 방식으로, 가장 높은 우선순위의 작업이 완료될 때까지 실행
  - **SCHED_RR**: 라운드 로빈 방식의 고정 우선순위 스케줄링으로, 같은 우선순위를 가진 작업들이 일정 시간 동안 번갈아 가며 실행
  - **SCHED_DEADLINE**: 작업의 데드라인을 기반으로 스케줄링을 수행하여, 주어진 시간 내에 작업이 완료되도록 보장
- **인터럽트 선점**
  - **하드웨어 인터럽트 처리**: 실시간 시스템에서는 빠른 인터럽트 처리가 필수적입니다. RT 커널은 인터럽트 핸들러를 빠르게 처리하고, 높은 우선순위의 인터럽트가 낮은 우선순위의 인터럽트를 선점 가능
  - **소프트웨어 인터럽트 처리**: 소프트웨어 인터럽트도 선점 가능하게 처리하여, 실시간 작업의 지연 최소화

리눅스 Preemptive RT 장점과 한계

- **Preemptive RT의 장점**
  - **향상된 응답성**: 실시간 작업의 응답 시간이 크게 향상
  - **예측 가능한 성능**: 시스템의 성능 예측 가능. 실시간 애플리케이션의 신뢰성 향상

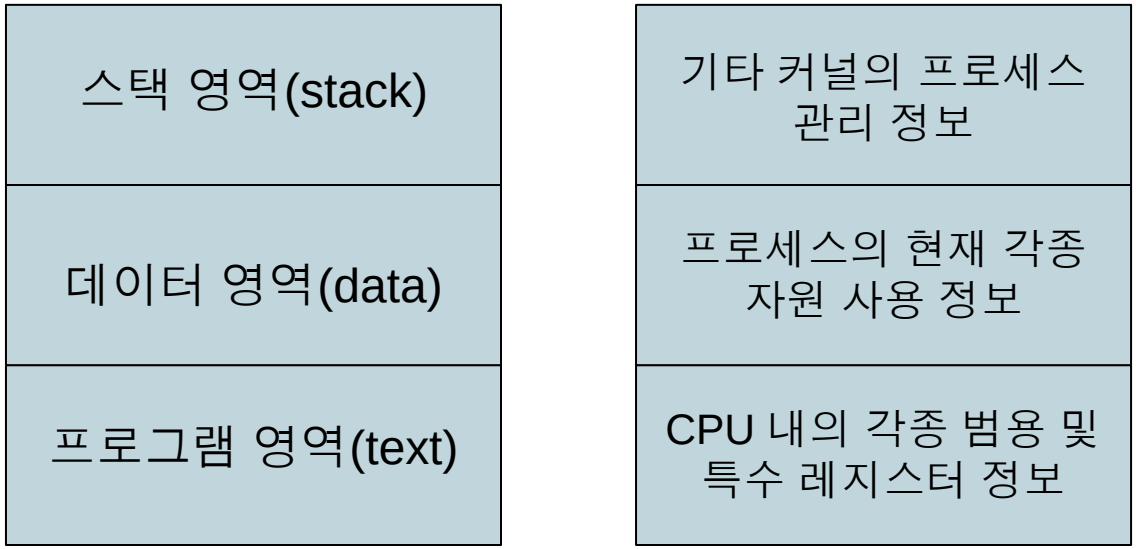
- **Preemptive RT의 한계**
  - **복잡성 증가**: RT 기능을 구현하고 유지보수하는 데 있어 복잡성이 증가
  - **오버헤드**: 선점성과 실시간 기능을 제공하기 위해 추가적인 시스템 오버헤드가 발생 가능

리눅스의 문맥 교환

- 문맥 교환(Context switching)
  - 프로세스가 변경될 때, 프로세스가 잠들 때(sleep), exit 할 때, system call, 인터럽트나 exception 처리를 하고 복귀할 때 프로그램이 사용하는 메모리 정보, CPU의 레지스터 정보 등을 저장하거나 복원 하는 것
- 문맥(Context) 정보
  - “task_struct” 구조체 정보
    - 프로그램의 메모리 사용
    - CPU 레지스터 등
- 프로세스 switching
  - “kernel/sched.c” 파일의 “context_switch()” 함수에 구현
  - 프로세스 변경을 위해 “switch_to()” 함수를 호출하면 “arch/arm/kernel/entry-armv.S”의 “__switch_to” 함수 실행

문맥(Context)

- 문맥(Context)
  - 프로세스의 실행 중단 시 보존되고, 재개 시 다시 원상 복구되어야 하는 프로세스의 실행을 위한 모든 정보를 말한다



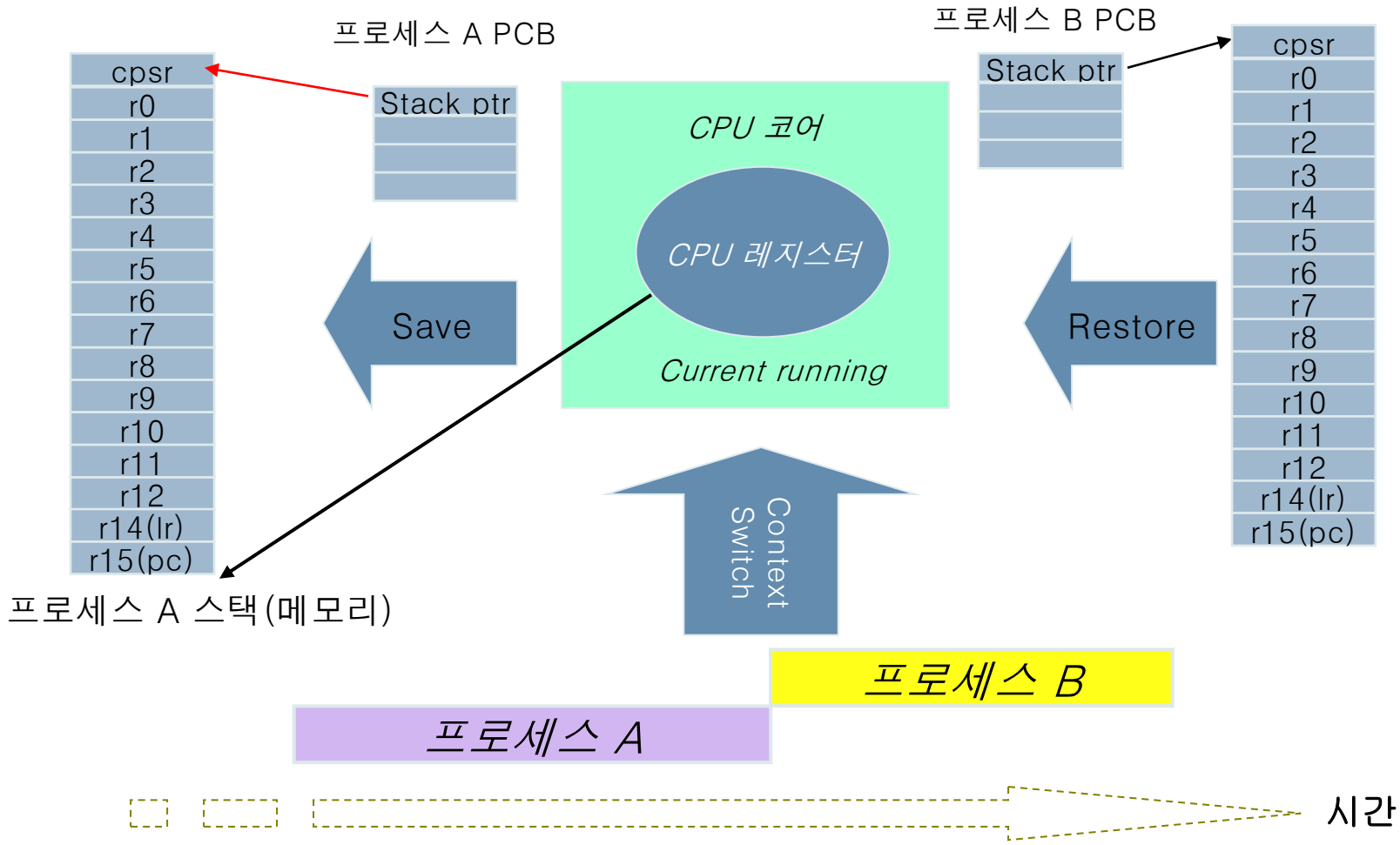
사용자 영역의 문맥 **커널 영역의 문맥**

문맥 교환(Context Switching)

- 문맥 교환(Context Switching)
  - 스케줄러는 문맥 교환이 필요한 경우 디스패처(Dispatcher)에 문맥 교환 처리를 요청
  - 문맥 교환
    - CPU가 하나의 프로세스를 처리하다 다른 프로세스를 처리 하도록 교환된다
      1. 이전 프로세스의 PCB(모든 CPU 정보와 메모리 상태 정보)를 저장
      2. 새로운 프로세스의 PCB(모든 CPU 정보와 메모리 상태 정보)를 로딩
    - 이전 프로세스에서 사용되던 데이터의 주소 공간에 대한 메모리 동작 발생
      - 메모리 동작을 줄이려면 쓰레드 사용

- 디스패처(Dispatcher)
  - 문맥 교환 처리
  - 사용자 모드로 전환
  - 프로그램의 적절한 위치로 점프하여 프로그램 재 시작
    - dispatch latency가 짧아야 함

문맥 교환과 레지스터


프로세스 A PCB
cpsr
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r14(lr)
r15(pc)
Stack ptr
프로세스 A 스택(메모리)
Save
CPU 코어
CPU 레지스터
Current running
Context Switch
Restore
프로세스 B PCB
Stack ptr
cpsr
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r14(lr)
r15(pc)
프로세스 B
프로세스 A
시간

프로세스 정보 (task_struct)

멤버	내용
상태 정보(state)	프로세스의 상태를 표시한다.
우선순위 정보	프로세스의 우선순위 정보를 가진다.
스케쥴링 정보	스케쥴링 정보를 가진다.
메모리 관리 정보	메모리를 관리하기 위한 정보를 가진다.
프로세스 ID	프로세스의 사용자 또는 그룹 ID(Identifier) 정보를 표시한다.
IPC 정보	파이프(pipe) 및 세마포어(semaphores)와 같은 프로세스간 통신 정보를 표시한다
링크(link) 정보	부모(parent) 프로세스와 자식(child) 프로세스 정보를 표시한다.
시간 정보	프로세스의 생성 시간 및 동작 된 시간 정보를 표시한다.
파일 시스템 및 오픈 파일 정보	VFS(Virtual File System)의 아이노드(inode) 형태로 파일 시스템 정보를 표시
입출력 장치 정보	입출력 장치 리스트 및 프로세스 내 할당 정보를 표시한다.
프로세서(CPU)의 문맥정보	프로세서의 레지스터, 스택 등 정보를 표시한다.
시그날 정보	시그날을 관리하기 위한 정보를 가진다.

프로세스의 상태

- Running (TASK_RUNNING)
  - 프로세스가 실행 중이거나 실행 할 준비가 완료된 상태
- Waiting (TASK_INTERRUPTABLE)
  - 프로세스가 대기모드로 이벤트를 기다리는 상태
- Stopped (TASK_STOPPED)
  - 프로세스가 signal에 의하여 정지 되어 있는 상태
- Zombie (TASK_ZOMBIE)
  - 프로세스가 비정상적으로 halt 되어 있는 상태
  - 보통 “dead process”라고 부른다.

프로세스 생성

- 프로세스 생성을 위한 시스템 콜
  - 경량급 프로세스 생성을 위한 clone() 시스템 콜
  - 전통적인 프로세스 생성 방법 fork() 시스템 콜
  - 빠른 프로세스 생성을 위한 vfork() 시스템 콜
- 프로세스 생성
  - 시스템 콜 clone(), fork(), vfork()가 호출 되면 서로 다른 인자로 do_fork() 함수를 호출
  - 호출된 요청에 따라 do_fork() 함수는 자식 프로세스 생성하고 프로세스 ID 반환
  - fork()를 호출한 프로세스는 부모(parent) 프로세스가 되고 새로운 프로세스는 자식(child) 프로세스가 된다.

fork(), vfork() 그리고 wait() 함수

- fork() 시스템 콜
  - 새로운 프로세스를 생성할 때 부모 프로세스의 정보를 자식 프로세스의 메모리 공간에 복사
  - fork()의 반환 값
    - -1 : 에러
    - 0 : 자식 프로세스
    - 다른 값 : 부모 프로세스
- vfork() 시스템 콜
  - 부모 프로세스의 text(코드) 와 data 세그먼트를 복사 하지 않는다.
- wait() 함수
  - 부모 프로세스가 자식 프로세스의 동작 상태 검사하여 좀비(Zombie)가 발생하지 않도록 한다.

fork()와 exec()의 만남

- exec() 함수
  - 호출된 새로운 프로그램으로 프로세스를 교체하여 실행
  - execl(), execv(), execve(), …..

- fork()와 exec()를 사용한 프로세스 생성
  1. 현재 프로세스에서 fork() 호출
  2. fork()로 부터 0 값이 반환 되면 exec()를 호출하여 새로운 프로그램으로 변경
  3. 부모 프로세스는 wait()를 호출하여 자식 프로세스의 종료를 기다리고, 자식 프로세스가 종료 되었다는 정보를 받으면 프로세스 종료

fork()와 exec()를 이용한 프로세스 생성

```
#include <unistd.h>
int fatal(char *s);
main()
{
        pid_t pid;
        switch (pid = fork()){
        case -1:
                fatal("fork failed.");
                break;
        case 0:  /* 자식 프로세스 */
                execl("/bin/ls","ls","-1",(char *)0)); /* 자식이 exec를 호출 */
                fatal("exec failed.");
                break;
        default: /* 부모 프로세스 */
                wait((int *)0);  /* 부모는 자식이 종료 될 때 까지 대기 */
                printf("ls completed\n");
                exit(0);
         }
}
```

```
int fatal(char *s)
{
        perror(s);
        exit(1);
}
```

프로세스 종료

- exit() 시스템 호출
  - 프로세스 처리를 끝내고 나서 자진하여 종료
  - 프로세스 종료 시
    - 부모에 보고 (pid, 상태...), 자원 반환

- abort(), kill() 함수
  - 다른 프로세스(부모)에 의해 강제적으로 종료
    - 자식의 자원 과다 사용
    - 자식의 임무 완성
    - 부모의 종료

쓰레드(Thread)

- 쓰레드
  - 프로세스 또는 프로그램에서 실행되는 또 다른 프로세스
  - 프로세스와 메모리 공간을 공유
- 쓰레드 종류
  - 사용자 프로세스에서 생성된 사용자 쓰레드(User Thread)
  - 커널에서 생성된 커널 쓰레드(Kernel Thread)

커널 쓰레드(Thread)

- 커널 쓰레드
  - 커널 모드에서 시스템 관리를 위한 백그라운드 동작
  - 디스크 캐시 저장, 페이지 수왑 아웃, 네트워크 서비스 관리
- 커널 쓰레드의 생성
  - kernel_thread() 함수의 호출에 의해서 생성
- 프로세스 0
  - 모든 프로세서의 조상이 되며 스와퍼(Swapper) 프로세스 라고도 한다.
  - 커널 초기화 과정을 담당하는 start_kernel() 함수가 리눅스를 초기화 하는 동안 생성
  - 새로운 커널 쓰레드인 프로세스 1로 알려진 init 프로세스를 생성한다
    - 프로세스 ID(PID) 1이 된다

