11. 운영체제 연동

11.1 프로세서와 운영체제 간 관계

> 하드웨어의 '핵심'과 소프트웨어의 '관리자' 간 협력 구조

🧠 개념 요약

컴포넌트	설명
프로세서(CPU)	명령어 실행, 연산, 제어 신호 생성
운영체제(OS)	자원 관리, 프로세스/메모리/입출력 제어

> CPU는 일을 "수행"하는 장치,
> OS는 "어떤 일이 언제, 어떻게 수행될지"를 결정하는 **중재자**

🔁 두 시스템의 상호 작용 구조

```
[ 사용자 프로그램 ]
        ↓ 시스템 콜
[ 운영체제 (커널 모드) ]
        ↓ 인터럽트 제어 / 자원 스케줄링
[ 하드웨어 (CPU, 메모리, I/O) ]
```

주요 키워드:

- **시스템 콜 (System Call)**: 사용자 프로그램이 OS 기능을 요청하는 인터페이스
- **인터럽트 (Interrupt)**: 하드웨어 이벤트가 CPU에게 신호를 보내는 방식
- **컨텍스트 스위칭 (Context Switch)**: 프로세서 상태를 다른 프로세스로 교체

📦 CPU가 OS에 제공하는 기능

기능	설명
특권 명령 지원	커널 모드에서만 실행 가능한 명령어 제공 (ex. I/O 제어, MMU 설정 등)
인터럽트 메커니즘	하드웨어 이벤트를 OS가 감지하고 처리할 수 있도록 지원
시스템 콜 인터페이스	사용자 코드에서 커널로 안전하게 진입할 수 있는 수단
모드 전환(User ↔ Kernel)	CPU가 모드 비트를 바꾸어 OS 보호
메모리 보호(MMU)	주소 공간을 분리하여 다중 프로세스 충돌 방지
타이머 인터럽트	시분할 운영체제 구현에 필요한 정기 인터럽트 제공

🛠️ 하드웨어 추상화 계층 (HAL: Hardware Abstraction Layer)

운영체제는 CPU의 세부적인 레지스터 구조나 명령어 집합을 **직접 사용하는 대신**,
일관된 인터페이스(HAL)를 통해 다양한 CPU를 지원하게 된다.

예시	설명
`x86`, `ARM`, `RISC-V`	서로 다른 구조지만 공통된 OS 인터페이스 제공
HAL 기능	타이머 설정, 인터럽트 등록, 캐시 제어 등

🔁 시스템 콜 처리 흐름

```
[1] 사용자 프로그램에서 system call 발생
[2] CPU → 특수 명령어 (예: `int 0x80`, `ecall`)
[3] 하드웨어가 모드를 User → Kernel로 전환
[4] OS 커널의 시스템 콜 핸들러로 진입
[5] OS가 요청을 처리하고 결과 반환
[6] 다시 User 모드로 복귀
```

> 이는 하드웨어의 **트랩(Trap)** 또는 **소프트웨어 인터럽트**로 구현됨.

🧩 인터럽트와 OS의 역할

인터럽트 종류	예시	OS에서 하는 일
하드웨어 인터럽트	타이머, 키보드, 네트워크	장치 드라이버 호출, 인터럽트 서비스 루틴(ISR) 실행
소프트웨어 인터럽트	시스템 콜	사용자 → 커널 진입

→ CPU는 인터럽트 벡터 테이블을 통해 각 인터럽트에 맞는 **OS 루틴으로 점프**함

🔐 보호 메커니즘: 모드 전환 및 MMU

✅ 모드 전환 (User ↔ Kernel Mode)

- User 모드에서는 **I/O, MMU 설정 등의 위험한 명령 금지**
- 시스템 콜이나 인터럽트로만 커널 진입 허용
- 커널 모드에서만 전체 자원 접근 가능

✅ MMU (Memory Management Unit)

