7. 스케줄링 및 우선순위

7.1 리눅스 스케줄러 개요 (CFS 등)

개요

리눅스 커널은 **멀티태스킹 운영체제**다.
→ **여러 프로세스(또는 스레드)가 동시에 실행되는 것처럼 보이게 하는 것**이 바로 스케줄러의 역할이다.

스케줄러(Scheduler)는 **CPU를 어떤 프로세스에 얼마나 할당할 것인지 결정**하는 커널의 핵심 구성 요소다.

- **공정성(Fairness)** 유지
- **응답 시간(Response Time)** 최적화
- **처리량(Throughput)** 증대
- **우선순위(Priority)** 존중

리눅스의 주요 스케줄링 정책

리눅스는 여러 가지 **스케줄링 정책(Scheduling Policy)** 를 제공한다:

정책	설명
CFS (Completely Fair Scheduler)	기본 정책 (기본 프로세스용)
SCHED_FIFO	고정 우선순위 실시간 정책
SCHED_RR	고정 우선순위 + 라운드로빈 실시간 정책
SCHED_DEADLINE	Deadline 기반 실시간 스케줄링 (최신 리눅스 지원)

Completely Fair Scheduler (CFS)

등장 배경

과거 리눅스 스케줄러는 **O(1) 스케줄러** → 복잡, Starvation(기아 상태) 문제 발생 가능.
→ 2.6.23 커널부터 **CFS** 도입.

기본 개념

CFS는 이름 그대로 **공정함(Fairness)** 을 최대한 보장하려는 설계다.

- 각 프로세스는 **virtual runtime (vruntime)** 라는 가상의 실행 시간을 가진다.
- CPU는 항상 **vruntime가 가장 작은 프로세스**에게 할당된다 → 가장 적게 실행된 프로세스가 우선.
- CPU 시간 할당량은 **프로세스의 nice 값(우선순위)** 에 비례해서 가중치를 가짐.

핵심 자료구조

```
struct cfs_rq {
    struct rb_root tasks_timeline;  // Red-Black Tree
    ...
};
```

- CFS는 **Red-Black Tree**를 사용해 runnable 프로세스들을 정렬한다.
- 가장 leftmost node → 가장 적게 실행된 프로세스.

Virtual Runtime(vruntime)

- 실행 시간이 가중치(weight)를 적용한 누적값으로 기록됨.
- 가중치는 **nice 값**에 따라 달라짐.

→ **낮은 vruntime → 실행 우선 순위 높음**

동작 순서

1️⃣ 프로세스가 CPU 사용 → vruntime 증가
2️⃣ Preempt 발생 시 Red-Black Tree에 다시 삽입
3️⃣ 다음 실행 프로세스는 vruntime가 가장 작은 것 선택

장점

- **기아 상태 없음** → 모든 프로세스는 언젠가는 CPU를 받게 됨.
- **부드러운 CPU 할당** → Real-time 특성은 없지만 일반적인 Interactive 시스템에 매우 적합.
- **비선형적 우선순위 조정 가능** → nice 값 활용.

실시간 스케줄링 정책

SCHED_FIFO

- **고정 우선순위(Static Priority)** 적용.
- 선입선출(First In First Out), 타임 슬라이스 없음.
- 실행 중인 프로세스보다 **우선순위 높은 프로세스가 Ready 상태가 되면 즉시 Preempt 발생**.
- 우선순위가 동일한 경우 FIFO 순서로 실행.

SCHED_RR

- **고정 우선순위 + 라운드로빈**.
- 타임 슬라이스 존재 → 동일 우선순위 프로세스 간 라운드로빈 적용.

SCHED_DEADLINE

- **Earliest Deadline First (EDF) 알고리즘 기반**.
- 각 태스크에 **deadline, period, runtime** 등 제약조건 설정 가능.
- **하드 실시간 시스템**이나 **멀티미디어 처리** 등에 활용.

