좋아, 나 완전 준비됐어.  
우리 **리눅스 커널 분석** 같이 하자! 🔥

어떻게 시작할까?

* 커널 구조부터 큰 그림을 잡을까?
* 아니면 바로 소스코드 (예: start\_kernel() 함수) 따라가면서 분석할까?
* 또는 특정 주제 (예: 프로세스 스케줄링, 메모리 관리, 시스템 콜)부터 파볼까?

너가 원하는 방식 알려줘.  
**내가 네 분석 파트너가 되어줄게.** ✨

어디서부터 시작하고 싶어? 👀

[유저 스페이스] ←→ [시스템 콜 인터페이스] ←→ [커널 스페이스]

커널 스페이스 내부:

├── 프로세스 관리 (Process Management)

├── 메모리 관리 (Memory Management)

├── 파일 시스템 (File Systems)

├── 네트워킹 (Networking)

├── 디바이스 드라이버 (Device Drivers)

├── 아키텍처 의존 코드 (Architecture-dependent Code)

└── 커널 서브시스템들 (ex. IPC, 보안 등)

| **영역** | **설명** |
| --- | --- |
| **시스템 콜 인터페이스** | 유저 프로그램이 커널에 요청하는 입구 (ex. read(), write() 호출) |
| **프로세스 관리** | fork(), exec(), schedule() 같은 프로세스 생성/스케줄링 관리 |
| **메모리 관리** | 가상 메모리, 페이지 테이블, 메모리 할당 (kmalloc, vmalloc) |
| **파일 시스템** | ext4, NFS 같은 파일시스템 핸들링 (디스크 읽기/쓰기) |
| **네트워킹** | TCP/IP 스택, 소켓 통신 |
| **디바이스 드라이버** | 하드웨어 제어 (USB, SSD, GPU 등) |
| **아키텍처 코드** | ARM, x86, RISC-V 등 CPU 아키텍처 별 차이 처리 |
| **기타** | IPC (Inter Process Communication), 보안 모듈 등 |

**진짜 기본 흐름**

부팅부터 운영까지 핵심 흐름만 뽑으면:

1. **부트로더** → 커널 이미지 로딩 (vmlinuz)
2. **커널 진입** → start\_kernel() 함수 실행
3. **메모리 초기화** → 페이지 테이블 설정, 슬랩/버디 할당자 준비
4. **프로세스 0 (swapper)** 생성 → 프로세스 관리 스타트
5. **프로세스 1 (init)** 생성 → 유저 스페이스로 넘어감
6. **드라이버 초기화** → 장치들 준비
7. **네트워킹 초기화**
8. **시스템 콜 대기 상태** → 유저 프로그램이 호출하기만 기다림

부트로더

↓

start\_kernel()

↓

초기화 과정 (메모리, 스케줄러, 파일 시스템, 드라이버)

↓

프로세스 0 (idle) → 프로세스 1 (init)

↓

User Space (shell, 프로그램 실행)

### 요약

* 커널은 \*\*'유저 프로그램을 돌려주고, 하드웨어를 관리하는 소프트웨어'\*\*야.
* 이걸 위해 내부에 수많은 서브시스템이 나뉘어 있어.
* 부팅하면 start\_kernel()부터 모든 게 시작된다.

**리눅스 커널 스케줄러 구성 (보통 이렇다)**

| **파일** | **역할** |
| --- | --- |
| kernel/sched/core.c | 스케줄러의 핵심 (컨텍스트 스위칭, 스케줄링 정책 등) |
| kernel/sched/fair.c | Completely Fair Scheduler (CFS) 구현 |
| kernel/sched/rt.c | 실시간 스케줄러 (RT 스케줄링) |
| kernel/sched/deadline.c | 데드라인 스케줄러 |
| kernel/sched/sched.h | 스케줄러 관련 데이터 구조체와 함수 선언 |
| include/linux/sched.h | 태스크(프로세스) 관련 구조체 (task\_struct) 정의 |