- 운영체제가 페이지 테이블을 설정하고,
  CPU가 이를 기반으로 가상 주소 → 물리 주소 변환

> 결과적으로 CPU는 **OS가 정의한 메모리 경계 내에서만 실행** 가능

📈 실제 구조 예시: 리눅스 + ARM Cortex-A

구성 요소	역할
SVC #0	시스템 콜 트랩 명령
Exception Vector Table	ARM의 인터럽트 벡터
CPSR 모드 비트	현재 모드 (User / Supervisor / IRQ 등) 표시
OS 커널의 핸들러	`_sys_call_handler`, `_irq_handler` 등
MMU 설정	페이지 테이블 주소 지정, 캐시 정책 설정

📌 요약 정리

항목	설명
CPU 역할	명령어 실행, 인터럽트 감지, 모드 전환 수행
OS 역할	자원 관리, 스케줄링, 시스템 콜 처리
상호작용	시스템 콜, 인터럽트, MMU 제어
보호 기법	특권 모드, MMU, 인터럽트 마스킹
하드웨어 추상화	HAL을 통해 다양한 CPU를 OS가 동일하게 다룸

11.2 시스템 콜 및 트랩 처리

> 사용자 공간에서 커널 공간으로의 '안전한 문'

🧠 시스템 콜(System Call)이란?

시스템 콜은 사용자 프로그램이 운영체제의 기능(파일 열기, 메모리 할당 등)을 요청하는 **유일한 인터페이스**이다.

> 사용자 애플리케이션은 직접 하드웨어 제어나 커널 자원에 접근할 수 없기 때문에,
> 반드시 시스템 콜을 통해 **CPU 모드를 전환**하고 **OS 커널 코드로 진입**해야 한다.

🛠️ 트랩(Trap)이란?

트랩(Trap)은 CPU가 감지하는 **소프트웨어적 예외 또는 이벤트**로,
운영체제에게 제어권을 넘겨주는 메커니즘이다.

> 트랩은 일반적으로 **시스템 콜 실행, 예외 처리(0으로 나누기 등)** 시 발생
> → 하드웨어적으로는 **인터럽트와 유사**하지만 소프트웨어 기원이 많음

🔄 시스템 콜 처리 전체 흐름

```
[사용자 프로그램]
    ↓ 시스템 콜 호출 (예: write())
[라이브러리] → `int 0x80`, `ecall`, `svc`
    ↓ Trap 발생 → 모드 전환 (User → Kernel)
[OS 커널 진입] → 트랩 벡터 → 핸들러 호출
    ↓ 시스템 콜 처리 (파일 열기, 읽기 등)
    ↓ 결과 설정
[리턴 → 모드 복귀 (Kernel → User)]
    ↓ 사용자 프로그램으로 복귀
```

🧩 주요 구성 요소

컴포넌트	역할
Trap 명령어	사용자 → 커널 진입 (`int`, `ecall`, `svc`)
Trap 벡터(예외 벡터)	각 트랩 번호에 따라 호출될 함수 주소
커널 모드 스택	커널 진입 시 사용할 안전한 스택
시스템 콜 번호	어떤 시스템 콜을 요청했는지 식별
레지스터/메모리	인자 전달 및 결과 반환

📦 플랫폼별 시스템 콜 트랩 명령

아키텍처	트랩 명령	번호 레지스터	인자 전달 방식
x86 (32bit)	`int 0x80`	`eax`	`ebx`, `ecx`, `edx`, …
x86 (64bit)	`syscall`	`rax`	`rdi`, `rsi`, `rdx`, …
ARM	`svc #0`	`r7`	`r0`, `r1`, …
RISC-V	`ecall`	`a7`	`a0`, `a1`, …

🔍 시스템 콜 번호 및 인자 처리 예 (x86 예시)

```
; write(fd, buf, len)
mov eax, 4      ; 시스템 콜 번호 (4 = write)
mov ebx, 1      ; fd = stdout
mov ecx, msg    ; 버퍼 주소
mov edx, 13     ; 길이
int 0x80        ; 트랩 호출
```