리눅스의 타이머

- 리눅스 타이머의 주요 동작
  - 시스템의 시작부터 경과한 시간을 갱신
  - 현재 날짜와 시간을 갱신
  - Asynchronous 스케쥴링
    - 현재 프로세스가 얼마나 오랫동안 CPU를 점유하고 있는지를 검사하고 할당 시간이 넘으면 프로세스를 선점
  - 각종 시스템 자원의 사용의 통계를 갱신
  - 소프트웨어 타이머로 사용
    - 시간 간격 측정, 딜레이 동작 등
- 타임 간격
  - 타임 간격을 주기로 인터럽트 발생
  - Centi-second (10ms) 간격

리눅스 타이머의 활용

- 변수 jiffies
  - 시스템이 시작할 때 부터 지나간 틱수를 나타내는 변수로 커널 초기화 때 0 으로 설정하고 타이머 인터럽트가 발생할 때마다 1씩 증가
- 타이머의 활용
  - 변수 jiffies와 원하는 틱(tick)수를 계산하여 실행
  - 장치 드라이버의 예외적인 상황 감지
    - KEY 입력 검사 등
  - 프로그래머가 특정 시각에 함수를 실행 하고자 할 때
    - settimer(), alarm() 등
  - 기타

목 차

메모리와 주소(Address)

- 메모리
  - 프로그램과 데이터를 저장하는 공간
  - 메모리의 위치를 지정하기 위해서는 주소(Address) 사용

- 메모리 주소
  - 메모리 셀의 내용에 대한 접근 위치 지정
  - 일반적으로 물리 주소를 칭하는 경우가 많음

리눅스의 주소(Address)

- 논리 주소(Logical Address)
  - 실제 CPU가 기계어 상에서 사용되는 주소
- 선형 주소(Linear Address)
  - CPU가 사용하는 4GB 영역까지의 억세스 가능한 주소
  - 가상(Virtual)적으로 할당하여 프로그램에서 사용하는 주소
- 물리 주소(Physical Address)
  - 메모리 칩의 메모리 셀의 위치를 지정
  - 어드레스 버스로 전달되는 전기적인 신호

리눅스의 메모리 관리

- 큰 주소 공간 사용(Large Address Space)
  - 실제 가지고 있는 메모리 보다 큰 메모리 공간을 가지고 있는 것처럼 사용할 수 있게 해 준다.
- 메모리 보호(Memory Protection)
  - 각 프로세스는 각각의 가상 메모리를 가지고 사용한다.
  - 가상 메모리를 사용하는 경우에는 자신 이외의 다른 프로세스는 동일한 메모리 공간을 참조 할 수 없어 메모리가 완전히 독립되어 보호된다.
- 메모리 매핑(Memory Mapping)
  - Text 와 데이터를 프로세스의 가상 주소 공간으로 주소를 맵핑하여 사용
- 공정한 물리 메모리(Physical Memory) 할당
  - 공유 메모리를 여러 프로세스가 공정하게 사용할 수 있도록 한다.
- 공유(Shared) Virtual 메모리
  - 프로세스 간에 메모리 공유를 위한 방법을 제공

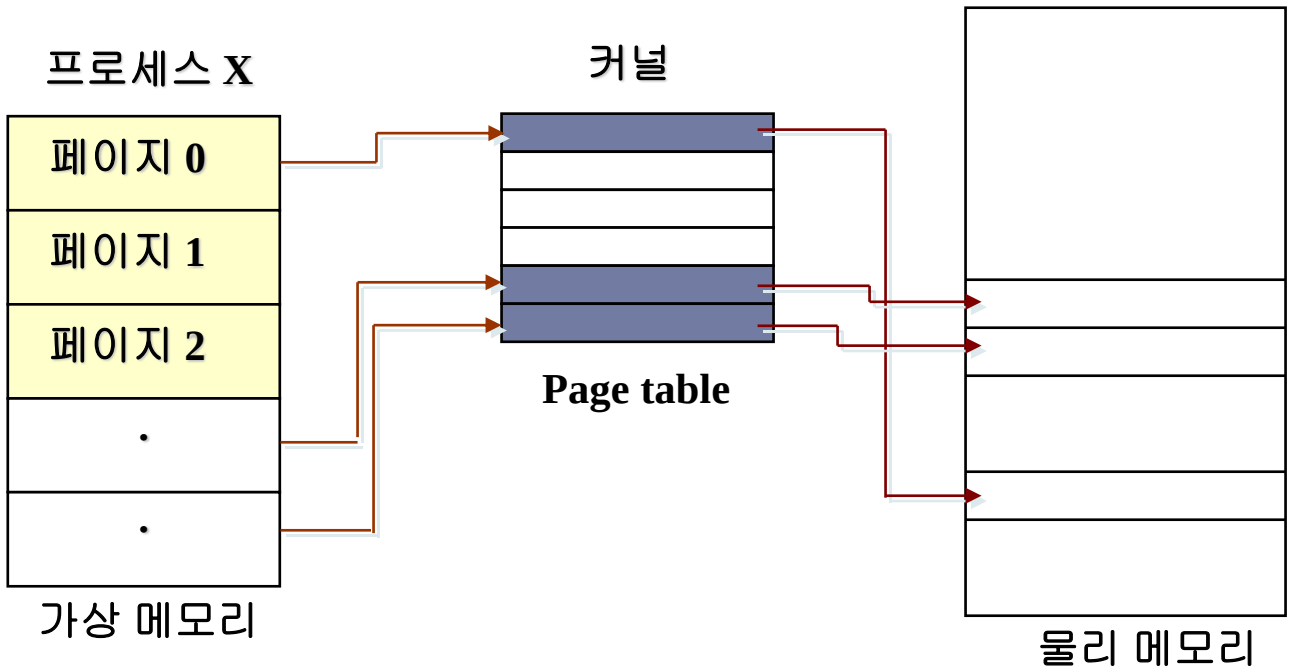
프로세스의 메모리 사용

- 프로세스 별 개별적인 주소 공간 사용
  - 개별적으로 코드, 데이터 및 스택 영역을 할당하여 사용
  - 메모리 디스크립터
    - 프로세스 디스크립터(task_struct)의 mm 필드에 저장되어 프로세스에서 사용하는 메모리 정보를 표시한다.
    - start_code, end_code, start_data, end_data, arg_start, ….
- 함수 malloc()
  - 프로세스는 동적 영역(heap) 확장하여 메모리를 할당하여 사용 가능
- 프로세스간 공유 메모리 사용 가능
- 시스템 콜 mmap() 지원
  - 파일의 일부나 장치의 메모리를 프로세스의 주소 공간의 일부로 매핑하여 사용

주소 공간의 조각화

- 세그먼테이션과 페이지로 주소 공간을 쪼갠다
- 세그먼테이션
  - 각각의 프로세스에 다른 선형 주소 공간 할당
  - 프로그램을 코드, 데이터, 스택 같은 논리적인 부분으로 쪼갠다.
- 페이지
  - 효율적인 주소 관리를 위하여 똑같은 선형 주소 공간을 일정한 크기로 쪼개고, 다른 물리 주소 공간에 매핑하여 사용
  - 페이지 프레임(page frame) 또는 물리적인 페이지(physical page)
    - 고정된 길이의 메모리 : ARM의 경우 4KB
    - 각 페이지 프레임에 페이지가 하나씩 할당
    - 각 페이지는 고유의 번호를 부여 받는다

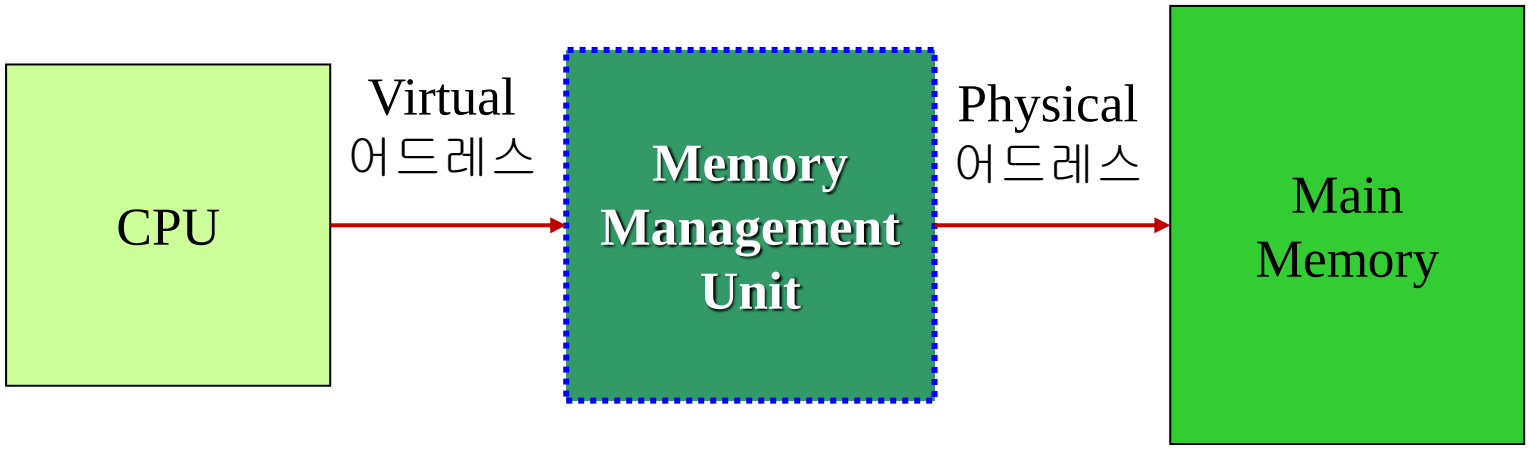
주소 변환



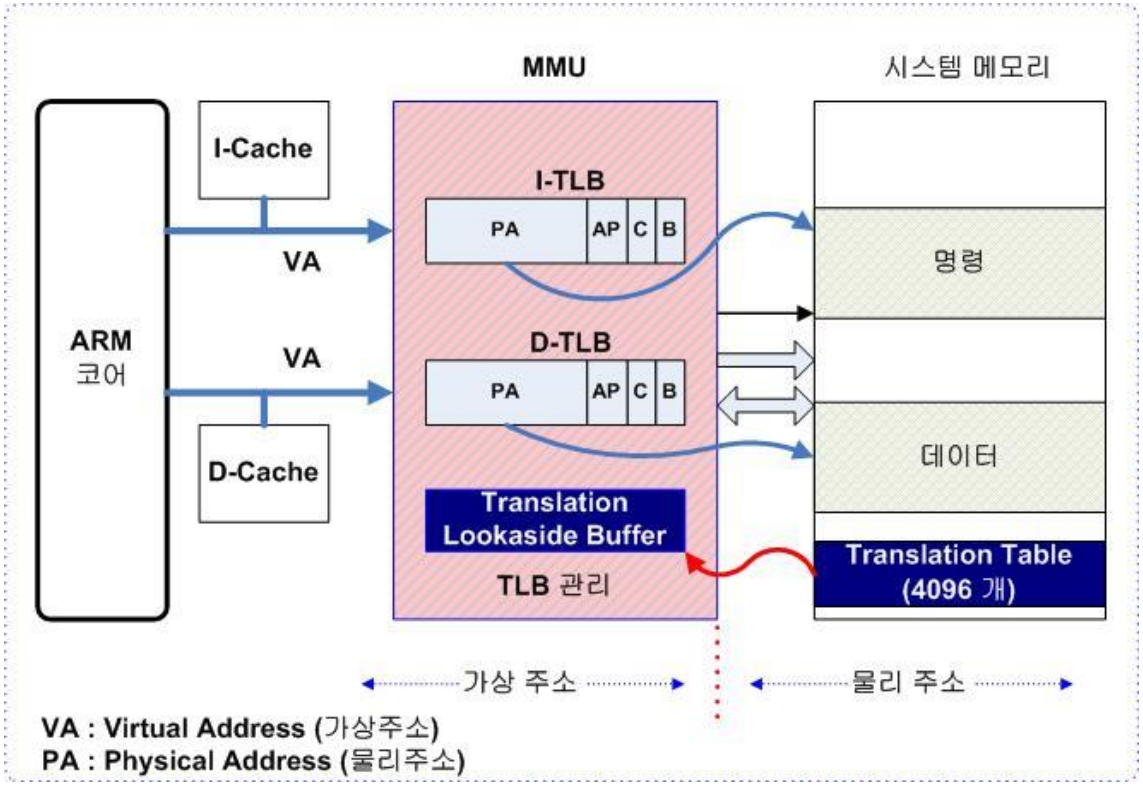
가상 주소를 물리 주소로 쉽고 빠르게 하기 위하여 캐시와 유사한 구조의 고속 메모리 장치 사용, ARM의 TLB(Translation Lookaside Buffer)가 존재

MMU (Memory Management Units)

- 메모리 보호(protection) 기능
- 어드레스 변환(translation) 기능
  - CPU에서 사용되는 logical 한 Virtual 어드레스를 physical 어드레스로 변환



MMU와 주소변환


MMU
시스템 메모리
I-Cache
I-TLB
PA
AP
C
B
VA
명령
ARM
코어
VA
D-TLB
PA
AP
C
B
D-Cache
데이터
Translation
Lookaside Buffer
Translation Table
(4096 개)
TLB 관리
가상 주소
물리 주소
VA : Virtual Address (가상주소)
PA : Physical Address (물리주소)

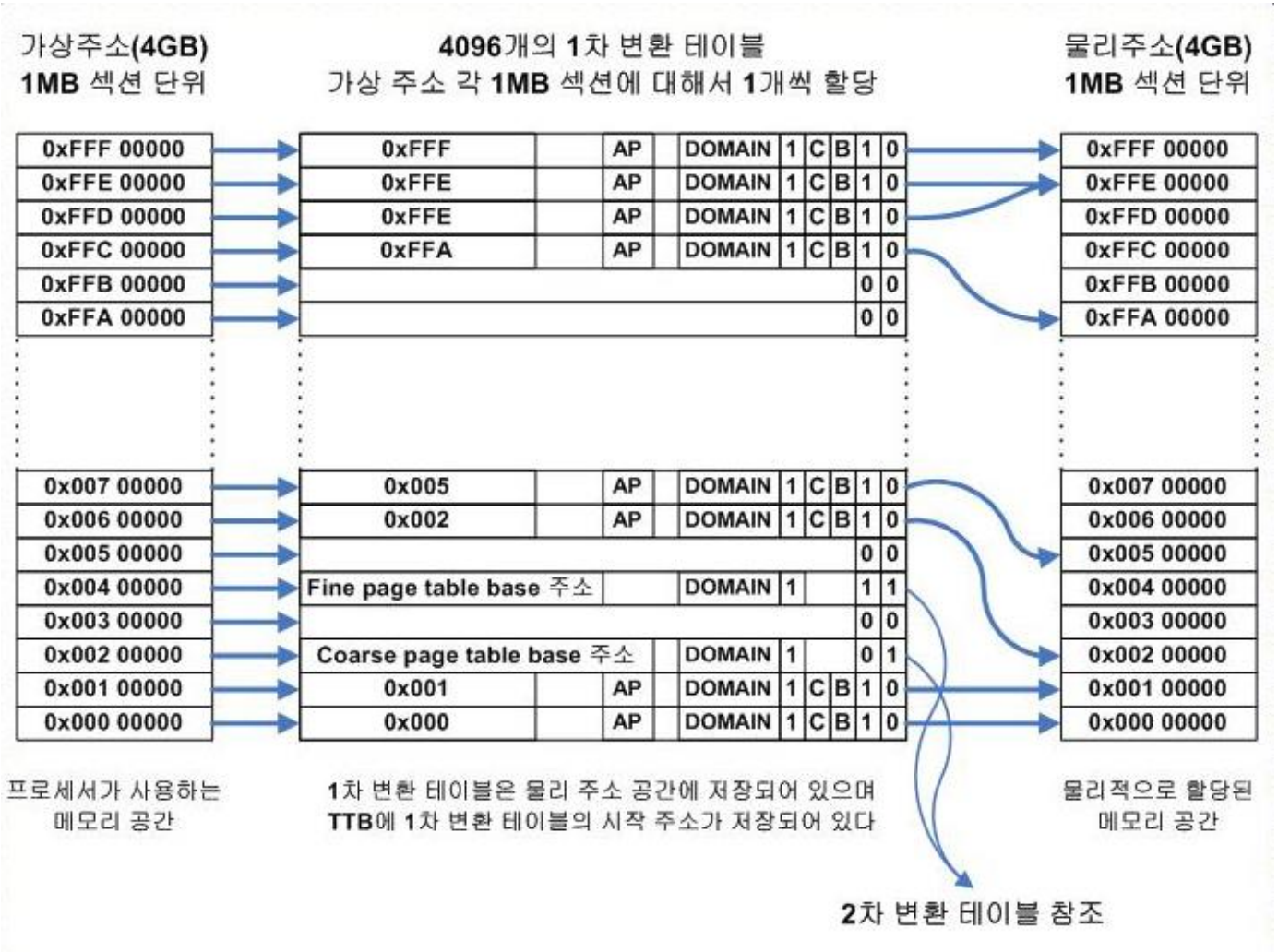
변환 테이블(Translation Table)