// ====== HEADER (구조체, define 등) ======

// ====== CORE (schedule(), context\_switch()) ======

// ====== CFS (pick\_next\_task\_fair()) ======

// ====== RT (pick\_next\_task\_rt()) ======

// ====== DEADLINE (pick\_next\_task\_dl()) ======

// ====== Main Scheduler (pick\_next\_task()) ======

유저 프로세스가 실행 →

커널로 진입 (ex. 인터럽트, 시스템 콜 등) →

schedule() 호출 →

pick\_next\_task() 호출 →

(CFS/RT/Deadline 중 하나 선택) →

만약 다른 프로세스가 선택되면 →

context\_switch() 호출 →

문맥 교환 (레지스터/메모리 스위칭) →

새 프로세스 실행

# task\_struct란?

**리눅스 커널이 프로세스/스레드를 관리하는 핵심 데이터 구조체**야.

쉽게 말해서,  
**하나의 프로세스 = 하나의 task\_struct** 인스턴스

모든 프로세스들은 이 task\_struct를 통해 관리되고,  
스케줄러도 이걸 가지고 놀아.

struct task\_struct {

// --- 프로세스 기본 정보 ---

pid\_t pid; // 프로세스 ID

pid\_t tgid; // 스레드 그룹 ID (멀티스레드용)

// --- 스케줄링 관련 ---

struct sched\_entity se; // CFS용 스케줄링 엔티티

struct rt\_entity rt; // RT 스케줄링 엔티티

struct dl\_entity dl; // Deadline 스케줄링 엔티티

int prio; // 현재 우선순위

int static\_prio; // 기본 우선순위

int normal\_prio; // 정책별 계산된 우선순위

unsigned int policy; // 스케줄링 정책 (SCHED\_NORMAL, SCHED\_FIFO 등)

struct list\_head run\_list; // Runqueue에 걸리는 리스트

// --- 메모리 관리 ---

struct mm\_struct \*mm; // 메인 메모리 영역 (코드, 데이터, 스택)

struct mm\_struct \*active\_mm; // 현재 활성 메모리 영역

// --- 컨텍스트 스위칭용 ---

struct thread\_struct thread; // CPU 레지스터 저장용 구조체

// --- 부모/자식 관계 ---

struct task\_struct \_\_rcu \*real\_parent; // 진짜 부모 프로세스

struct list\_head children; // 자식 프로세스 리스트

struct list\_head sibling; // 형제 프로세스 리스트

// --- 상태 관리 ---

volatile long state; // TASK\_RUNNING, TASK\_INTERRUPTIBLE 등

unsigned int flags; // 프로세스 상태 플래그

};

task\_struct

├── PID / TGID

├── 스케줄링 정보 (CFS/RT/DL)

├── 우선순위, 정책

├── 런큐 리스트 연결

├── 메모리 관리 (mm\_struct)

├── CPU 컨텍스트 (thread\_struct)

├── 부모/자식 관계

├── 상태(state) + 플래그

# 스케줄러와 task\_struct 연결

* schedule() 은 rq->curr (= task\_struct) 를 바꾼다.
* pick\_next\_task\_\*() 함수들은 run\_list를 타고 다음 task\_struct를 고른다.
* context\_switch() 는 thread\_struct 안의 레지스터 저장/복원한다.

즉,  
**task\_struct = 프로세스의 모든 것**  
이고,  
**스케줄러는 task\_struct를 선택하고, 바꾸는 역할**을 한다.

# 요약

| **항목** | **역할** |
| --- | --- |
| pid | 프로세스 ID |
| se, rt, dl | 스케줄링 정책별 엔티티 |
| mm, active\_mm | 프로세스 메모리 영역 |
| thread | CPU 레지스터 저장소 |
| run\_list | 런큐에 등록되는 리스트 노드 |
| state | 실행/슬립 등 프로세스 상태 |

# sched\_entity란?

**CFS(Completely Fair Scheduler)** 가  
태스크(task\_struct)를 스케줄링할 때 사용하는 **핵심 구조체**야.

즉,  
task\_struct 하나하나가  
→ **sched\_entity** 를 하나씩 품고 있고,  
→ CFS 스케줄러는 이 sched\_entity끼리 비교해서  
**누굴 먼저 실행할지** 결정한다.

struct sched\_entity {

struct load\_weight load; // 가중치 (CPU 점유율 영향)

struct rb\_node run\_node; // 레드-블랙 트리 노드

u64 vruntime; // 가상 런타임 (스케줄 기준)

u64 sum\_exec\_runtime; // 총 실행 시간

struct sched\_entity \*parent; // 부모 엔티티 (그룹 스케줄링 지원)

struct cfs\_rq \*cfs\_rq; // 내가 속한 CFS runqueue

struct cfs\_rq \*my\_q; // 내가 그룹이면 내 자식들의 runqueue

};