→ 커널은 `eax`의 값을 보고 해당 syscall table에서 `sys_write()` 함수를 호출함

🧠 트랩과 인터럽트의 차이

구분	트랩 (Trap)	인터럽트 (Interrupt)
발생 원인	소프트웨어 명령 / 예외	하드웨어
예시	`int`, `ecall`, divide-by-zero	타이머, 키보드, 디스크
발생 시점	명령어 실행 중	명령어 사이
제어권 이동	즉시 커널로	즉시 커널로
목적	OS 기능 호출, 예외 처리	장치 응답, 시분할

🔁 커널 진입 시 처리 흐름 (상세)

1. 현재 **사용자 모드 상태 저장**
    - PC, 스택 포인터, 플래그 레지스터, 레지스터 등
2. **모드 전환** (User → Kernel)
3. 커널 스택으로 스위칭
4. 시스템 콜 번호를 기준으로 핸들러 테이블 참조
5. 시스템 콜 실행
6. 결과값을 사용자 영역 레지스터에 저장
7. **모드 복귀** 후 사용자 프로그램으로 복귀

🧬 커널 내 시스템 콜 핸들러 구조 (리눅스 예)

```
asmlinkage long sys_write(unsigned int fd, const char __user *buf, size_t count) {
    // 유효성 검사
    // 커널 버퍼 복사
    // 실제 파일 디바이스에 쓰기
    // 결과 반환
}
```

→ syscall 번호는 `sys_call_table[]` 배열로 연결되어 있음

📌 요약 정리

항목	설명
시스템 콜	사용자 프로그램이 OS 기능을 요청하는 안전한 인터페이스
트랩(Trap)	CPU가 커널에게 제어를 넘기기 위한 이벤트
모드 전환	트랩 발생 시 CPU가 User → Kernel 모드로 변경
처리 흐름	트랩 명령 → 핸들러 → 작업 처리 → 복귀
아키텍처 구현	`int`, `ecall`, `svc` 명령으로 트랩 구현

항목	설명
OS 내 처리	시스템 콜 번호 기반으로 함수 디스패치 후 처리 결과 반환

11.3 컨텍스트 스위칭

> 프로세서의 현재 사용자를 다른 작업으로 '전환'하는 메커니즘

🧠 컨텍스트란?

컨텍스트(Context)는 **프로세스 또는 스레드의 실행 상태를 구성하는 정보 집합**이다.
CPU는 오직 하나의 명령어 흐름만을 실행할 수 있기 때문에,
다른 작업으로 전환할 때는 **이전 작업의 컨텍스트를 저장하고 새로운 컨텍스트를 복원**해야 한다.

📦 컨텍스트의 구성 요소

구성 요소	설명
일반 레지스터	R0~~Rn, x86의 EAX~~EDI 등
스택 포인터 (SP)	현재 함수 호출의 스택 위치
프로그램 카운터 (PC)	다음에 실행될 명령어의 주소
프로세서 상태 레지스터 (PSR/FLAGS)	모드, 인터럽트 상태 등
메모리 매핑 정보	페이지 테이블 주소, MMU 설정
코어별 특수 레지스터	x86의 cr3 (페이지 디렉토리), ARM의 SPSR 등

> 이 정보가 저장되어야 프로세스가 **중단된 지점부터 정확히 재개**될 수 있다.

🔄 컨텍스트 스위칭의 발생 시점

시점	설명
타이머 인터럽트	시분할 스케줄링에서 주기적으로 발생
시스템 콜	`sleep()`, `yield()` 등에서 스스로 CPU 반납
I/O 대기	블로킹 호출 → 다른 프로세스로 전환 필요
우선순위 선점	높은 우선순위의 프로세스가 준비되었을 때

🔁 컨텍스트 스위칭 전체 흐름

```
[1] 타이머 인터럽트 or 이벤트 발생
 ↓
[2] 커널 진입 (trap)
 ↓
[3] 현재 프로세스 상태 저장 (레지스터, PC, SP 등)
 ↓
[4] 스케줄러가 다음 프로세스 결정
 ↓
[5] 선택된 프로세스의 상태 복원
 ↓
[6] 사용자 모드 복귀
```