- Physical 메모리에 있는 translation 정보를 가지고 있는 Table
- Level 1 Translation Table
  - 4096개의 32비트 translation table entry
    - 4GB 메모리를 virtual address 1MB 단위로 나누어 관리
    - Virtual address 비트 [31:20]로 정렬
  - Physical memory에 대한 1MB section 단위 또는 16MB super section 단위의 address translation 정보와 access control 정보를 가지거나, 레벨 2 table에 대한 주소 정보를 가진다.
- Level 2 Translation Table
  - 64KB(large page), 4KB(small page) 단위의 translation table을 정보를 가지고 있다.
  - 각 translation table에는 address translation 정보와 access control 정보를 가진다.

Translation Lookaside Buffer (TLB)

- 최근에 사용된 Virtual address를 physical address로 변화하는 정보와 access permission에 대한 정보를 저장하고 있는 일종의 Cache
- TLB가 Virtual 어드레스에 대한 translation table entry를 가지고 있으면 access control logic이 access 가능을 판단
  - 접근이 허용되면 virtual address를 physical address로 변환 후 access
  - 접근의 허용이 않 되면 CPU에 Abort 구동
- TLB에 virtual 어드레스에 대한 정보가 없으면 translation table walking logic에서 table 정보를 physical 메모리에서 읽어 TLB update

변환 테이블과 어드레스 변환



캐시와 쓰기 버퍼 제어

- Section 또는 page 별로 캐시와 쓰기 버퍼의 사용여부 결정
  - Cacheable
    - Page 내의 데이터가 Cache될 수 있음을 나타낸다
  - Bufferable
    - Page 내의 데이터가 write buffer에 write될 수 있음을 나타낸다.
  - Memory mapped I/O 장치의 경우에는 반드시 disable 되어 있어야 한다.
- Cacheable 과 Bufferable에 의한 메모리 시스템 특징

C	B	의 미	Cache의 Write 동작
0	0	Cache 불가, 쓰기 버퍼 불가	
0	1	Cache 불가, 쓰기 버퍼 동작	
1	0	Cache 동작, 쓰기 버퍼 불가	Write-through Cache
1	1	Cache 동작, 쓰기 버퍼 동작	Write-back Cache

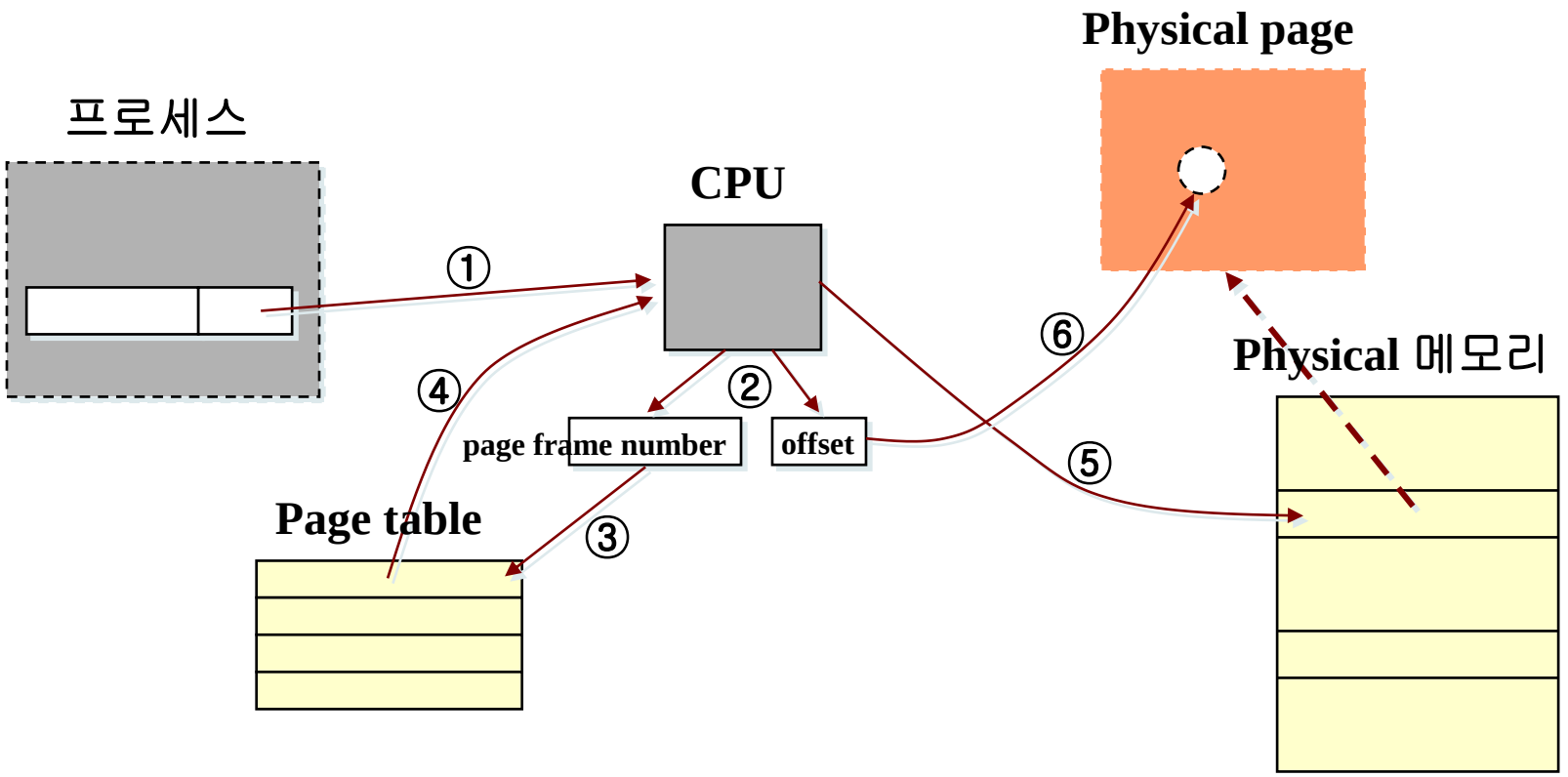
접근 권한(Access Permission)

- Section 또는 page 별로 메모리의 access permission 제한
  - Access 권한은 각 descriptor의 AP 정보와 S(System) 비트와 R(Rom) 비트에 의해서 제어
    - S, R비트는 control 레지스터의 비트 8과 9이다
  - Access가 불가하면 permission fault가 발생

AP[2]	AP[1:0]	접근 권한	
		Privileged	Un-privileged
0	00	No Access	No Access
0	01	Read/Write	No Access
0	10	Read/Write	Read Only
0	11	Read/Write	Read/Write
1	00	Reserved	
1	01	Read Only	No Access
1	10	Read Only	Read Only
1	11	Read Only	Read Only

프로세스와 메모리 접근

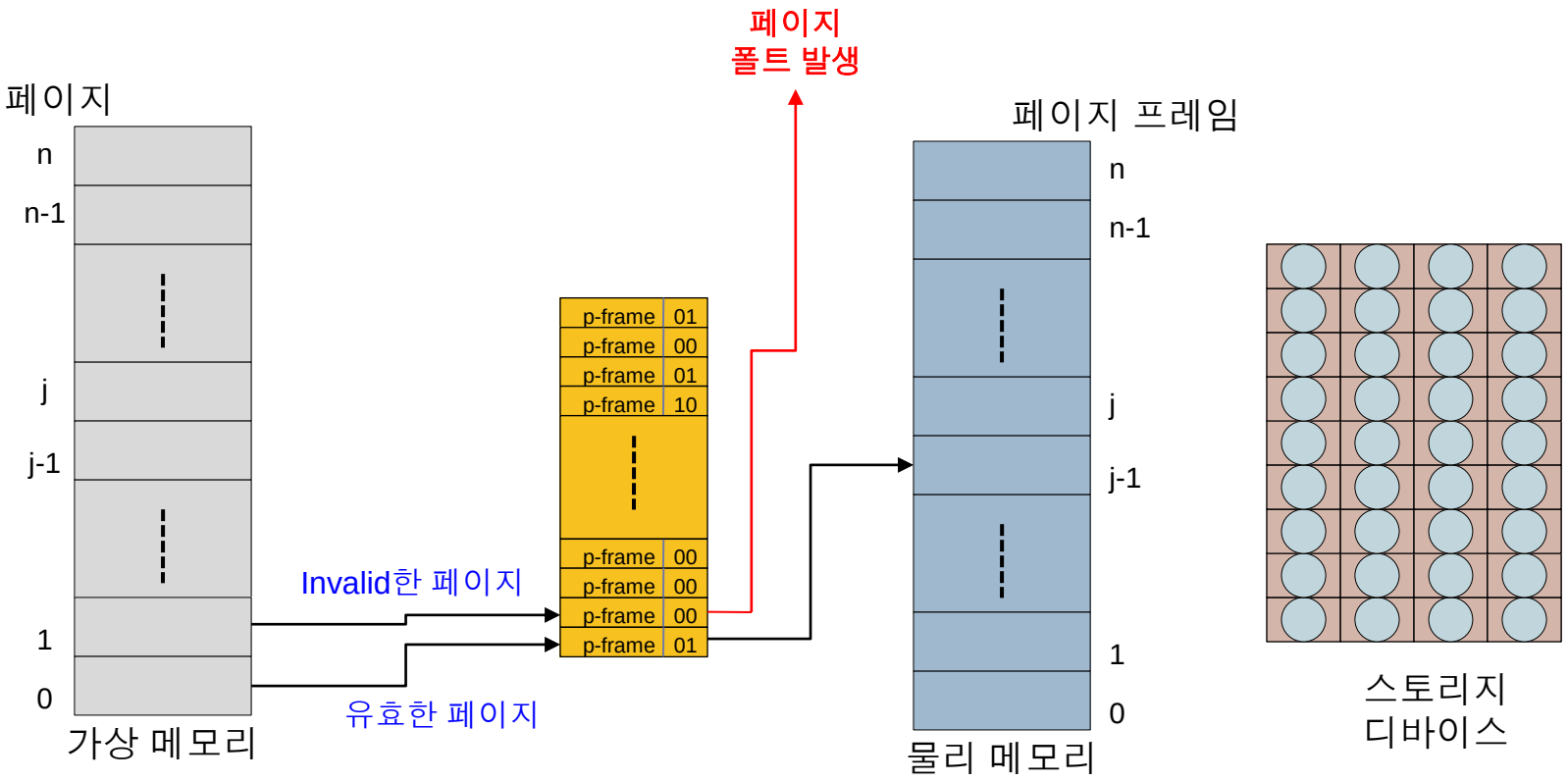
1. 프로세스에서 메모리 접근 (①)
2. 페이지 프레임 번호를 가지고 페이지 테이블에서 정보를 가져온다 (②, ③, ④)
3. 물리 메모리의 해당 페이지에서 offset위치를 접근한다(⑤ ,⑥)



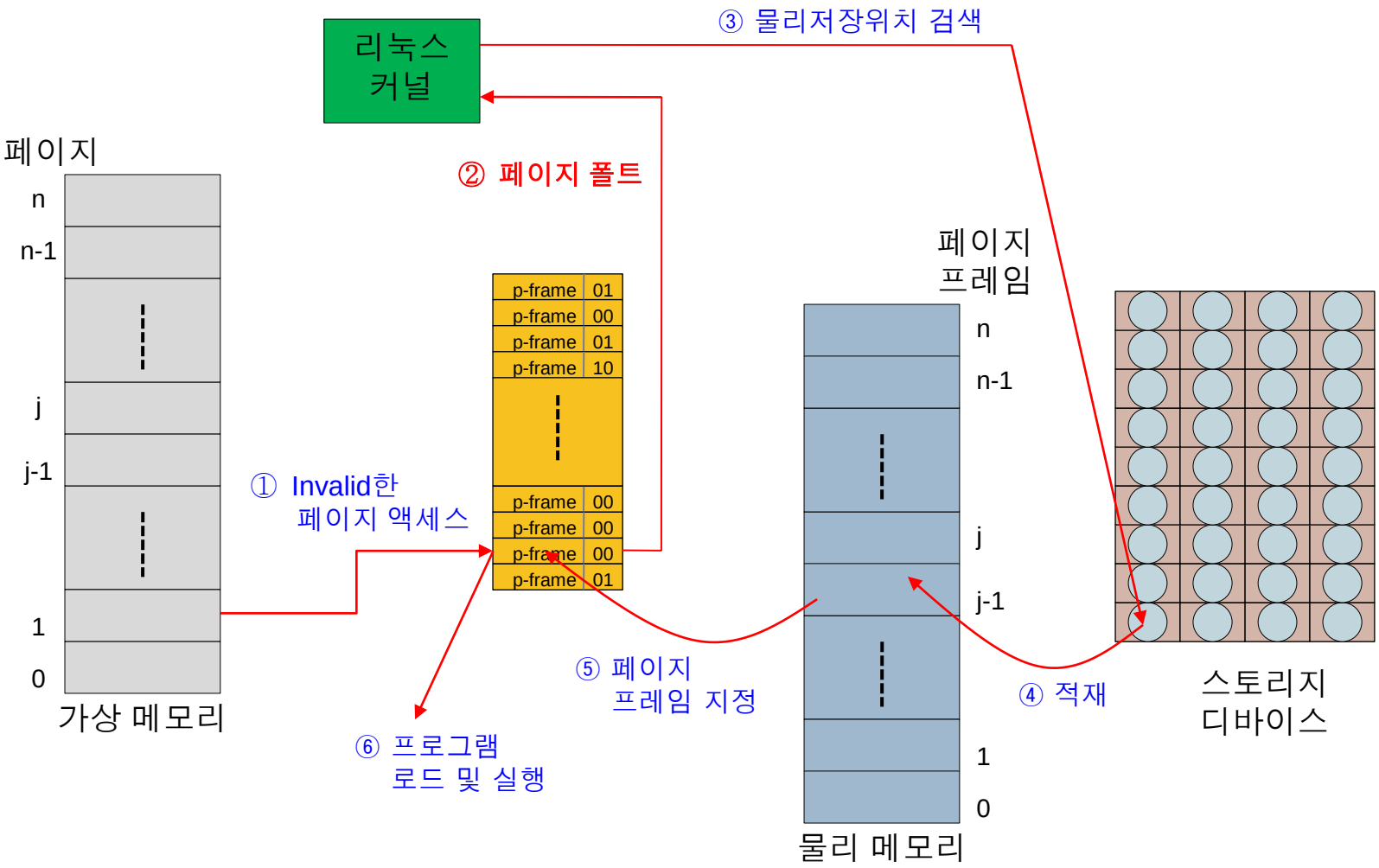
Demand Paging

- Demand Paging
  - 프로세스가 시작 하면서 모든 페이지 프레임을 할당하여 사용하지 않고, 일부 페이지 프레임만 할당하여 사용하다 새로운 프레임의 요구가 있을 때 새로운 페이지 프레임을 할당하여 사용
- Demand Paging 원리
  - 프로세스가 처음부터 자신의 메모리 공간에 있는 모든 영역을 접근하지 않는다
  - 지역성 원리(Locality Principle)
    - 프로그램은 실행되는 각 단계에서 프로세스 페이지의 일부만 사용
- 장점
  - 똑같은 메모리의 양으로 더 많은 작업을 처리할 수 있다