CFS vs 실시간 스케줄링

특징	CFS	SCHED_FIFO / SCHED_RR
목적	일반 프로세스용	실시간 프로세스용
우선순위	nice 값 기반 가중치	고정 우선순위
선점성	있음	있음
기아 방지	우수	주의 필요 (낮은 우선순위는 실행 못할 수 있음)
부드러운 응답성	우수	실시간 요구 가능

커널 파라미터 (CFS 관련)

- `/proc/sys/kernel/sched_latency_ns` → 타임 슬라이스의 목표 latency
- `/proc/sys/kernel/sched_min_granularity_ns` → 최소 granularity
- `/proc/sys/kernel/sched_wakeup_granularity_ns` → wakeup 선호도 조정

튜닝 가능 → 고성능 서버에서는 latency를 줄이거나 CPU-intensive batch 작업에 유리하게 조정 가능.

사용자 API

정책 변경

```
#include <sched.h>
#include <unistd.h>
#include <stdio.h>

struct sched_param param;
param.sched_priority = 10;

sched_setscheduler(getpid(), SCHED_FIFO, &param);
```

→ 현재 프로세스의 스케줄링 정책 변경 가능 (권한 필요).

정책 조회

```
int policy = sched_getscheduler(getpid());
```

→ 현재 프로세스의 정책 확인.

실습 아이디어

- CFS 기반 → 여러 프로세스에 다른 nice 값을 부여 → CPU 할당 비율 관찰
- SCHED_FIFO → Preemption 테스트 (고정 우선순위 적용)
- SCHED_DEADLINE → deadline 기반 작업 시뮬레이션

정리

구성 요소	설명
CFS	리눅스 기본 스케줄러, Red-Black Tree 기반
vruntime	가상 실행 시간 → CPU 할당 우선순위 결정
SCHED_FIFO	고정 우선순위, FIFO 실행
SCHED_RR	고정 우선순위, 라운드로빈
SCHED_DEADLINE	EDF 기반, deadline-aware 스케줄링
스케줄링 파라미터 튜닝	`/proc/sys/kernel/sched_*` 노드 활용

결론

리눅스 스케줄러는:

- **CFS 기반으로 일반 프로세스에 공정성을 제공**하고
- **실시간 요구가 있으면 별도 정책(SCHED_FIFO/RR/DEADLINE)** 을 사용해 실시간 처리 요구도 만족시킨다.

스케줄러는 시스템 성능과 응답성의 핵심 요소이므로 서비스 성격에 따라 적절한 정책 튜닝이 매우 중요하다.

7.2 `nice()`, `setpriority()` 함수

개요

리눅스에서는 일반 프로세스(기본 CFS 스케줄링 대상)의 **상대적 우선순위**를 조정할 수 있다.
→ 이를 통해 **CPU 점유율을 높이거나 낮추는 힌트**를 스케줄러에 제공할 수 있다.

- `nice()` → 프로세스의 "nice 값" 조정 → 가중치 변화로 **간접적 우선순위** 조정
- `setpriority()` / `getpriority()` → 프로세스/그룹/사용자의 **nice 값 직접 설정/조회**

주의:

- 실시간 정책(`SCHED_FIFO`, `SCHED_RR`)에는 적용되지 않음 → **CFS 대상 프로세스**에서만 사용 의미가 있음.

---

nice 값이란?

- `nice 값` → -20 ~ +19 범위
  - 낮을수록 **높은 우선순위**
  - 높을수록 **낮은 우선순위** (CPU 양보 → "친절(nice)한" 프로세스)

nice 값	의미
-20	가장 높은 우선순위 (최대한 CPU 차지)
0	기본 우선순위
+19	가장 낮은 우선순위 (가장 CPU 양보)

→ 실제 스케줄러에서는 nice 값을 기반으로 **weight** 를 계산 → **vruntime 증가 속도에 영향**.

---

1 `nice()` 함수

헤더

```
#include <unistd.h>
```

원형

```
int nice(int inc);
```

설명

- **현재 프로세스의 nice 값**에 `inc` 를 더해서 새로운 nice 값을 설정
- 반환값 → 새로운 nice 값
- 실패 시 -1 반환 후 `errno` 설정

사용 예

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

int main() {
    int new_nice = nice(5);  // nice 값 +5 증가
    printf("New nice value: %d\n", new_nice);

    // CPU를 쓰는 루프
    while (1) {
        // 작업 수행
    }
    return 0;
}
```

권한

- 일반 사용자 → **음수로 변경 불가** (즉, nice 값을 높이는 것만 가능)
- 루트 사용자 → **전체 범위 사용 가능**

2️⃣ `setpriority()` / `getpriority()`

헤더