🧪 실제 컨텍스트 저장 방식 (예: x86, ARM)

✅ x86 시스템 예시

- 저장 대상: `EAX`, `EBX`, ..., `ESP`, `EIP`, `EFLAGS`
- 전용 구조체 `struct pt_regs` 사용
- `switch_to(prev, next)` 함수로 커널 스택을 이동
- `iret` 명령어로 유저 모드 복귀

✅ ARM Cortex-A 예시

- `R0~R12`, `LR`, `PC`, `SPSR` 등 저장
- 컨텍스트는 커널 스택에 푸시됨
- 모드 전환을 위해 `SVC` → `Handler`로 진입
- 복귀 시 `MOVS PC, LR`로 이전 상태 복원

🧩 Linux 커널 예시 (ARM 기준)

```
asmlinkage void __switch_to(struct task_struct *prev, struct task_struct *next)
{
    // prev의 커널 스택에서 컨텍스트 저장
    // next의 커널 스택으로 이동
    // 레지스터 복원
}
```

→ `task_struct` 내부에는 `thread_struct`가 포함되어 있으며,
→ 여기에 SP, PC, FP 등의 **하드웨어 컨텍스트**가 들어있음.

⏱️ 컨텍스트 스위칭의 비용

자원	영향
시간	수백 ~ 수천 사이클 소모 (특히 캐시 무효화 발생 시)

자원	영향
캐시 영향	새 프로세스는 기존 캐시를 거의 사용 못함
TLB 영향	주소 공간 전환 시 TLB 플러시 발생 가능
파이프라인 플러시	실행 중 명령어 취소 후 새 명령어 로딩 필요

> 그래서 "스레드 간 컨텍스트 스위칭 비용이 프로세스보다 낮다"는 것도 맞음 (주소 공간이 같으면 TLB 플러시 없음)

💡 최적화 전략

기법	설명
Lazy Switching	FPU, SIMD 레지스터는 필요할 때만 저장
TLB Shootdown 방지	SMP 환경에서 불필요한 TLB flush 줄이기
Core Affinity 유지	프로세스가 항상 같은 코어에서 실행되도록
커널 스택 공유	유사한 커널 태스크들 간 공유 스택 재활용

📌 요약 정리

항목	설명
컨텍스트	프로세스 또는 스레드의 실행 상태 전체
스위칭 이유	인터럽트, I/O, 타임슬라이스 만료 등
저장 항목	레지스터, PC, SP, PSR, MMU 상태 등
비용	캐시 및 TLB 영향, 지연 발생
OS 내 처리	저장 → 스케줄 → 복원 흐름
최적화 기법	Lazy save, 코어 고정, TLB 공유 등

11.4 프로세스 및 스레드 관리 구조

> 운영체제의 핵심 자원인 "실행 단위"의 내부 구조를 해부하다

🧠 프로세스(Process)란?

프로세스는 **실행 중인 프로그램의 인스턴스**로,
독립적인 주소 공간, 코드, 데이터, 스택, 시스템 자원을 보유한다.

구성 요소	설명
코드 영역 (Text)	실행 명령어 저장
데이터 영역	전역변수, static 변수 등

구성 요소	설명
힙 (Heap)	동적 메모리 할당 영역
스택 (Stack)	함수 호출 및 로컬 변수 저장
커널 구조체	`task_struct` 등에서 관리

📝 스레드(Thread)란?

스레드는 **프로세스 내부의 실행 흐름 단위**로,
코드/데이터/힙 영역은 **공유**하고,
레지스터/스택/PC만 별도로 관리함.

> 하나의 프로세스 안에 여러 개의 스레드가 **동시에 실행** 가능하고
> 모든 스레드는 **같은 주소 공간을 공유**함.