페이지 폴트 발생


페이지
폴트 발생
페이지
n
n-1
j
j-1
1
0
가상 메모리
Invalid한 페이지
유효한 페이지
p-frame 01
p-frame 00
p-frame 01
p-frame 10
p-frame 00
p-frame 00
p-frame 00
p-frame 01
페이지 프레임
n
n-1
j
j-1
1
0
물리 메모리
스토리지
디바이스

페이지 폴트 처리


리눅스
커널
③ 물리저장위치 검색
② 페이지 폴트
페이지
n
n-1
j
j-1
1
0
가상 메모리
① Invalid한
페이지 액세스
p-frame 01
p-frame 00
p-frame 01
p-frame 10
p-frame 00
p-frame 00
p-frame 00
p-frame 01
⑤ 페이지
프레임 지정
⑥ 프로그램
로드 및 실행
페이지
프레임
n
n-1
j
j-1
1
0
물리 메모리
④ 적재
스토리지
디바이스

Copy On Write (COW)

- Copy on Write (COW)
  - 자식 프로세스를 생성할 때 모든 페이지 프레임을 복사하지 않고 부모 프로세스와 자식 프로세스 간에 페이지 프레임을 공유하고 있다가 부모나 자식 프로세스가 공유된 페이지 프레임에 쓸(Write) 때 페이지 프레임을 복사(Copy)하여 새로운 페이지 프레임을 만드는 방식

- COW의 장점
  - 메모리 접근을 줄이고 CPU의 사용량을 줄일 수 있다

프로세스 생성과 Copy-on-Write

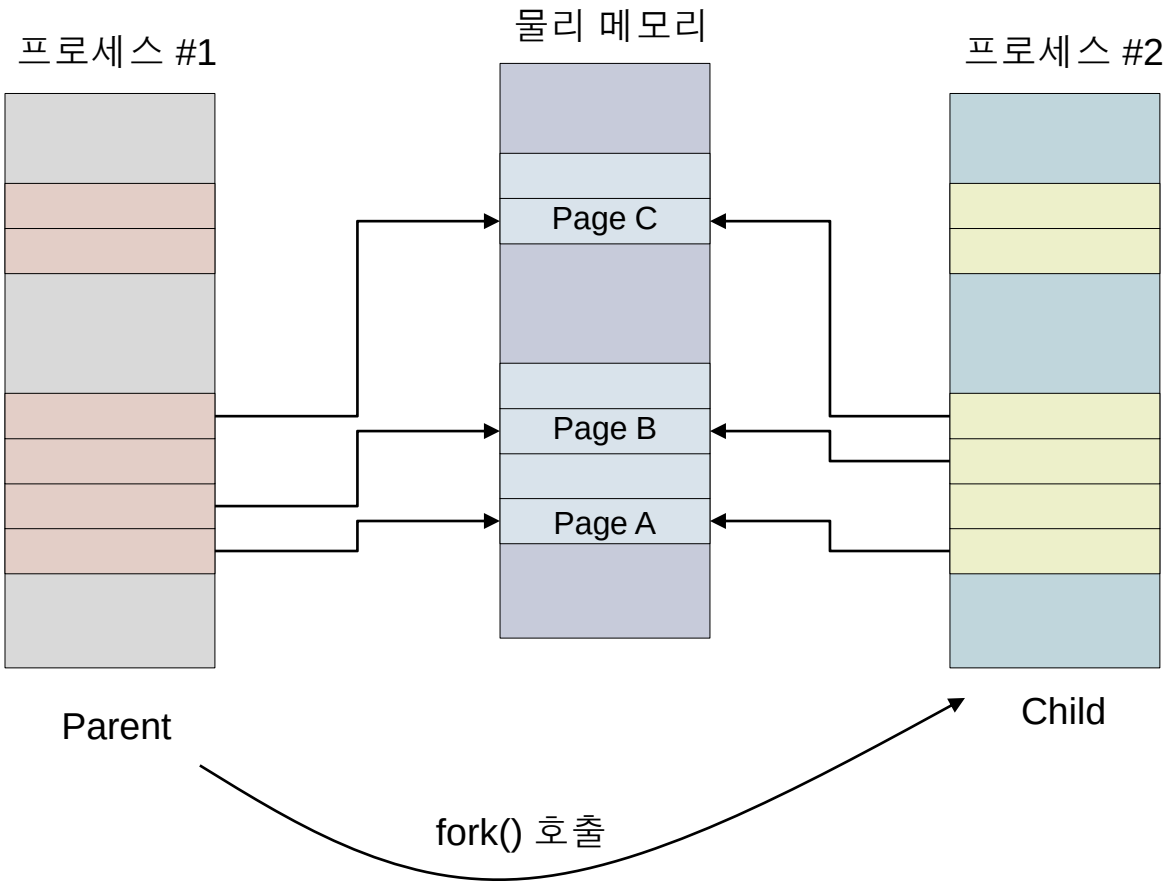
- 프로세스를 생성할 때 fork()을 일반적으로 사용
- fork()를 호출 후 동작
  - 부모 프로세스의 페이지들을 자식 프로세스에 복사
  - 자식 프로세스는 주소 공간에 exec()를 호출하여 프로그램 교체 후 실행
    - 그러면 부모로부터 복사해 온 페이지들은 다 쓸모 없는 것들이 된다.
    - 불필요한 복사 동작 발생

- fork()와 Copy-on-Write
  - 부모 프로세스는 작식 프로세스를 생성하고, 부모 페이지들을 다 복사해오는 대신 자식 프로세스가 시작할 때 부모의 페이지를 당분간 공유하여 사용
    - 공유되는 페이지를 COW 페이지라고 표시
  - 둘 중 한 프로세스가 공유중인 페이지에 쓸 때 그 페이지의 복사본이 만들어 진다

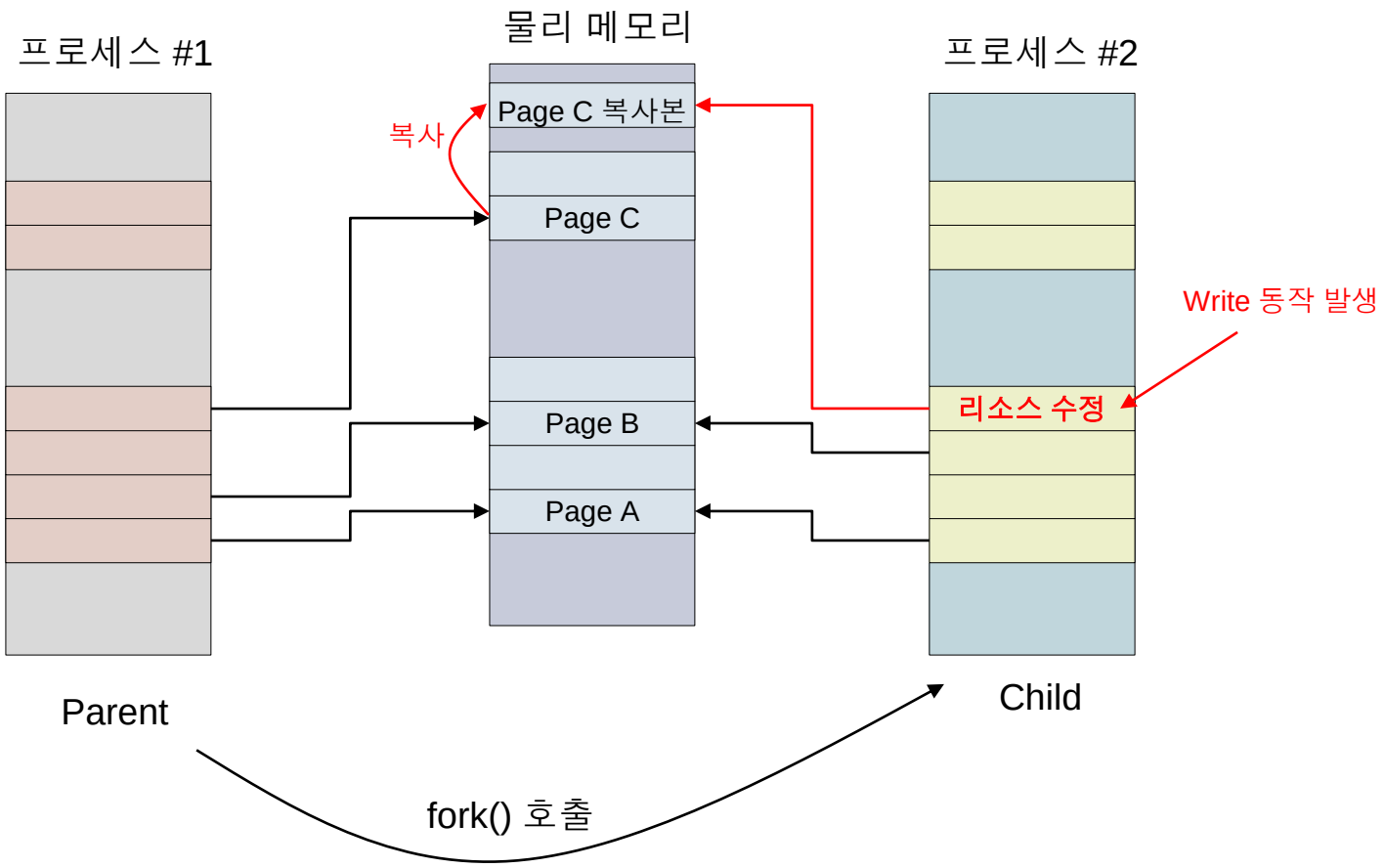
디맨드 페이징과 Copy-on-Write

- 모든 프로세스는 디맨드 페이징 기법 사용
- fork()를 호출하여 새로운 프로세스를 생성할 때 COW를 사용하면 리소스를 공유하고 있기 때문에 디맨드 페이징 동작도 불필요
- 새로운 프로세스 생성을 매우 빠르게 할 수 있고, 새롭게 할당되어야 하는 페이지의 수도 최소화 할 수 있다.

메모리 수정 전



메모리 수정 후

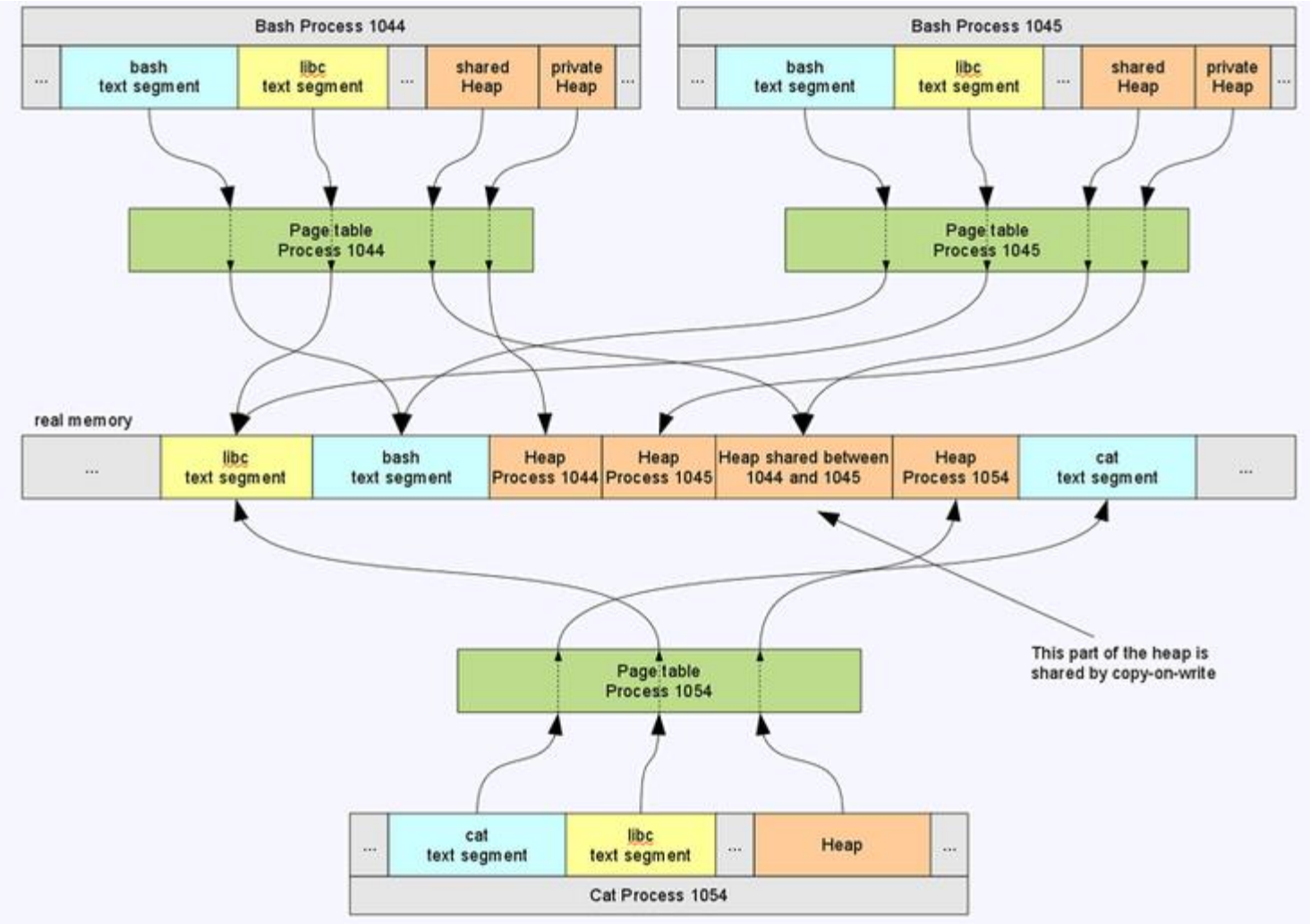

프로세스 #1
물리 메모리
프로세스 #2
Page C 복사본
복사
Page C
Write 동작 발생
리소스 수정
Page B
Page A
Parent
Child
fork() 호출