# 하나하나 설명 🔍

| **필드명** | **설명** |
| --- | --- |
| load | CPU 점유 "가중치". nice 값이나 priority에 따라 달라짐 |
| run\_node | Red-Black Tree에서 내 위치 (정렬용) |
| vruntime | 내 "가상 실행 시간". 이걸 기준으로 비교해서 공평하게 스케줄 |
| sum\_exec\_runtime | 지금까지 누적 실행한 시간 총합 |
| parent | 그룹 스케줄링할 때 부모 엔티티 |
| cfs\_rq | 내가 걸려 있는 runqueue (CFS 전용) |
| my\_q | 만약 내가 그룹이라면, 내 자식들의 runqueue |

task\_struct

└── sched\_entity

├── load

├── run\_node

├── vruntime

├── sum\_exec\_runtime

├── parent

├── cfs\_rq

└── my\_q

# 📚 Digging into the Linux Scheduler (요약)

## 1. 스케줄링 문제란 무엇인가?

* 운영체제는 모든 프로세스에 **공평하게 CPU를 배분**해야 한다.
* 어떤 프로세스는 **우선순위가 높고**, 어떤 프로세스는 **낮다**.
* 고우선 프로세스가 CPU를 독점하지 못하게 해야 한다.
* 고우선 프로세스가 생기면 **현재 실행 중인 프로세스를 강제로 중단(preemption)** 할 수 있어야 한다.
* 시스템이 불안정해지지 않도록 해야 한다.

## 2. 리눅스 스케줄러의 큰 구조

* **Runqueue (rq)**: CPU 하나당 하나씩 존재. 실행 대기 중인 모든 태스크를 관리.
* **스케줄링 클래스**(Scheduling classes):
  + 커널 개발자들이 다양한 요구사항을 충족시키기 위해 **스케줄러를 여러 클래스로 분리**했다.
  + 각 클래스는 **자기만의 규칙**에 따라 태스크를 관리한다.

## 3. 스케줄링 클래스 종류

| **클래스** | **설명** |
| --- | --- |
| **DL (Deadline)** | 데드라인 내에 끝나야 하는 작업 (예: 오디오/비디오 처리) |
| **RT (Real-time)** | 즉각적인 응답이 필요한 작업 (예: IRQ 스레드) |
| **FAIR (CFS)** | 일반적인 사용자 프로세스 (공정 스케줄링) |
| **IDLE** | 위 클래스들이 모두 비었을 때 CPU를 놀리는 idle task |

* 스케줄링 클래스는 **Linked List** 형태로 연결되어 있다. (DL → RT → FAIR → IDLE 순서)

## 4. 스케줄링 클래스 구현 방식

* 각 스케줄링 클래스는 **sched\_class 구조체**를 통해 관리된다.
* sched\_class는 **함수 포인터 테이블**(jump table)이다.
  + 예: pick\_next\_task, enqueue\_task, dequeue\_task 등

## 5. pick\_next\_task 함수

* **리눅스 스케줄러의 심장**.
* 동작 방식:

c

복사편집

for (class = sched\_class\_highest; class; class = class->next) {

p = class->pick\_next\_task(rq, prev, &rf);

if (p)

return p;

}

* 즉, **스케줄링 클래스를 순서대로 순회하면서**
  + 실행할 프로세스를 찾으면 **즉시 리턴**한다.
* 없으면 idle 프로세스가 실행된다.

## 6. schedule() 함수

* 커널 코드 전체에서 **가장 많이 호출되는 함수** 중 하나.
* 어떤 시점에 CPU에 무슨 프로세스를 돌릴지 결정할 때 호출된다.
* 내부적으로 \_\_schedule(bool preempt)를 호출해서 실제 스케줄링을 수행한다.
* 중요한 규칙:  
  **Critical Section (락 잡고 있는 부분)** 에서는 schedule() 호출하면 안 된다.

## 7. scheduler\_tick()

* CPU마다 **타이머 인터럽트**가 주기적으로 발생해서
* scheduler\_tick() 이 호출된다.
* 이걸 통해 스케줄러가 주기적으로 **태스크 상태 업데이트**하고,  
  **필요시 preemption**(강제 전환) 할 준비를 한다.