🧩 프로세스 vs 스레드 비교

항목	프로세스	스레드
주소 공간	독립	공유
자원 소유	고유 (파일, 메모리 등)	공유
생성 비용	높음	낮음
컨텍스트 스위치	무거움 (MMU, TLB 포함)	가벼움 (레지스터, 스택만)
충돌 위험	적음	높음 (동기화 필요)

📦 운영체제 내부 구조체 (Linux 기준)

✅ 1. `task_struct`: 프로세스/스레드 관리의 중심

```
struct task_struct {
    pid_t pid;
    long state;
    struct mm_struct *mm;          // 메모리 공간
    struct thread_struct thread;   // 하드웨어 컨텍스트
    struct files_struct *files;    // 열린 파일 테이블
    struct task_struct *parent;    // 부모 프로세스
    struct list_head children;     // 자식 리스트
    ...
};
```

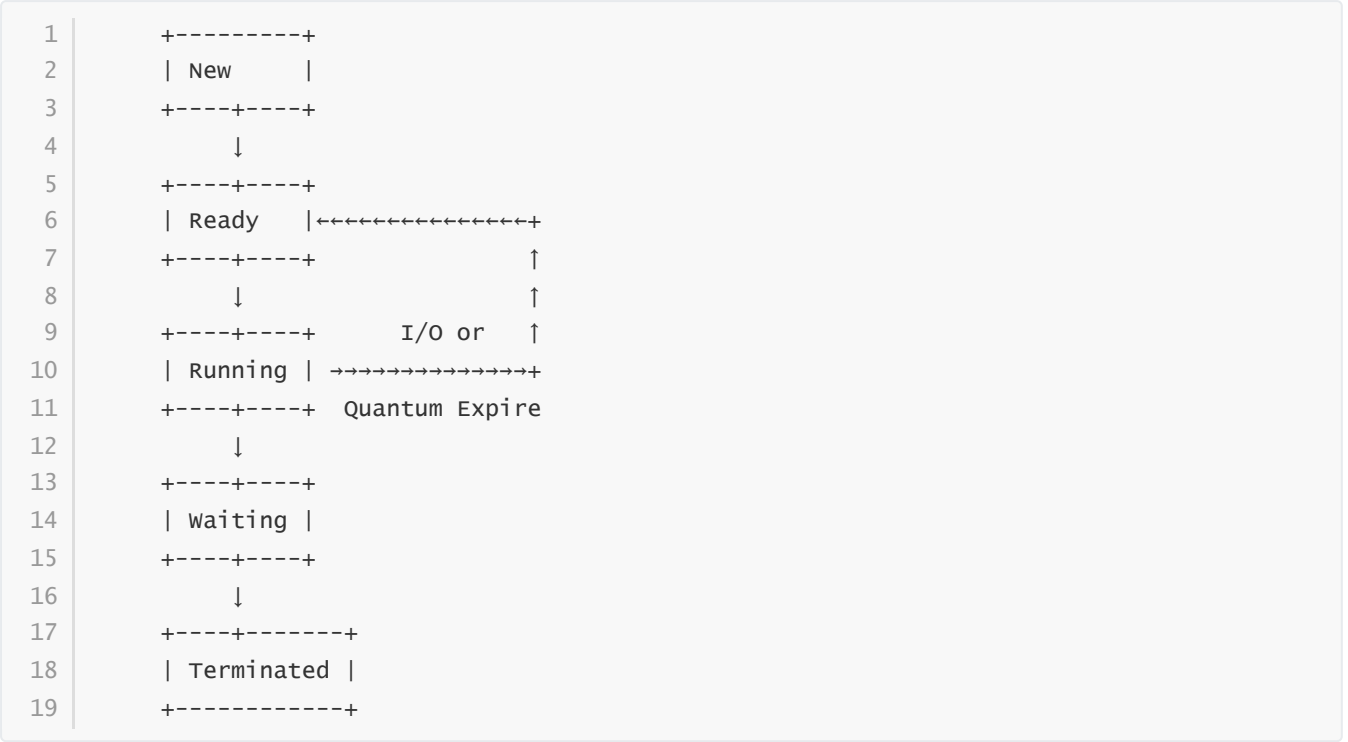
> 프로세스와 스레드는 `task_struct`를 **공유**하되,
> 스레드는 **주소 공간(mm), 파일 테이블(files) 등을 공유**함