쓰기 동작과 페이지 할당

- 페이지 할당
  - 쓰기 시 복사 처리 과정에서 페이지 복사본을 만들 때 빈 페이지를 할당
  - 운영체제에서 요구를 처리하기 위해 빈 페이지 집합(pool)을 가진다.
  - zero-fill-on-demand 기법으로 페이지를 할당
    - 페이지를 할당할 때 그 내용을 영(0)으로 채워 이전 내용을 지운다

- 빈 페이지가 프로세스에게 할당 되는 경우
  - 쓰기 시 복사(Copy-on-Write)를 할 때
  - 스택이나 힙 공간을 확장해야 할 때

Copy-on-Write와 메모리 사용



스와핑(Swapping)

- 디스크의 일부 공간을 램의 확장으로 사용
- 스와핑을 사용하는 목적
  - 프로세스가 실제로 사용할 수 있는 주소공간 확장
  - 프로세스를 적재할 동적 램 공간 확장
- 스왑 아웃과 스왑 인
  - 스왑 아웃(Swap Out)
    - 커널이 빈 메모리를 계속 감사 하다가 고정된 값보다 적어지면, 프로세스 주소 공간을 디스크에 복사한다.
  - 스왑 인(Swap In)
    - 스케쥴링에서 스왑 아웃된 프로세스를 선택하면 프로세스를 디스크로 부터 읽어온다.
- 스와핑 단위
  - 프로세스 단위가 아닌 작은 페이지 단위로 스와핑

스왑 아웃

- 커널 쓰레드 kswapd
  - 빈 페이지 프레임이 미리 정의한 기준치 보다 작아질 때 마다 1초 간격으로 스왑 아웃 수행

- 스왑 아웃되는 페이지
  - 가장 많은 페이지를 소유한 프로세스의 페이지를 회수
  - LRU(Least Recently Used)
    - 가장 오랜 시간 동안 사용 하지 않는 페이지를 스왑 아웃

페이지의 스왑 아웃과 폐기

- 스왑 아웃 또는 폐기될 페이지 검색
  - kswapd는 시스템에 있는 각 프로세스를 차례로 관찰하면서, 어떤 프로세스가 스왑하기 좋은 후보 인지 판단
  - 스왑하기 좋은 후보는 메모리 에서 스왑되거나 폐기될 수 있는 페이지를 하나 이상 가진 프로세스 들이다.
  - 스왑 할 프로세스를 결정하면 그 프로세스의 가상 메모리 영역을 전부 보면서 공유되거나 락이 걸리지 않은 영역을 찾는다.
    - 선택된 프로세스의 스왑 가능 한 페이지를 모두 스왑 아웃하지 않고 페이지 몇 개만 제거
    - 메모리에 락되어 있는 페이지는 스왑하거나 폐기할 수 없다.
  - 스왑이 불가능한 프로세스도 있다.
- 스왑 아웃되는 페이지
  - 페이지 안에 저장된 데이터를 다른 방법으로 얻어올 수 있는 방법이 없을 때만, 물리적 메모리로부터 시스템의 스왑 파일에 스왑 아웃된다.
- 폐기되는 페이지
  - 페이지 내용을 쉽게 다시 읽을 수 있는 경우는 폐기
  - 실행 이미지의 상당수는 실행 파일에서 가져온 것이며 파일에서 쉽게 다시 읽을 수 있다.
    - 예로 이미지에 들어있는 실행 명령은 변경되지 않기 때문에 스왑 파일에 쓸 필요 가 없다.
    - 이들 페이지는 무조건 폐기하고, 프로세스가 이들을 다시 참조할 때, 실행 이미지에서 메모리에 다시 가져오게 된다.

스왑 파티션을 사용하지 않는 경우

- 스왑 파티션이 없는 경우 swap 이 동작하지 않는다
  - 하지만 공간 확보를 위해 페이지 제거 동작은 처리된다.
  - 제거되는 페이지
    - 코드, RO 데이터 등
  - RW 데이터는 제거되지 않는다.

- 대부분 임베디드 리눅스는 스왑 파티션을 사용하지 않는 경우가 많다

목 차

예외처리 (Exception)

- 예외처리(Exception)
  - 외부의 요청이나 오류에 의해서 정상적으로 진행되는 프로그램의 동작을 잠시 멈추고 프로세서의 동작 모드를 변환하고 미리 정해진 프로 그램을 이용하여 외부의 요청이나 오류에 대한 처리를 하도록 하는 것
  - Exception의 예
    - I/O 장치에서 인터럽트를 발생시키면 IRQ Exception이 발생하고, 프로세서는 발생된 IRQ Exception을 처리하기 위해 IRQ 모드로 전환되어 요청된 인터럽트에 맞는 처리 동작 수행
- Exception 종류
  - Reset
  - Undefined Instruction
  - Software Interrupt(SWI)
  - Secure Monitor(SMC)
  - Prefetch Abort
  - Data Abort
  - IRQ(Interrupt Request)
  - FIQ(Fast Interrupt Request)
  - Breakpoint Instruction

Exception Vector 와 우선순위

- Exception Vector
  - Exception이 발생하면 미리 정해진 어드레스의 프로그램을 수행
  - 미리 정해진 프로그램의 위치를 Exception Vector라 한다.
- Exception Vector Table
  - 발생 가능한 각각의 Exception에 대하여 Vector를 정의해 놓은 테이블
    - 각 Exception 별로 1 word 크기의 명령어 저장 공간을 가진다.
  - Vector Table에는 Branch 또는 이와 유사한 명령으로 실제 Exception을 처리하기 위한 루틴으로 분기 할 수 있는 명령으로 구성되어 있다.
    - FIQ의 경우는 Vector Table의 맨 상위에 위치하여 분기명령 없이 처리루틴을 프로그램 할 수 있다.
  - ARM은 기본적으로 0x00000000에 Vector Table을 둔다.
    (MMU 제어 프로그램에 의해 위치 변경 가능 : 0xFFFF0000)
- Exception 우선 순위
  - 동시에 Exception이 발생하는 경우 처리를 위해 우선 순위 지정

Exception Vector Table의 명령어

- Exception 마다 1 word 크기의 명령어 저장 공간 할당
  - Exception vector 테이블에서는 1 개의 ARM 명령만을 사용할 수 있다.
    - 실제 Exception Handler가 있는 분기 명령으로 만들어 진다.
  - FIQ의 Vector Table은 맨 상위에 있으므로 핸들러를 직접 작성할 수 있다.
- Exception Vector Table에서 사용할 수 있는 명령어
  - Branch 명령 (B)
    - 가장 일반적으로 사용된다.
    - Branch 명령은 PC 값을 기준으로 +/− 32MB 내에 있어야 한다.
      - Handler가 32MB 영역을 벗어나면 다른 명령을 사용하여야 한다.
  - Move 명령 (MOV)
    - Destination 레지스터를 PC로 하면 Branch 명령과 같이 사용 가능
    - 한 사이클 내에 처리된다.
    - Handler 어드레스가 8비트 상수와 ROR로 표시 가능해야 사용 가능하다.
  - Load 명령(LDR)과 Literal Pool
    - 메모리 영역 내의 어떤 위치라도 이동이 가능하다.
    - 메모리에서 주소를 읽기 위한 1 사이클이 더 필요하다.

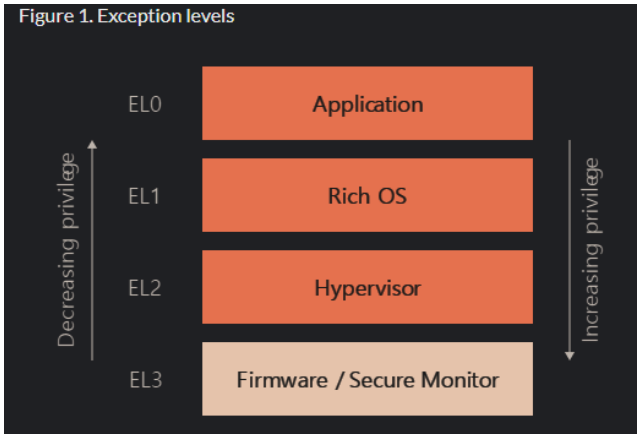
ARM32 Exception Vector Table구성 예

Vector Address	Exception	우선순위	동작모드 전환
0x0000 0000	Reset	1 (High)	Supervisor(SVC)
0x0000 0004	Undefined Instruction	6 (Low)	Undefined
0x0000 0008	SWI / SMC	6	Supervisor(SVC)
0x0000 000C	Prefetch Abort / BKPT	5	Abort
0x0000 0010	Data Abort	2	Abort
0x0000 0014	Reserved		
0x0000 0018	IRQ	4	IRQ
0x0000 001C	FIQ	3	FIQ

ARM64 Exception

- ARM64 (ARM Architecture v8) Exception
  - https://developer.arm.com/documentation/102412/0103?lang=en

- ARM64 (ARM Architecture v8) Level



- Synchronous & Asynchronous exceptions
  - Synchronous exceptions
    - Exceptions that can be caused by, or are related to, the instruction that is currently being executed
  - Asynchronous exceptions
    - Some types of exceptions are generated externally and therefore are not synchronous with the current instruction stream.
    - Asynchronous exceptions are also known as interrupts.

ARM64 Linux Exception Vector Table

- arch/arm64/kernel/entry.S

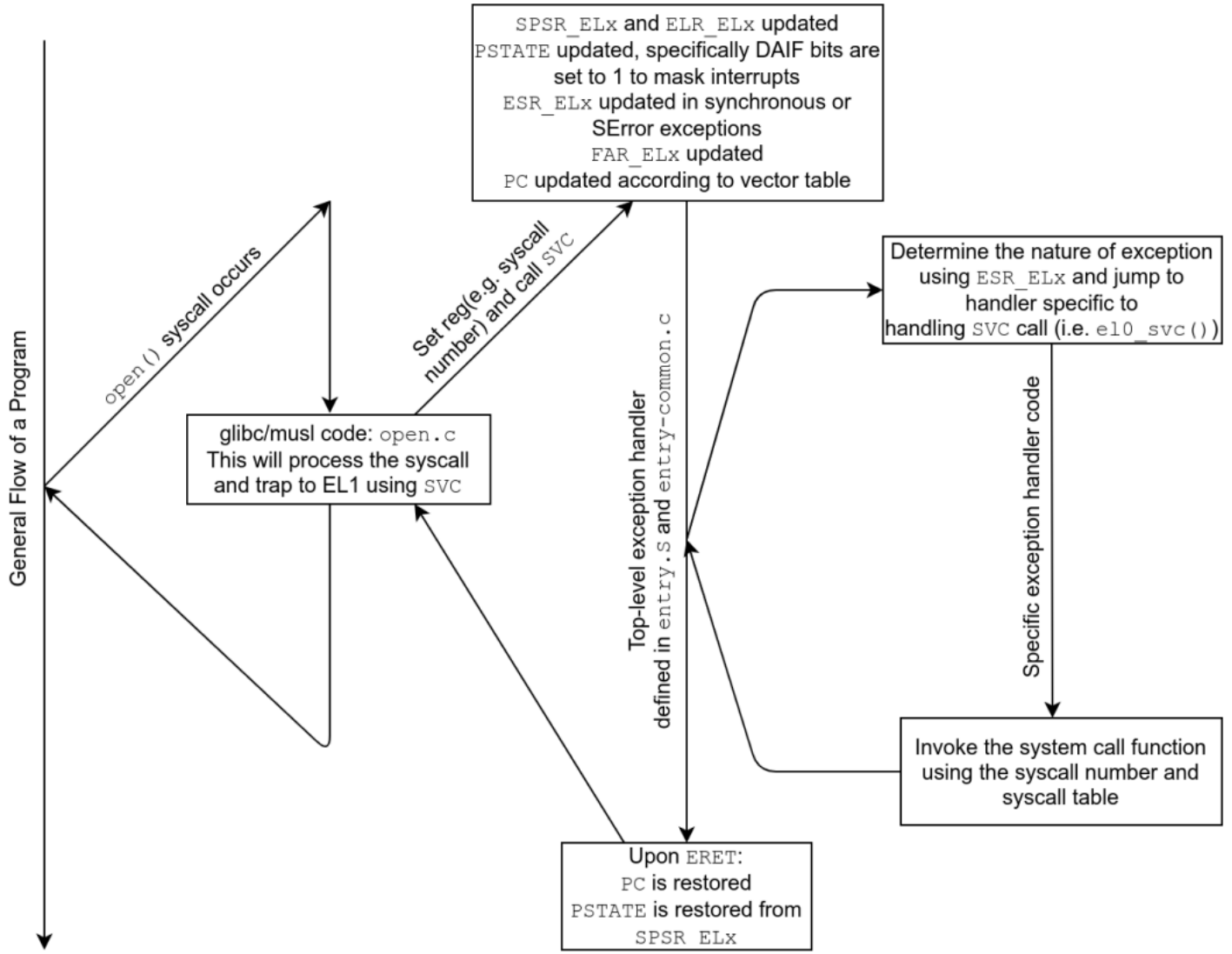
```
551 /*
552  * Exception vectors.
553  */
554     .pushsection ".entry.text", "ax"
555
556     .align  11
557 SYM_CODE_START(vectors)
558     kernel_ventry   1, sync_invalid         // Synchronous EL1t
559     kernel_ventry   1, irq_invalid          // IRQ EL1t
560     kernel_ventry   1, fiq_invalid          // FIQ EL1t
561     kernel_ventry   1, error_invalid        // Error EL1t
562
563     kernel_ventry   1, sync             // Synchronous EL1h
564     kernel_ventry   1, irq              // IRQ EL1h
565     kernel_ventry   1, fiq_invalid          // FIQ EL1h
566     kernel_ventry   1, error            // Error EL1h
567
568     kernel_ventry   0, sync             // Synchronous 64-bit EL0
569     kernel_ventry   0, irq              // IRQ 64-bit EL0
570     kernel_ventry   0, fiq_invalid          // FIQ 64-bit EL0
571     kernel_ventry   0, error            // Error 64-bit EL0
572
573 #ifdef CONFIG_COMPAT
574     kernel_ventry   0, sync_compat, 32      // Synchronous 32-bit EL0
575     kernel_ventry   0, irq_compat, 32       // IRQ 32-bit EL0
576     kernel_ventry   0, fiq_invalid_compat, 32   // FIQ 32-bit EL0
577     kernel_ventry   0, error_compat, 32     // Error 32-bit EL0
578 #else
579     kernel_ventry   0, sync_invalid, 32     // Synchronous 32-bit EL0
580     kernel_ventry   0, irq_invalid, 32      // IRQ 32-bit EL0
581     kernel_ventry   0, fiq_invalid, 32      // FIQ 32-bit EL0
582     kernel_ventry   0, error_invalid, 32        // Error 32-bit EL0
583 #endif
584 SYM_CODE_END(vectors)
```