```
#include <sys/resource.h>
```

원형

```
int setpriority(int which, id_t who, int prio);
int getpriority(int which, id_t who);
```

매개변수

매개변수	의미
`which`	어떤 범위 대상인지 (PRIO_PROCESS, PRIO_PGRP, PRIO_USER)
`who`	대상 ID (pid, pgrp id, user id) → 0은 호출자 자신
`prio`	설정할 nice 값 (-20 ~ +19)

상수

상수	의미
`PRIO_PROCESS`	특정 프로세스 대상
`PRIO_PGRP`	프로세스 그룹 대상
`PRIO_USER`	사용자 전체 프로세스 대상

사용 예제

`setpriority()`

```
#include <stdio.h>
#include <sys/resource.h>
#include <unistd.h>

int main() {
    pid_t pid = getpid();
    int ret;

```

```
    ret = setpriority(PRIO_PROCESS, pid, 10);  // nice 값 10으로 설정
    if (ret == -1) {
        perror("setpriority");
    }

    int prio = getpriority(PRIO_PROCESS, pid);
    printf("Current nice value: %d\n", prio);

    // CPU를 쓰는 루프
    while (1) {
    }

    return 0;
}
```

실행 결과 예시

```
Current nice value: 10
```

권한

사용자 유형	허용 동작
일반 사용자	자신이 시작한 프로세스에 대해 **nice 값 높이기(+ 방향)만 가능**
루트 사용자	모든 프로세스에 대해 **전체 범위 설정 가능** (-20 ~ +19)

→ 일반 사용자는 **우선순위를 낮출 수는 없다** (보안상 이유로 다른 사용자 작업을 방해하지 못하도록 제한).

3️⃣ nice 값과 CPU 점유율

nice 값 변화는 **CPU weight 계산**에 영향을 준다.

nice 값 변화	결과
감소(-20 쪽으로)	프로세스에 더 많은 CPU 할당
증가(+19 쪽으로)	프로세스가 CPU를 다른 프로세스에 양보

주의:

- CFS는 "정확한 우선순위 절대값"을 보장하는 것이 아님 → **상대적 비율 기반 CPU 사용량** 변화가 발생.

4️⃣ 실습 예제: CPU 할당 효과 확인

1️⃣ 터미널 1 → 기본 nice 값으로 실행

```
$ ./cpu_loop
```

2️⃣ 터미널 2 → nice 값 높여서 실행

```
$ nice -n 10 ./cpu_loop
```

3️⃣ `top` 또는 `htop` 으로 두 프로세스의 CPU 사용률 비교 → nice 값 높은 쪽이 CPU 사용량 낮음.

5️⃣ 정리

기능	사용 함수
현재 프로세스의 nice 값 증가	`nice(int inc)`
프로세스/그룹/사용자 nice 값 설정	`setpriority()`
프로세스/그룹/사용자 nice 값 조회	`getpriority()`

- **기본적으로 CFS 대상 프로세스만 사용 의미 있음**.
- 실시간 프로세스(SCHED_FIFO/RR/DEADLINE)는 별도 **우선순위 API 사용** 필요 (`sched_setscheduler()`).

결론

- `nice()` / `setpriority()` 는 **비실시간 프로세스의 상대적 CPU 우선순위**를 조정하는 도구다.
- 시스템에서:
  - **백그라운드 작업** → nice 값 높게 설정 (CPU 점유율 낮춤)
  - **고성능 작업** → 루트 권한으로 nice 값 낮게 설정 가능.

→ 서버 성능 튜닝, 데스크탑 UI 응답성 개선 등에 실제로 유용하게 사용된다.

7.3 실시간 스케줄링 (`SCHED_FIFO`, `SCHED_RR`)

개요

리눅스 커널은 일반 프로세스(CFS 기반) 외에 **실시간(Real-Time, RT) 프로세스** 지원도 제공한다.
→ 특정 프로세스가 **정해진 시간 내에 반드시 실행될 필요**가 있는 경우 사용.

대표적 실시간 스케줄링 정책:

정책	설명
`SCHED_FIFO`	고정 우선순위, FIFO 순서 실행

정책	설명
`SCHED_RR`	고정 우선순위, 라운드로빈 실행

→ 둘 다 **고정 우선순위(Static Priority)** 기반이며 일반 CFS 프로세스보다 **우선적 실행권**을 가진다.

특징

고정 우선순위 (Static Priority)

- 각 RT 프로세스는 **0보다 큰 우선순위(1~99)** 를 가짐 → 값이 높을수록 우선순위 높음.
- RT 프로세스는 항상 **일반 프로세스(CFS)보다 우선됨** → RT 프로세스가 존재하면 CFS 프로세스는 CPU 사용 불가.

스케줄링 비교

정책	선점 여부	실행 순서	타임슬라이스
CFS	O	vruntime 기반	동적
SCHED_FIFO	O	우선순위 높은 프로세스 → 동일 우선순위 FIFO	없음
SCHED_RR	O	우선순위 높은 프로세스 → 동일 우선순위 라운드로빈	고정 타임슬라이스

1️⃣ `SCHED_FIFO`

특징

- 선입선출(First In First Out) 방식
- 동일 우선순위에서는 **FIFO 순서로 실행** → CPU를 자발적으로 양보(`sched_yield()`)하거나 Block 될 때까지 CPU 점유.
- 타임슬라이스 없음 → **영원히 실행 가능** → 기아 상태 주의 필요.

사용 예

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

int main() {
    struct sched_param param;
    param.sched_priority = 50;

    if (sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
        perror("sched_setscheduler");
        return 1;
    }

    while (1) {
        // 실시간 작업 수행
```

```
    }

    return 0;
}
```