✅ 2. `thread_struct`: CPU 컨텍스트 저장용

```
struct thread_struct {
    unsigned long sp;     // 스택 포인터
    unsigned long ip;     // 명령어 포인터
    unsigned long flags;  // 상태 플래그
    ...
};
```

> 컨텍스트 스위칭 시 레지스터 등의 하드웨어 정보를 저장하는 구조

🔁 프로세스 상태 전이 (5-State Model)



상태	설명
New	생성 중
Ready	실행 대기 중 (스케줄러 큐 대기)
Running	CPU에서 실행 중
Waiting	I/O 또는 이벤트 대기
Terminated	종료됨 (exit)

⚙️ 프로세스/스레드 생성 방식

✅ fork() + exec()

- `fork()` → 부모를 복제한 새로운 프로세스 생성 (주소 공간 복제)
- `exec()` → 새로운 프로그램으로 메모리 덮어쓰기

✅ clone() / pthread_create()

- `clone()`은 특정 자원(`CLONE_VM`, `CLONE_FILES`, ...)을 공유할 수 있음
- `pthread_create()`는 일반적인 사용자 스레드 생성 방식

🔁 스케줄러와의 연계

항목	설명
프로세스 큐	Ready 상태의 `task_struct`들이 연결 리스트로 저장됨
우선순위 기반	NICE, CFS(Completely Fair Scheduler) 등
타이머 인터럽트	주기적으로 스케줄러 호출 → 컨텍스트 스위칭 발생
자원 블로킹	Waiting 상태로 전환 시 다른 프로세스 선택

📈 사용자 vs 커널 스레드

유형	설명
사용자 스레드 (User Thread)	POSIX `pthread`, 경량, 빠름, 커널 unaware
커널 스레드 (Kernel Thread)	커널이 인식하고 스케줄링하는 스레드 (`ksoftirqd`, `kworker` 등)
Hybrid Model	M:N 모델 (예: old Solaris)

📌 요약 정리

항목	설명
프로세스	자원(메모리, 파일 등)을 보유한 독립 실행 단위
스레드	프로세스 내부의 실행 흐름, 주소 공간 공유
task_struct	모든 실행 단위를 나타내는 커널 구조체
thread_struct	하드웨어 컨텍스트 저장 공간
상태 전이	Ready ↔ Running ↔ Waiting 등으로 이동
스케줄링 대상	대부분 task_struct (프로세스 + 커널 스레드 포함)

11.5 인터럽트와 커널 진입

> 하드웨어 신호가 운영체제로 이어지는 실시간 경로

🧠 인터럽트란?

인터럽트(Interrupt)는
CPU가 명령어 실행 중 **외부 또는 내부 이벤트에 의해 흐름을 중단하고**
특정 루틴(Interrupt Handler)을 실행한 뒤
다시 원래 작업으로 복귀하도록 만드는 하드웨어 메커니즘이다.

📦 인터럽트의 분류

분류	종류	예시
하드웨어 인터럽트	외부 장치	키보드 입력, 디스크 완료, 타이머
소프트웨어 인터럽트	명령어로 발생	`int 0x80`, `ecall`, `svc` (→ 시스템 콜)
예외(Exception)	CPU 내부 오류	Divide-by-zero, Page Fault, Invalid Opcode

🔄 인터럽트 처리 흐름 (일반 구조)

```
[1] 하드웨어 장치 → 인터럽트 요청 신호(IRQ)
 ↓
[2] CPU → 현재 실행 중인 명령어 완료 후 인터럽트 감지
 ↓
[3] CPU → 현재 컨텍스트 저장 (PC, PSR, 레지스터 등)
 ↓
[4] CPU → 인터럽트 벡터 테이블 참조 (ISR 주소)
 ↓
[5] ISR 실행 → 해당 이벤트 처리
 ↓
[6] 컨텍스트 복원 후 원래 위치로 복귀 (iret 등)
```

📍 인터럽트 벡터 테이블 (IVT)