- arch/arm64/kernel/head.S

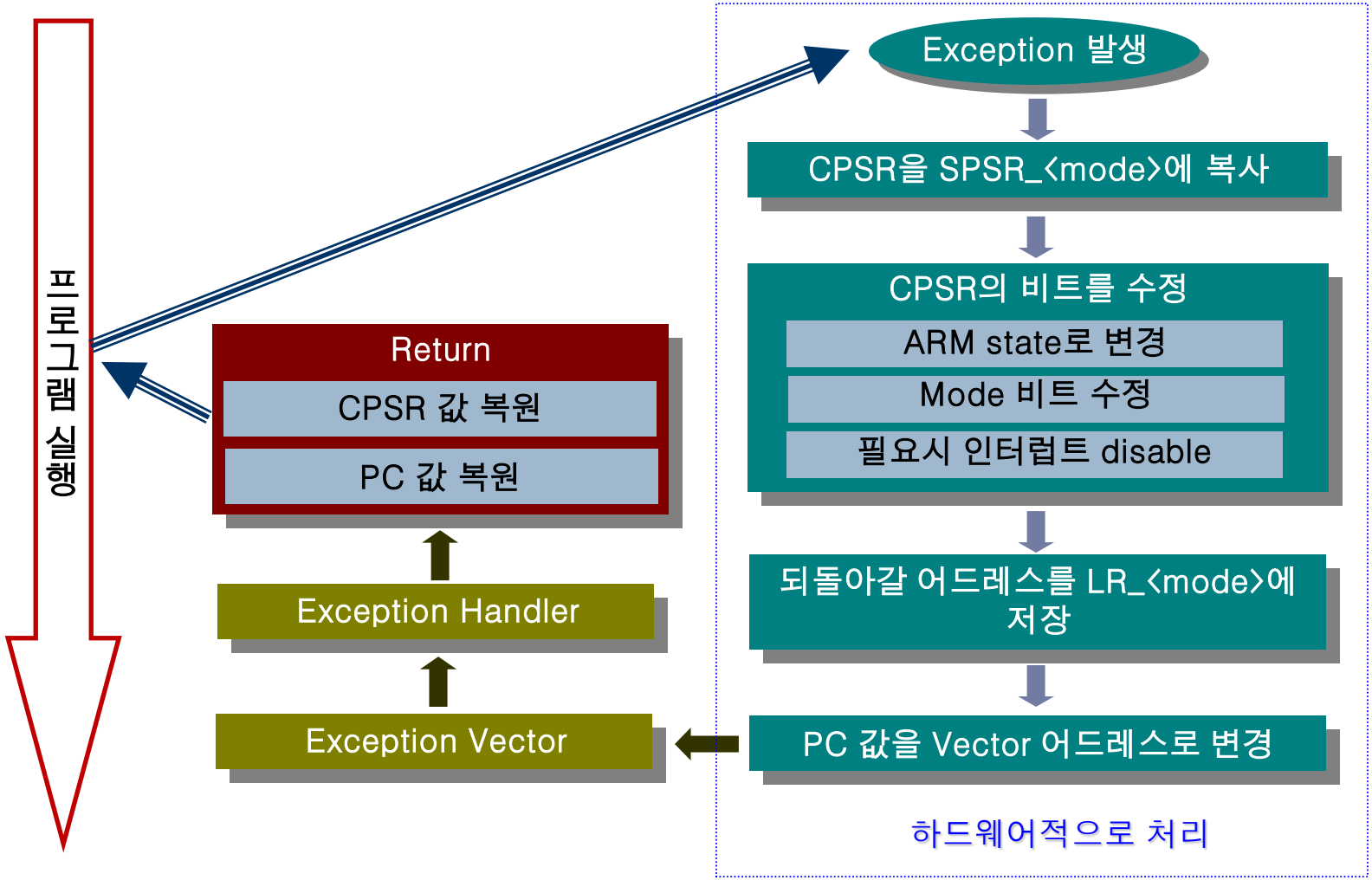
```
417 SYM_FUNC_START_LOCAL(__primary_switched)
418     adrp    x4, init_thread_union
419     add sp, x4, #THREAD_SIZE
420     adr_l   x5, init_task
421     msr sp_el0, x5          // Save thread_info
~~~

427     adr_l   x8, vectors         // load VBAR_EL1 with virtual
428     msr vbar_el1, x8            // vector table address
429     isb
430
431     stp xzr, x30, [sp, #-16]!
432     mov x29, sp
```

General ExceptionHanfling



Exception Handling



인터럽트(Interrupt)

- 인터럽트(Interrupt)
  - 주기적으로 발생되는 타이머(Interval Timer) 인터럽트, 키보드를 누를 때와 같이 입출력 장치에서 정상적인 처리를 요청

- 예외처리와 인터럽트 벡터(Vector)
  - 예외 처리나 인터럽트는 사용자의 프로그램에 의해서 발생할 수도 있으나 대부분은 하드웨어적인 요인으로 발생
    - 보통은 Abort, IRQ와 같은 하드웨어 핀을 가진다
  - 예외 처리나 인터럽트가 발생하면 처리해야 할 프로그램(인터럽트 핸들러:Interrupt Handler)이 저장되어 있는 위치

I/O 자원 관리와 인터럽트(Interrupt)

- I/O 자원 관리
  - 폴링(Polling), 인터럽트(Interrupt), DMA(Direct Memory Access)
  - 인터럽트는 I/O 자원을 관리하는 방법 중에서 가장 효과적이고 멀티 프로세스 지원을 위한 필수 요건
- 인터럽트 제어기
  - 여러 입출력 장치에서 발생되는 인터럽트 관리
  - 인터럽트 펜딩 레지스터에 발생된 인터럽트 정보를 저장

인터럽트 처리

- 인터럽트 발생
  - 입출력 장치에서 CPU에게 처리할 준비가 완료 되었거나 처리가 완료 되었음을 알린다.
- 인터럽트 핸들러
  - 발생된 인터럽트를 처리한다
  - 대부분 발생된 인터럽트 정보는 인터럽트 제어기의 펜딩 레지스터에 있고, 인터럽트 처리 전에 해당하는 펜딩 비트를 클리어 한다.
  - 하드웨어 인터럽트 처리 루틴은 되도록 짧게 한다
- 인터럽트 핸들러의 분리
  - Top half
    - 하드웨어 레벨의 인터럽트 핸들러
    - do_IRQ() 함수에서 처리
  - Bottom half
    - 소프트웨어 레벨의 인터럽트 핸들러로 안정된 시점에 실행된다.
    - 현재는 개념적으로 이 용어를 사용하고 있다

ARM64(ARMv8) Linux IRQ Exception Handler(1)

- arch/arm64/kernel/entry.S

```
676 SYM_CODE_START_LOCAL_NOALIGN(el1_irq)
677     kernel_entry 1
678     el1_interrupt_handler handle_arch_irq
679     kernel_exit 1
680 SYM_CODE_END(el1_irq)

~~~


715 SYM_CODE_START_LOCAL_NOALIGN(el0_irq)
716     kernel_entry 0
717 el0_irq_naked:
718     el0_interrupt_handler handle_arch_irq
719     b   ret_to_user
720 SYM_CODE_END(el0_irq)
```

①

```
512     .macro el1_interrupt_handler, handler:req
513     enable_da_f
514
515     mov x0, sp
516     bl  enter_el1_irq_or_nmi
517
518     irq_handler \handler
519
520 #ifdef CONFIG_PREEMPTION
521     ldr x24, [tsk, #TSK_TI_PREEMPT] // get preempt count
522 alternative_if ARM64_HAS_IRQ_PRIO_MASKING
523     /*
524      * DA_F were cleared at start of handling. If anything is set in DAIF,
525      * we come back from an NMI, so skip preemption
526      */
527     mrs x0, daif
528     orr x24, x24, x0
529 alternative_else_nop_endif
530     cbnz    x24, 1f             // preempt count != 0 || NMI return path
531     bl  arm64_preempt_schedule_irq  // irq en/disable is done inside
532 1:
533 #endif
534
535     mov x0, sp
536     bl  exit_el1_irq_or_nmi
537     .endm
```

②

- arch/arm64/kernel/entry-common.c
  - enter_el1_irq_or_nmi

```
96 asmlinkage void noinstr enter_el1_irq_or_nmi(struct pt_regs *regs)
97 {
98     if (IS_ENABLED(CONFIG_ARM64_PSEUDO_NMI) && !interrupts_enabled(regs))
99         arm64_enter_nmi(regs);
100    else
101        enter_from_kernel_mode(regs);
102 }
```

ARM64(ARMv8) Linux IRQ Exception Handler(2)

- arch/arm64/configs/wn9_defconfig (.config)

```
CONFIG_GENERIC_IRQ_MULTI_HANDLER=y
```

- kernel/irq/handle.c
  - set_handle_irq

```
21 #ifdef CONFIG_GENERIC_IRQ_MULTI_HANDLER                    ③
22 void (*handle_arch_irq)(struct pt_regs *) __ro_after_init;
23 #endif

~~~


220 #ifdef CONFIG_GENERIC_IRQ_MULTI_HANDLER
221 int __init set_handle_irq(void (*handle_irq)(struct pt_regs *))
222 {
223     if (handle_arch_irq)
224         return -EBUSY;
225
226     handle_arch_irq = handle_irq;
227     return 0;
228 }
229 #endif
```

- drivers/irqchip/irq-gic-v3.c
  - gic_init_base

```
1717 static int __init gic_init_bases(void __iomem *dist_base,
1718                  struct redist_region *rdist_regs,
1719                  u32 nr_redist_regions,
1720                  u64 redist_stride,
1721                  struct fwnode_handle *handle)
1722 {
1723     u32 typer;
1724     int err;
1725
~~~
1782     set_handle_irq(gic_handle_irq);
1783
1784     gic_update_rdist_properties();
1785
1786     gic_dist_init();                                        ④
```

ARM64(ARMv8) Linux IRQ Exception Handler(3)

- drivers/irqchip/irq-gic-v3.c
  - gic_handle_irq

```
700 static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
701 {
702     u32 irqnr;
703
704     irqnr = do_read_iar(regs);
705
706     /* Check for special IDs first */
707     if ((irqnr >= 1020 && irqnr <= 1023))
708         return;
709
710     if (gic_supports_nmi() &&
711         unlikely(gic_read_rpr() == GICD_INT_RPR_PRI(GICD_INT_NMI_PRI))) {
712         gic_handle_nmi(irqnr, regs);
713         return;
714     }
715
716     if (gic_prio_masking_enabled()) {
717         gic_pmr_mask_irqs();
718         gic_arch_enable_irqs();
719     }
720
721     if (static_branch_likely(&supports_deactivate_key))
722         gic_write_eoir(irqnr);
723     else
724         isb();
725
726     if (handle_domain_irq(gic_data.domain, irqnr, regs)) {
727         WARN_ONCE(true, "Unexpected interrupt received!\n");
728         gic_deactivate_unhandled(irqnr);
729     }
730 }
```

⑤

소프트인터럽트(softirq)

- SMP 머신에서 BH 개선, 어떤 프로세서에서든 동시에 실행 가능
- 처리시간이 매주 중요하여 빨리 처리해야 하는 경우 유리
- 소프트 인터럽트 실행
  - 하드웨어 인터럽트를 처리한 후
  - 커널 스레드 ksoftirqd에 의해
  - 네트워크 서브시스템과 같이 코드에서 직접 softirq를 검사하여 실행

워크 큐(Work Queue)

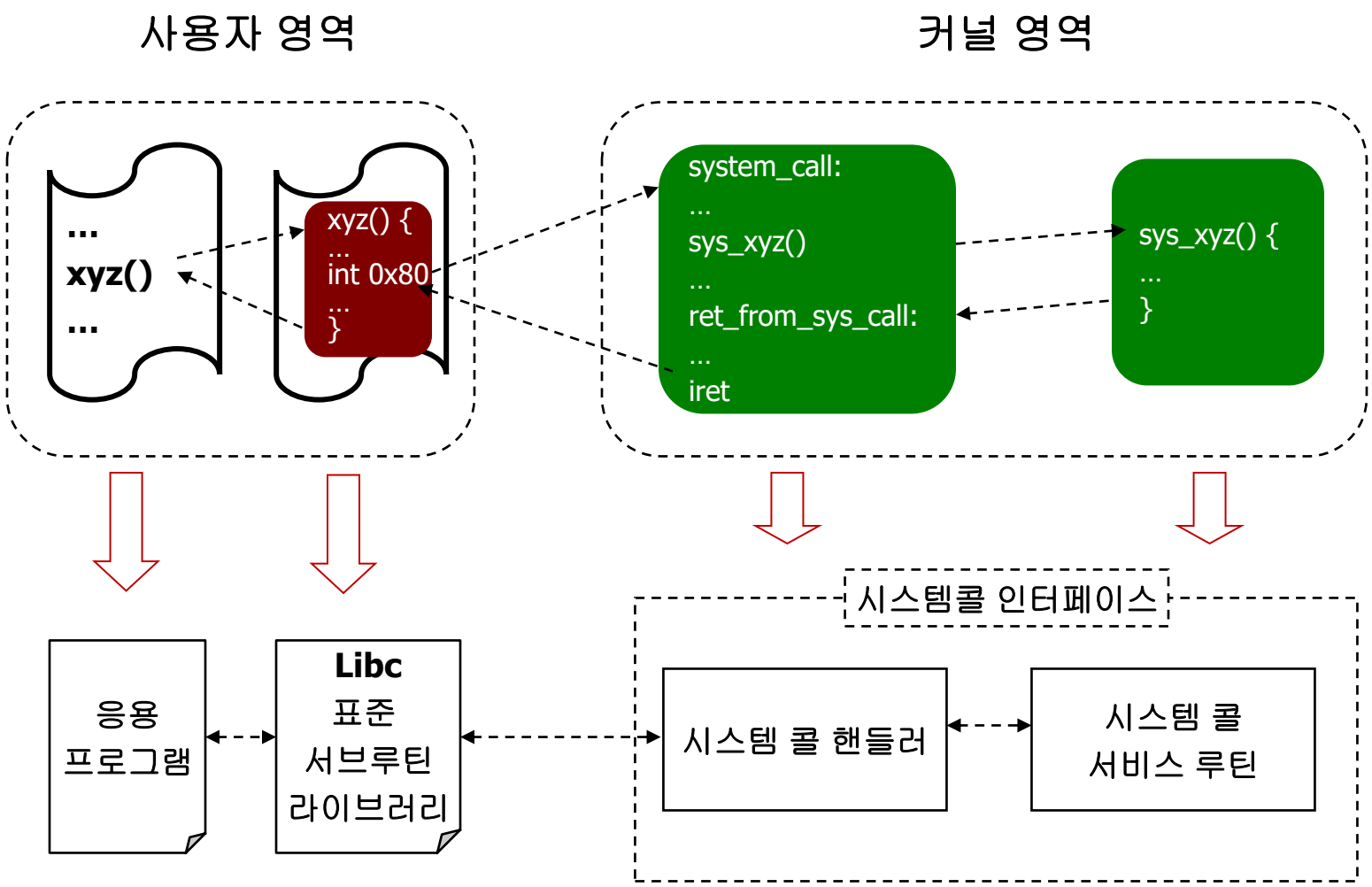
- 워크 큐 사용
  - 리눅스 2.4에 지원되던 태스크 큐를 리눅스 2.6에서 수정 보완
  - 디바이스 드라이버의 인터럽트 서비스 루틴 등에서 함수 등록
  - 워크큐 구조체를 등록하면 keventd와 같은 커널 쓰레드에서 수행
- 등록된 함수 실행될 때 인터럽트 허용 가능
- 일반 사용자 프로세스와 같은 태스크의 특성 포함
  - 2.4커널의 태스크 큐 보완
- 네트워크와 같이 많은 인터럽트가 발생하는 경우는 적합하지 않음