# ✨ 요약 키포인트

| **키워드** | **의미** |
| --- | --- |
| **Runqueue** | CPU 하나당 하나씩 존재하는 태스크 리스트 |
| **Scheduling Classes** | DL → RT → CFS → IDLE 순서로 관리 |
| **sched\_class** | 각 스케줄링 클래스의 함수 포인터 모음 |
| **pick\_next\_task()** | 다음 실행할 태스크를 선택하는 심장부 |
| **schedule()** | 스케줄링이 필요할 때 호출하는 API |
| **scheduler\_tick()** | 타이머 인터럽트로 스케줄러를 깨움 |
| 📚 cgroups (Control Groups) 완전 기초부터 정리1. cgroups란? **리눅스 커널에서 특정 프로세스 집합에 대해 리소스(CPU, 메모리, I/O, 네트워크 사용량)를 제한하거나 측정하거나 격리하는 기능.**  즉,   * CPU를 20%만 쓰게 * 메모리는 512MB 넘으면 죽여 * 디스크 쓰기 속도를 10MB/s로 제한해   이런 걸 커널 레벨에서 **강제**할 수 있어.  **컨테이너 = 결국 cgroups으로 리소스를 제한한 리눅스 프로세스**다! 2. cgroups 핵심 키워드  | **항목** | **설명** | | --- | --- | | **group** | 하나 이상의 프로세스 집합 | | **controller(subsystem)** | 관리할 리소스 종류 (cpu, memory, blkio 등) | | **hierarchy** | 그룹들을 트리 구조로 관리 | | **tasks** | 그룹에 속한 프로세스 ID 리스트 |  3. cgroups 컨트롤러 종류  | **컨트롤러** | **관리하는 것** | | --- | --- | | cpu | CPU 사용량 제어 | | cpuacct | CPU 사용량 모니터링 | | cpuset | 특정 CPU Core에만 실행 제한 | | memory | 메모리 사용량 제한 | | blkio | 디스크 I/O 속도 제한 | | net\_cls, net\_prio | 네트워크 대역폭/우선순위 제어 |  4. 기본 흐름 **cgroups가 리소스를 관리하는 방법**   1. 커널이 cgroups 가상 파일시스템(mount된 /sys/fs/cgroup)을 통해 2. 각 그룹별로 디렉토리 생성 3. 거기에 프로세스 추가 (tasks 파일에 pid 등록) 4. 리소스 제어 설정 (cpu.max, memory.max 등 설정) 5. 커널이 자동으로 제한/격리 적용  5. 그림으로 보면 lua  복사편집  /sys/fs/cgroup/  ├── cpu/  │ ├── group1/  │ │ ├── tasks (pid 목록)  │ │ ├── cpu.max (CPU 제한)  │ ├── group2/  │ ├── tasks  │ ├── cpu.max  ├── memory/  │ ├── group1/  │ │ ├── tasks  │ │ ├── memory.max (메모리 제한)  컨테이너마다 이렇게 디렉터리가 만들어지고 관리돼. 6. cgroups가 Kubernetes랑 무슨 상관?  * 쿠버네티스에서 resources: limits: 설정하면 * kubelet이 컨테이너를 만들 때 * 그 컨테이너 프로세스를 특정 cgroup에 등록하고 * CPU, 메모리 제한을 적용해주는 거야.   ✅ **즉, 컨테이너 리소스 격리 = cgroups 설정** ✨ 요약 **cgroups는 리눅스 프로세스들의 리소스를 강제 관리하는 시스템이다.** **컨테이너는 사실상 cgroups 안에서 돌고 있는 고립된 프로세스다.** 추가로 이어서 공부할 수 있는 것  | **번호** | **주제** | | --- | --- | | 1 | cgroups 버전 1 vs 버전 2 차이점 | | 2 | 실제 cgroup 파일 직접 조작해서 제한 걸어보기 (실습) | | 3 | Kubernetes가 cgroups를 어떻게 설정하는지 흐름도 | | 4 | cgroups + namespace로 컨테이너 직접 만드는 mini 프로젝트 |   **/sys/fs/cgroup/**  **├── cpu/**  **│ ├── group1/**  **│ │ ├── tasks (pid 목록)**  **│ │ ├── cpu.max (CPU 제한)**  **│ ├── group2/**  **│ ├── tasks**  **│ ├── cpu.max**  **├── memory/**  **│ ├── group1/**  **│ │ ├── tasks**  **│ │ ├── memory.max (메모리 제한)** |  |

# 💬 결론

**리눅스 스케줄러는**  
복잡한 작업들을 클래스로 분리하고,  
심플한 흐름으로 급한 작업부터 공정하게 CPU를 배분하는 구조다.

이제 이 기반으로  
**CFS 세부 로직** (\_\_pick\_next\_task\_fair(), vruntime 조정 등)  
**RT/DL 스케줄링**  
**Multi-core 확장 (SMP)**