- **IVT**는 인터럽트 번호에 따라 ISR의 **시작 주소를 저장한 테이블**
- CPU가 인터럽트를 받으면 해당 번호에 해당하는 **엔트리로 점프**

예시 (x86)	주소	인터럽트
`0x00`	Divide by Zero	
`0x20`	Timer	
`0x21`	Keyboard	
`0x80`	System Call	

> ARM과 RISC-V는 각각 **vector base register(VBAR, MTVEC)**로 벡터 테이블 위치 지정

🧬 커널 진입 시 동작

커널 진입은 다음 조건 중 하나일 때 발생:

- 하드웨어 인터럽트
- 시스템 콜 (소프트웨어 인터럽트)
- 예외 (페이지 폴트, 보호 위반 등)

이때 CPU는:

1. 현재 **모드를 User → Kernel**로 전환
2. **인터럽트 벡터를 따라 핸들러로 점프**
3. **커널 스택 사용** 시작
4. ISR이 끝나면 **원래 컨텍스트로 복귀**

🔁 인터럽트 컨텍스트 vs 프로세스 컨텍스트

항목	인터럽트 컨텍스트	프로세스 컨텍스트
정의	인터럽트 핸들러 실행 중 상태	일반 사용자/커널 모드 실행 중 상태
스케줄링	불가능	가능
차단 가능성	비동기, preempt 불가	가능
커널 API 사용 제한	제한됨 (`sleep()`, `malloc()` 불가)	자유로움

> 인터럽트 핸들러는 빠르게 끝나야 하며, **슬립 불가, 스케줄 불가, 복잡 연산 제한**

🛠️ 실제 구조 예: x86 아키텍처

1. 인터럽트 발생
2. CPU는 **IDT(Interrupt Descriptor Table)**를 사용해 ISR 주소 결정
3. 현재 `EFLAGS`, `CS`, `EIP` 저장 → 커널 스택으로 이동
4. ISR 실행
5. 복귀 시 `iret`로 원래 위치 복원

🛠️ 실제 구조 예: ARM 아키텍처

1. 인터럽트 발생 시 자동으로 모드 전환 (IRQ 모드 등)
2. `SPSR`, `LR_irq`, `SP_irq` 자동 설정
3. VBAR (Vector Base Address Register) 기준으로 ISR 진입
4. 복귀 시 `SUBS PC, LR, #4` 또는 `MOVS PC, LR` 사용

💡 인터럽트의 우선순위 및 마스킹

기능	설명
우선순위	여러 인터럽트가 동시에 발생 시 처리 순서 결정
마스킹 (Masking)	특정 인터럽트를 일시적으로 차단 (`cli`, `cpsid i`)
Nesting 지원	높은 우선순위 인터럽트가 낮은 인터럽트 핸들러 중단 가능

📌 요약 정리

항목	설명
인터럽트	외부 신호로 CPU 흐름을 강제로 변경하는 장치
벡터 테이블	인터럽트 번호 → 핸들러 주소 연결
커널 진입	사용자 모드에서 인터럽트 발생 시 보호 모드 전환
컨텍스트 저장	PC, PSR, 레지스터 등 커널 스택에 저장
인터럽트 컨텍스트	제한된 실행 환경, 빠르게 처리해야 함
복귀 명령어	x86: `iret`, ARM: `subs pc, lr, #4`, RISC-V: `mret` 등

11.6 RTOS와 일반 OS의 차이

> 실시간성과 범용성 사이의 핵심 구조적 차이

🧠 기본 정의

구분	RTOS (실시간 운영체제)	일반 OS (범용 운영체제)
정의	**정해진 시간 내에 작업을 확실히 완료**할 수 있도록 보장하는 운영체제	다양한 목적의 작업을 **최대한 효율적으로 처리**하는 운영체제
목적	**정확한 응답 시간 보장 (Deadlines)**	**처리량, 사용자 편의성, 자원 효율 극대화**
예시	FreeRTOS, VxWorks, ThreadX, RTEMS	Linux, Windows, macOS, Android

⏱️ 실시간성 (Determinism)