목 차

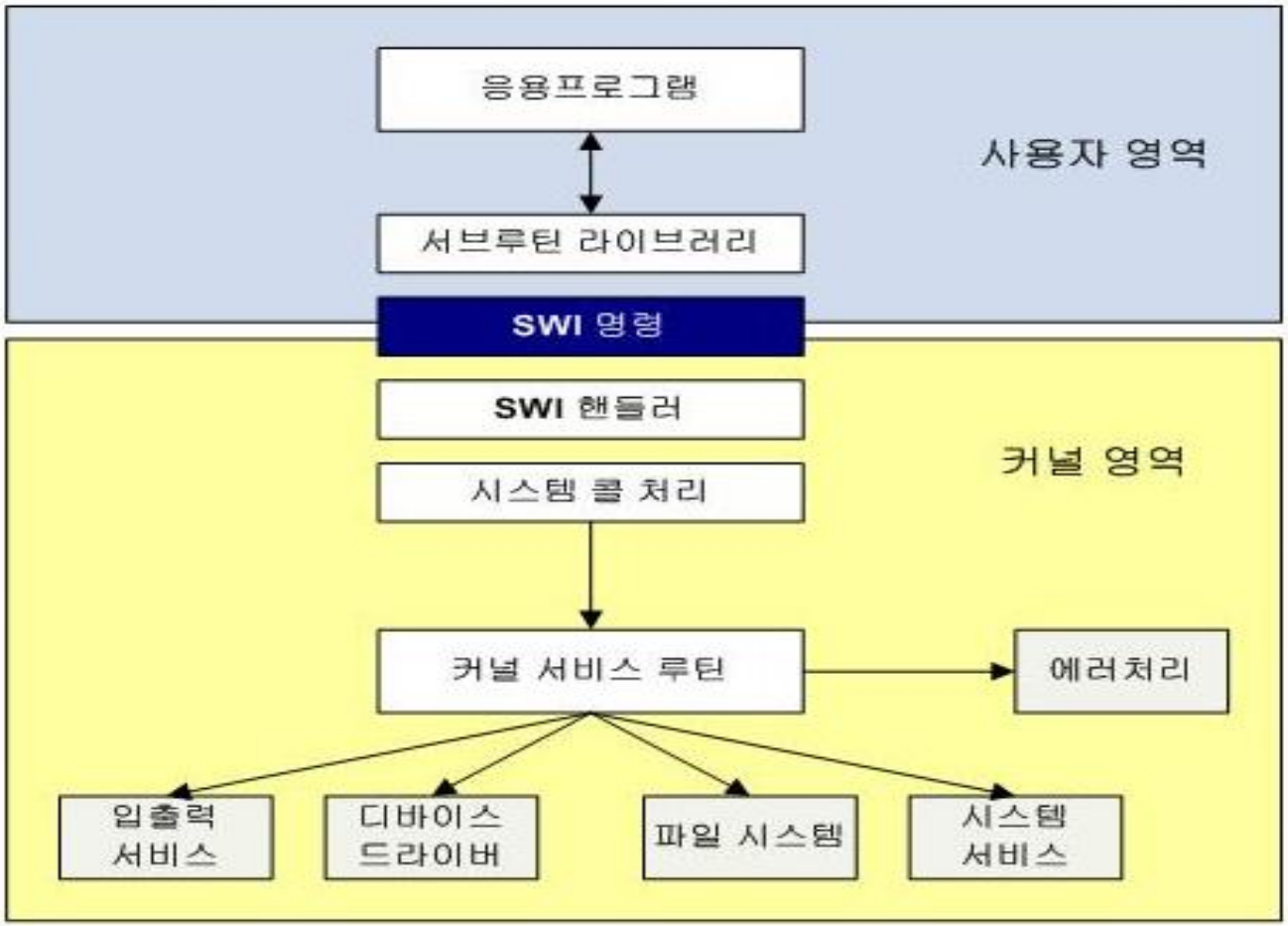
시스템 콜(System Call)

- 시스템 콜(System call)
  - 사용자 프로그램에서 커널 자원을 사용 할 수 있도록 한다.
  - 디바이스 제어, 프로그램 실행, 파일 전송 등
- 포직스 API와 시스템 콜
  - 표준 libc에 정의되어 있는 C 라이브러리의 API 중 일부는 시스템 콜 호출을 위한 루틴 제공
  - 포직스 포준 API를 사용하는 시스템 콜을 사용하는 이유
    - 저수준 프로그래밍이 불필요 ➔ 쉬운 프로그래밍
    - 시스템 보안성 증가
    - 프로그램 호환성 증가

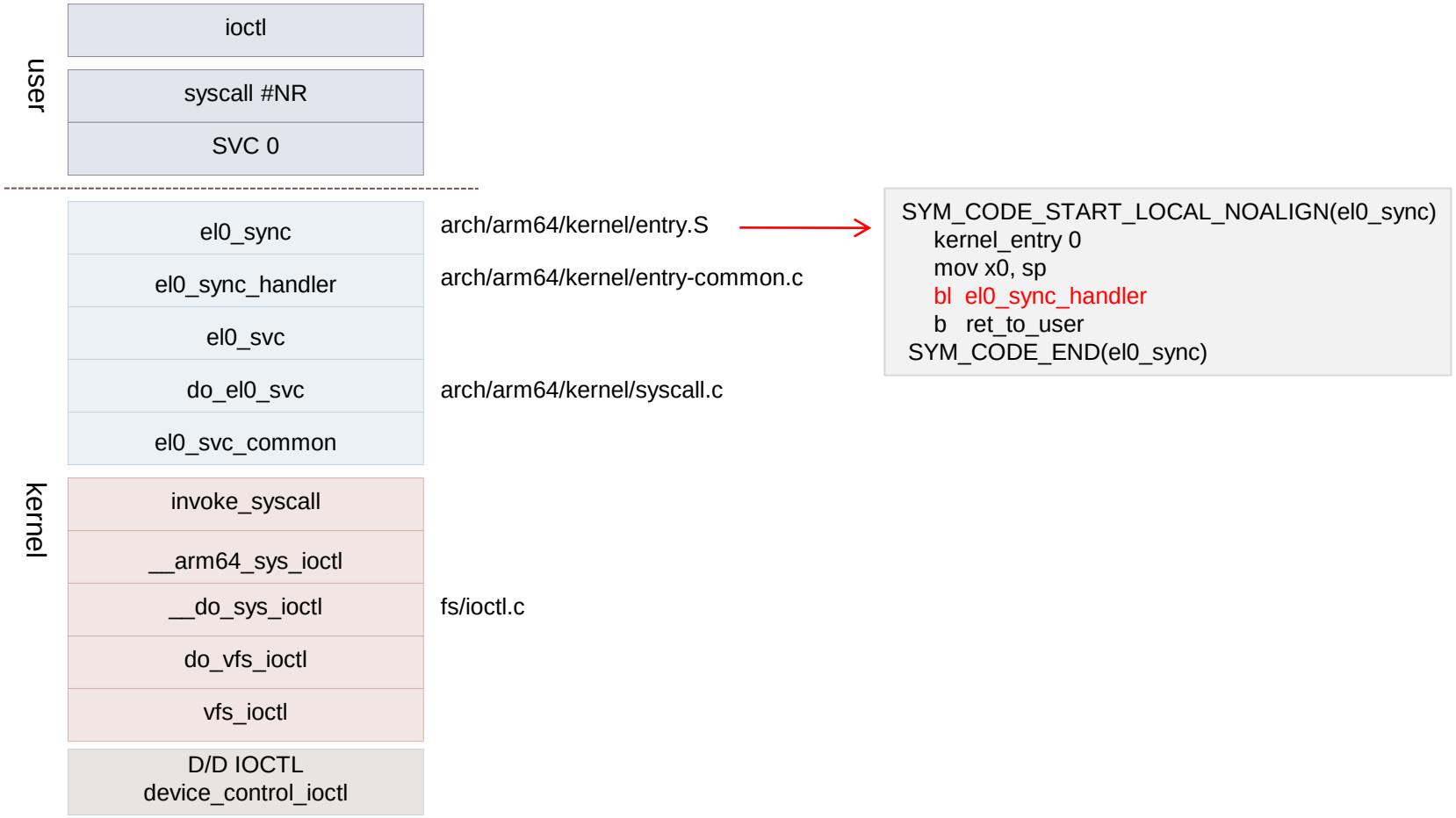
시스템 콜 처리 절차


사용자 영역
커널 영역
...
xyz()
...
xyz() {
...
int 0x80
...
}
system_call:
...
sys_xyz()
...
ret_from_sys_call:
...
iret
sys_xyz() {
...
}
시스템콜 인터페이스
응용
프로그램
Libc
표준
서브루틴
라이브러리
시스템 콜 핸들러
시스템 콜
서비스 루틴

시스템 콜 처리

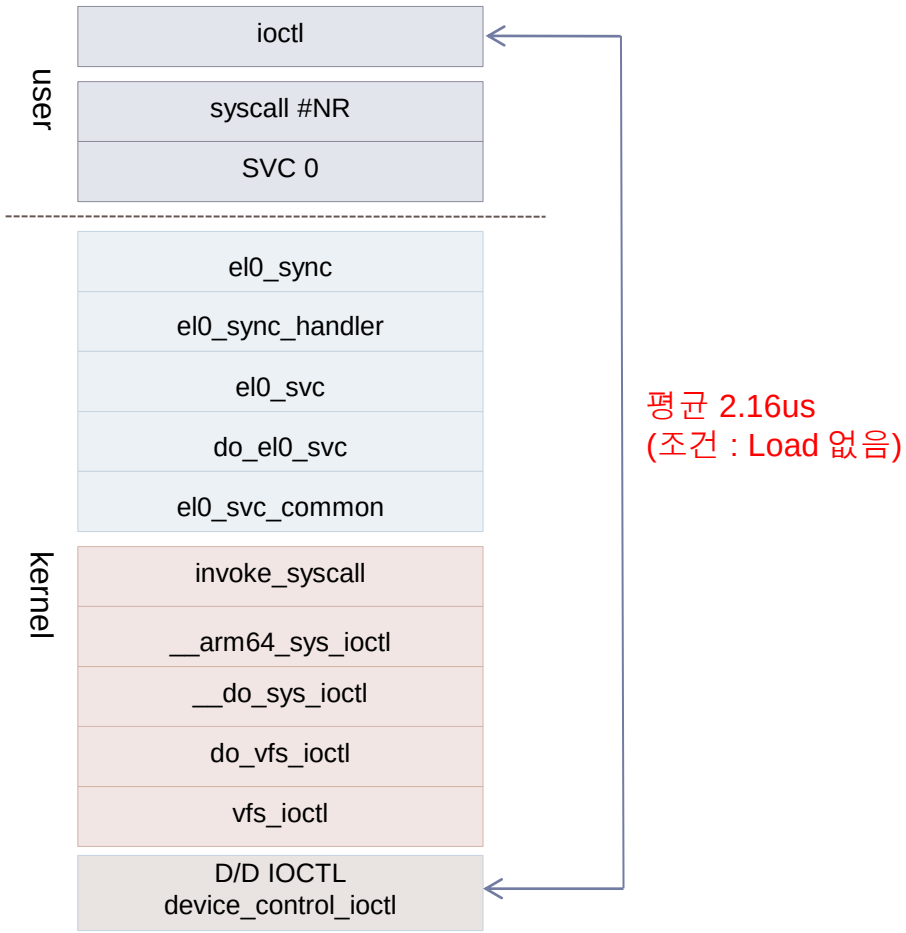

응용프로그램
사용자 영역
서브루틴 라이브러리
SWI 명령
SWI 핸들러
커널 영역
시스템 콜 처리
커널 서비스 루틴
에러처리
입출력 서비스
디바이스 드라이버
파일 시스템
시스템 서비스

ARM64 시스템 콜 처리 단계



ARM64 시스템 콜 처리 시간

User 영역에서 ioctl 호출 후 return 시간 측정 : 평균 2.16 us




```
# nmi-test 1 7
NMI Test : Start.
IOCTL delay time = 2167 ns(2.167 us)
Start = 926782462ns,  End = 926784629 ns
Start time            23:58:34
End time              23:58:34
NMI Test : END. isr_called_count=0
# nmi-test 1 7
NMI Test : Start.
IOCTL delay time = 2250 ns(2.250 us)
Start = 476939588ns,  End = 476941838 ns
Start time            23:58:36
End time              23:58:36
NMI Test : END. isr_called_count=0
# nmi-test 1 7
NMI Test : Start.
IOCTL delay time = 2000 ns(2.000 us)
Start = 580924005ns,  End = 580926005 ns
Start time            23:58:37
End time              23:58:37
NMI Test : END. isr_called_count=0
# nmi-test 1 7
NMI Test : Start.
IOCTL delay time = 2208 ns(2.208 us)
Start = 996835089ns,  End = 996837297 ns
Start time            23:58:38
End time              23:58:38
NMI Test : END. isr_called_count=0
#
```

시스템 콜 정의

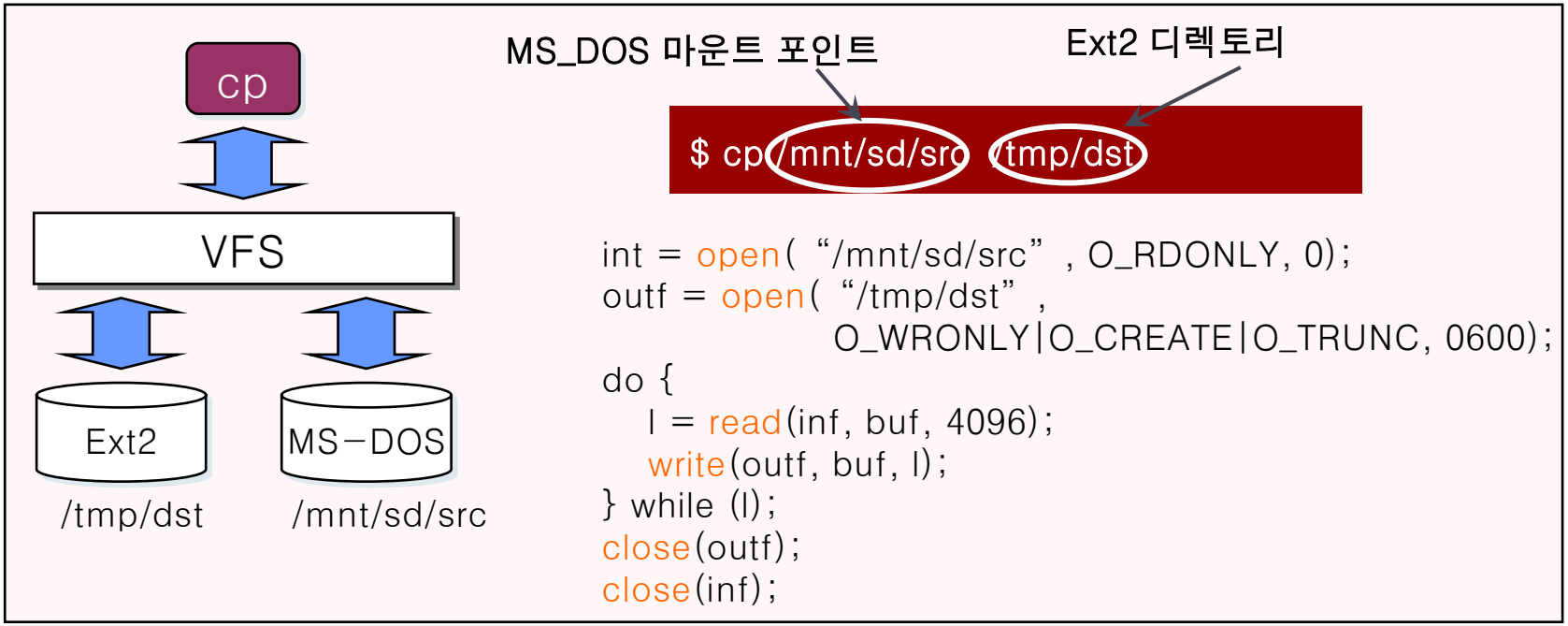
- /linux/include/asm/unistd.h

```
#define __NR_exit                (__NR_SYSCALL_BASE+  1)
#define __NR_fork                (__NR_SYSCALL_BASE+  2)
#define __NR_read                (__NR_SYSCALL_BASE+  3)
#define __NR_write               (__NR_SYSCALL_BASE+  4)
#define __NR_open                (__NR_SYSCALL_BASE+  5)
#define __NR_close               (__NR_SYSCALL_BASE+  6)
--- 이하 생략 ---
```