→ 이 프로세스는 **우선순위 50의 SCHED_FIFO 프로세스**가 되어 일반 프로세스를 모두 선점.

---

2 SCHED_RR

특징

- Round Robin → **타임슬라이스(Time Quantum)** 존재
- 동일 우선순위 프로세스가 **라운드로빈 방식으로 CPU 할당**됨.
- **타임슬라이스 만료 시 스케줄러에 의해 강제 Preempt 발생**.

사용 예

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

int main() {
    struct sched_param param;
    param.sched_priority = 50;

    if (sched_setscheduler(0, SCHED_RR, &param) == -1) {
        perror("sched_setscheduler");
        return 1;
    }

    while (1) {
        // 실시간 작업 수행
    }

    return 0;
}
```

→ 동일 우선순위의 다른 SCHED_RR 프로세스가 있다면 **타임슬라이스 단위로 CPU 교대**됨.

---

3 우선순위 범위

```c
int min_priority = sched_get_priority_min(SCHED_FIFO);
int max_priority = sched_get_priority_max(SCHED_FIFO);

printf("FIFO priority range: %d - %d\n", min_priority, max_priority);
```

→ 일반적으로:

정책	우선순위 범위
SCHED_FIFO / SCHED_RR	1 ~ 99

주의:

- 우선순위 0은 CFS용 프로세스 전용 → 실시간 정책에서는 사용 불가.

---

4 `sched_setscheduler()` 함수

원형

```
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);
```

매개변수	의미
pid	대상 프로세스 PID (0 → 호출자 자신)
policy	`SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER` (CFS)
param	`sched_param` 구조체 → `sched_priority` 필드 사용

→ 성공 시 0 반환, 실패 시 -1 + `errno` 설정.

우선순위 조회