항목	RTOS	일반 OS
응답 시간	예측 가능 (정해진 시간 안에 항상 동작)	불확실, 지연 발생 가능
지터(Jitter)	거의 없음 (일관된 응답)	OS 상태, 부하에 따라 변동

항목	RTOS	일반 OS
인터럽트 응답	수 μs 이내, 정밀 제어 가능	수 ms 이상 소요될 수 있음
데드라인 대응	하드/소프트 실시간 보장	데드라인 보장 불가능

🔄 스케줄링 방식

항목	RTOS	일반 OS
스케줄링 정책	**우선순위 기반, 선점형** 중심	우선순위 + **공정성(Fairness), 시분할** 중심
스레드 선점	고우선순위 태스크 즉시 실행	공정성을 고려해 제한됨
고정 우선순위	가능 (Priority Inversion 방지 필요)	동적 우선순위 또는 CFS(Completely Fair Scheduler) 사용

⚙️ 커널 구조

항목	RTOS	일반 OS
커널 크기	작고 모듈화됨 (수 KB~수백 KB)	크고 복잡 (수 MB 이상)
구조	모놀리식 또는 마이크로커널 기반	대부분 모놀리식 커널
전환 속도	빠름 (μs 단위)	상대적으로 느림 (ms 단위)
하드웨어 제어	직접 제어 중심	드라이버 계층화 후 처리

🧠 메모리 및 자원 관리

항목	RTOS	일반 OS
메모리 관리	일반적으로 **정적 할당**	동적 메모리 관리 지원
MMU	대부분 없음 / 제한적	전체 가상 메모리 지원
멀티태스킹	기본 지원, 단순한 구조	복잡한 컨텍스트 스위칭 포함
파일 시스템	경량 또는 없음	완전한 POSIX 파일 시스템 제공

🔁 우선순위 역전 (Priority Inversion) 대응

RTOS는 높은 우선순위 작업이 **낮은 우선순위 작업 때문에 지연되지 않도록**
`Priority Inheritance`, `Priority Ceiling Protocol` 등의 메커니즘을 반드시 포함해야 함.

📦 응용 분야 차이

RTOS	일반 OS
임베디드 시스템	데스크탑 / 서버
항공우주, 의료기기, 자동차	게임, 웹 브라우징, 멀티미디어
로봇 제어, PLC, IoT	데이터 처리, 가상화, 사용자 앱 실행
하드 실시간 필수	실시간성은 옵션

🛠️ 대표 RTOS vs 일반 OS 비교

항목	FreeRTOS (RTOS)	Linux (GPOS)
커널 크기	수십 KB	수 MB
스케줄링	우선순위 기반, 선점형	공정 스케줄링(CFS), time-slice
응답 시간	수 μs	수 ms
API	제한적 (POSIX 일부)	POSIX 전체, 다양한 시스템 콜
MMU	필요 없음	필수
파일 시스템	없음 / 간단한 FAT	ext4, Btrfs 등 완전한 FS
인터럽트 처리	직접 핸들러 등록	`request_irq()` → IRQ handler 분기

🧩 실시간성이 중요한 분야에서 RTOS가 필요한 이유

분야	실시간 요구 이유
항공 제어 시스템	입력 후 몇 μs 내 반응 필요
자동차 ECU	ABS, 에어백 등 마이크로초 단위 판단
로봇 제어기	센서 → 계산 → 모터 제어까지 딜레이 최소화
의료기기	실시간 생체신호 처리 필수
산업용 장비	타이밍 보장 없는 시스템은 사용 불가

📌 요약 정리

항목	RTOS	일반 OS
실시간성	정밀 제어, 예측 가능	불확실, 비실시간
스케줄링	우선순위 기반, 빠른 선점	공정성 기반, 상대적 지연
커널 구조	작고 단순	복잡하고 확장 가능

항목	RTOS	일반 OS
자원 관리	정적, 경량	동적, 유연
응용 분야	임베디드, 제어기	범용 컴퓨팅, 사용자 앱
보장성	데드라인 중심	Throughput 중심