목 차

가상 파일시스템(Virtual File System)

- 가상 파일시스템(VFS, Virtual File System)
  - 모든 파일시스템에 대한 공통(Common) 인터페이스 제공
  - 리눅스 파일시스템과 관련된 모든 시스템 콜을(System Call)을 처리하는 커널 소프트웨어 계층

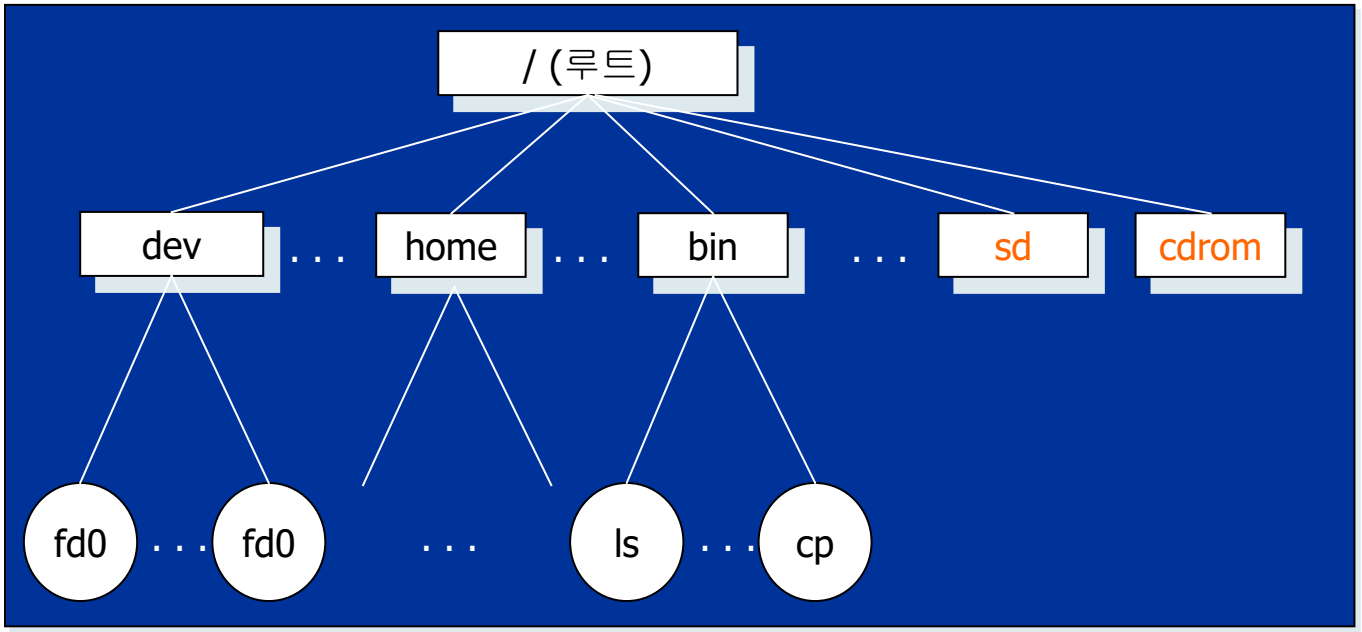


VFS가 지원하는 파일시스템

- 디스크 기반 파일시스템
  - 로컬 디스크 파티션의 기억 장소를 관리
  - 하드디스크, 플로피, CD-ROM 같은 블록 디바이스에 기반
  - ext2, msdos, vfat, ntfs, iso9660(CD-ROM), hpfs(OS/2) 등
- 네트워크 파일시스템
  - 다른 네트워크의 컴퓨터에 속한 파일 시스템에 접근
  - NFS, Coda, AFS, SMB, NCP
- 특수 파일시스템
  - /proc 파일시스템
    - 사용자가 커널의 데이터에 접근할 수 있는 인터페이스 제공
  - /dev 파일시스템
    - 디바이스를 표준 파일시스템으로 관리할 수 있는 인터페이스 제공

VFS와 리눅스 파일시스템

- 루트(ROOT:/) 디렉토리를 기준으로 트리 구성
  - 루트(/) 디렉토라는 대부분 ext3/ext4 유형의 파일 시스템 사용
  - 모든 파일시스템을 루트 파일시스템의 서브 디렉토리에 마운트 하여 사용



VFS와 Common File 모델

- VFS의 핵심
  - 모든 파일시스템을 표현 할 수 있는 Common File(공통 파일) 모델의 도입
- Common File 모델
  - 전형적인 유닉스 파일시스템에서 제공하는 모델 사용
    - mount, read, write, ioctl,..
  - 특정 파일시스템을 구현할 때 자신의 물리적인 구조를 공통 파일 모델로 변환
  - 각 디렉토리는 파일 리스트와 하위 디렉토리를 포함한 일반 파일로 간주
- Common File 모델의 객체 유형
  - 수퍼블록(superblock) 객체
    - 마운트 시킨 파일 시스템 정보를 저장
  - 아이노드(inode) 객체
    - 특정 파일에 대한 일반 정보를 저장
  - 파일(file) 객체
    - 열린 파일과 프로세스 사이의 상호 작용과 관련한 정보 저장
  - 디엔트리(dentry) 객체
    - 디렉토리 항목과 대응하는 파일간 연결에 대한 정보 저장

파일시스템 마운트

- 파일시스템 사용 전에 실행해야 할 동작
  - 등록(Registration)
    - 파일시스템에서 구현한 함수를 커널에서 사용할 수 있도록 하는 동작
    - 시스템 부팅 또는 파일시스템을 구현하는 모듈을 탑재할 때 등록
  - 마운트(Mount)
    - 파일시스템을 사용하기 위하여 리눅스에 연결하는 동작

- 루트 파일시스템과 일반 파일시스템 마운트
  - 루트 파일시스템
    - 시스템 부팅하는 동안 파일시스템 설정 이후에 마운트
    - 루트 파일시스템에 사용되는 장치의 major 번호는 ROOT_DEV 변수에 있다
  - 일반 파일시스템
    - 루트 파일시스템 이후에 루트 파일시스템의 디렉토리 트리에 마운트
    - 파일시스템을 마운트 하기 위해 mount() 시스템 콜을 사용한다.

VFS와 파일 관리

- VFS 기반의 파일 관리
  - 시스템 콜(System Call)을 사용한다.
  - open(), close(), read(), write(), …
- open()
  - Open할 Pathname : Filename
  - 접근 flag : flags
  - 퍼미션 bit mask : mode
  - 리턴 값
    - 성공 : 파일 discriptor
    - 실패 : –1
- read() 와 write()
  - File descriptor : fd
  - 메모리 영역의 Address : buf
  - 한번에 전송할 byte수 : number
  - 리턴 값
    - 성공 : byte의 number
    - 실패 : –1

```
int = open("/mnt/sd/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
        O_WRONLY|O_CREATE|O_TRUNC, 0600);
do {
    l = read(inf, buf, 4096);
    write(outf, buf, l);
} while (l);
close(outf);
close(inf);
```

VFS 시스템 콜

- Open the device

```
int open (char * filename, int flags)
```

- Read from kernel space

```
int read (int fd, void *buf, size_t nbytes)
```

- Write to kernel space

```
int write (int fd, void *buf, size_t nbytes)
```

- I/O control

```
int ioctl (int fd, int cmd, arguments)
```

- Close the device

```
int close (int fd)
```

Define Access Mode

O_RDONLY

O_WRONLY

O_RDWR

O_CREAT

…….

목 차

사용자 프로그램 실행

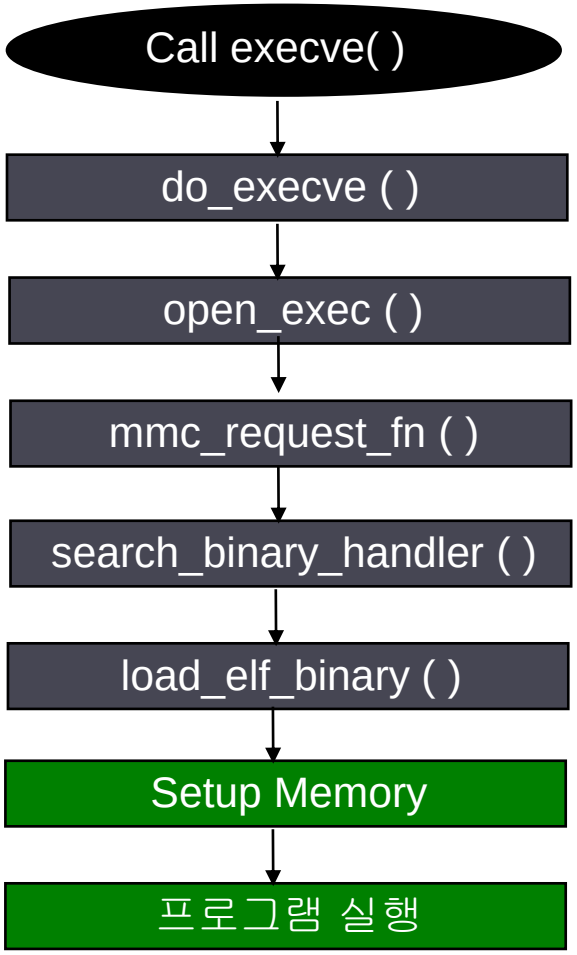
- Linux에서 실행 가능한 파일 Format
  - a.out, ELF 또는 Binary script
- 실행 가능한 Object File
  - Code, data와 함께 프로그램을 메모리에 로드하고 실행하는데 필요한 정보를 가지고 있다.
- Command Interpreter
  - 사용자의 프로그램이나 명령은 command interpreter에 의해서 실행 된다.

fork()와 exec()를 사용한 프로그램 실행

- fork()와 exec()를 사용한 프로세스 생성
  1. 현재 프로세스에서 fork() 호출
  2. fork()로 부터 0 값이 반환 되면 exec()를 호출하여 새로운 프로그램으로 변경
  3. 부모 프로세스는 wait()를 호출하여 자식 프로세스의 종료를 기다리고, 자식 프로세스가 종료 되었다는 정보를 받으면 프로세스 종료

```
switch (pid = fork()){
case -1:
        fatal("fork failed.");
        break;
case 0:  /* 자식 프로세스 */
        execl("/bin/ls","ls","-1",(char *)0));
        ……..
```

프로그램 실행 절차



$ ls -al

“fs/exec.c”

“fs/exec.c”

“drivers/mmc/card/queue.c”

“fs/exec.c”

“fs/binfmt_elf.c”

텍스트, 데이터, **BSS** 및 스택 영역 초기화

다양한 프로그램 실행 함수

함수명	명령 인자	경로 검색
execl()	다양한 개수의 매개변수 사용	경로 검색 없음
execlp()	다양한 개수의 매개변수 사용	PATH 환경 변수에 따른 경로 검색
execle()	다양한 개수의 매개변수 사용	ENV 환경 변수 전달
execv()	배열로 구성된 하나의 매개 변수 사용	경로 검색 없음
execvp()	배열로 구성된 하나의 매개 변수 사용	PATH 환경 변수에 따른 경로 검색
execve()	배열로 구성된 하나의 매개 변수 사용	ENV 환경 변수 전달

목 차

프로세스간 통신(IPC)

- IPC(Inter-Process Communication)
  - 프로세스들은 상호간의 활동을 조정하기 위하여 프로세스간, 커널과 통신을 하기 위한 메커니즘
- IPC의 종류
  - 시그널 (Signal)
  - 파이프 (Pipe)
  - 소켓 (Socket)
  - 메시지 큐 (Message Queue)
  - 세마포어 (Semaphore)
  - 공유메모리 (Shared Memory)

프로세스간 통신

IPC : Inter Process Communication

시그널(Signal)

- 초기 UNIX 시스템에서 간단하게 프로세스간 통신을 하기 위한 메커니즘으로 사용
  - 간단하고 효율적이다
- 시그널의 기능
  - 특정 이벤트를 프로세스에게 알림
  - 시그널을 받은 프로세스는 자신의 코드에 들어있는 시그널 핸들러 함수를 실행
- 시그널의 응용
  - 프로세스나 프로세스 그룹에 짧은 메시지(번호)를 보내기 위해 사용
  - 커널에서 시스템 이벤트를 프로세스에 알리기 위해 사용

시그널과 관련된 시스템 콜

시스템 콜	설명
kill()	프로세스에 시그널을 보낸다.
sigaction()	시그널과 관련된 동작을 변경한다
signal()	sigaction()과 비슷하다.
sigpending()	대기 중인 시그널이 있는지 검사한다.
sigprocmask()	블록할 시그널 목록을 수정한다.
sigsuspend()	시그널을 기다린다.

시그널 응용 예

- 시그널 전송

```
…. 이하 생략 …
kill ( pid, SIGUSR1 ) ;          /* pid에 해당하는 프로세스에 SIGUSR1 시그널 전송 */
…. 이하 생략 …
```

- 시그널 처리

```
void SigHandler (int ) ;
int main (int argc, char **argv)
{
        -- 이하 생략 --
        /* SIGUSR1 시그널 수신되면 SigHandler 함수 실행 */
        signal ( SIGUSR1, (void *)SigHandler ) ;
        -- 이하 생략 --
}
void SigHandler ( int signo )
{
        If ( signo == SIGUSR1 )
                printf("Signal Handler\n");
}
```

시스템 V IPC

- 세마포어(Semaphore)
  - 프로세스간 중요한 데이터 접근에 있어 서로 간의 접근을 통제
  - 프로세스 간의 데이터 교환이 아닌 공유 데이터의 접근에 필요한 동기화 제공
- 메시지 큐(Message Queue)
  - 프로세스 간 메시지 교환
- 공유 메모리(Shared Memory)
  - 다른 프로세스가 사용하는 메모리 영역에 대해 직접 접근하여 읽거나 씀
  - IPC 방법 중 가장 빠르다.

시스템 V IPC를 위한 시스템 콜

	메시지 큐	세마포어	공유 메모리
헤더 파일	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
Create와 open 시스템 콜	msgget	semget	shmget
Control 시스템 콜	msgctl	semctl	shmctl
IPC 동작을 위한 시스템 콜	msgsnd	semop	shmat
	msgrcv		shmdt

질의 응답