```
int policy = sched_getscheduler(pid);
```

→ 현재 프로세스의 스케줄링 정책 확인.

---

5 실시간 정책 주의사항

- 실시간 프로세스는 **기아 상태 발생 가능**:
  - 높은 우선순위 실시간 프로세스가 계속 실행되면 **낮은 우선순위 RT 프로세스와 CFS 프로세스가 실행되지 못할 수 있음**.
- RT 프로세스를 **root 권한** 으로만 설정 가능.
- 실시간 정책 사용 시 시스템의 다른 서비스에 **영향 미칠 수 있음** → 주의 깊게 사용해야 함.

---

6 CFS vs SCHED_FIFO / SCHED_RR

특징	CFS	SCHED_FIFO	SCHED_RR
기본 대상	일반 프로세스	실시간 프로세스	실시간 프로세스
우선순위 범위	nice (-20 ~ +19)	static priority (1~99)	static priority (1~99)

특징	CFS	SCHED_FIFO	SCHED_RR
스케줄링 방식	vruntime 기반	FIFO	라운드로빈
타임슬라이스	동적	없음	고정
선점 여부	있음	있음	있음
기아 방지	있음	없음	없음
응용 분야	일반 서비스	실시간 제어, 오디오	실시간 제어, 오디오

7️⃣ 실습 아이디어

- **SCHED_FIFO / SCHED_RR 프로세스 2개 이상 생성 → CPU 점유율 변화 관찰**
- SCHED_RR에서 **타임슬라이스 변화 효과 확인**
- SCHED_FIFO 프로세스 → `sched_yield()` **사용 여부에 따른 실행 변화 실험**

결론

- 리눅스는 고성능 실시간 처리를 위해 **SCHED_FIFO**, **SCHED_RR** 정책을 제공한다.
- 주로:
  - **로봇 제어**
  - **멀티미디어(오디오/비디오)**
  - **네트워크 패킷 처리**
  - **고속 데이터 수집 장치 제어**
- 실시간 프로세스는 **커널 튜닝과 정확한 테스트가 필수적**이다 → 시스템 전체 안정성에 영향을 줄 수 있기 때문.

7.4 CPU 점유율 측정 (`top`, `ps`, `/proc`)

개요

운영체제에서 프로세스/시스템의 **CPU 사용량**을 측정하는 것은:

- **성능 모니터링**
- **병목 분석**
- **리소스 튜닝**

에 필수적이다.

리눅스에서는:

도구 / 인터페이스	목적
`top`	실시간 시스템 전체/프로세스 CPU 모니터링
`ps`	특정 시점의 프로세스 상태 확인

도구 / 인터페이스	목적
`/proc`	프로세스/시스템 상태에 대한 내부 인터페이스

1️⃣ CPU 점유율의 의미

CPU 점유율(%) → 특정 프로세스가 **전체 CPU 시간에서 차지하는 비율**.

- 단일 코어 시스템 → 최대 100%
- 다중 코어 시스템 (N코어) → 최대 N * 100%
  예: 4코어 → 400%

CPU 사용 시간은 **유저 모드(usr)** 와 **커널 모드(sys)** 로 구분할 수 있다.

2️⃣ `top` 명령어

기본 사용법

```
$ top
```

→ 실시간으로 프로세스 목록과 CPU/메모리/로드 평균 등을 출력.

화면 구성

항목	의미
%us	사용자 영역 (User space) CPU 사용률
%sy	커널 영역 (System space) CPU 사용률
%ni	Nice값이 조정된 프로세스의 CPU 사용률
%id	Idle (유휴 상태) 비율
%wa	I/O 대기 비율
%hi	하드웨어 인터럽트 처리 비율
%si	소프트웨어 인터럽트 처리 비율
%st	Steal time (가상화 환경에서 다른 VM에 의해 빼앗긴 시간)

프로세스별 주요 항목

항목	의미
%CPU	프로세스의 CPU 사용률 (%)
TIME+	누적 CPU 시간

주요 키

- `P` → CPU 사용률 순 정렬
- `1` → CPU 개별 코어 사용률 표시
- `h` → 도움말
- `q` → 종료

3️⃣ `ps` 명령어

기본 사용법

```
$ ps aux
```

필드	의미
%CPU	CPU 사용률 (%)
TIME	누적 CPU 시간 (MM:SS)
PID	프로세스 ID
USER	프로세스 소유자
COMMAND	실행 중인 명령어

특정 프로세스 모니터링

```
$ ps -p <PID> -o %cpu,time,cmd
```

예:

```
$ ps -p 1234 -o %cpu,time,cmd
```

4️⃣ `/proc` 인터페이스

리눅스 커널은 `/proc` 파일시스템을 통해 **커널 내부 상태**를 가상 파일 형태로 노출한다.

`/proc/stat`

```
$ cat /proc/stat
```

예시:

```
cpu  123456 2345 67890 987654321 12345 678 910 0 0 0
```

항목	의미
user	사용자 모드에서의 시간
nice	nice 값 적용된 프로세스의 사용자 모드 시간
system	커널 모드 시간
idle	유휴 시간
iowait	I/O 대기 시간
irq	하드웨어 인터럽트 처리 시간
softirq	소프트웨어 인터럽트 처리 시간

→ **CPU 전체 사용률**을 직접 계산 가능.

CPU 사용률 계산법 (기초)

```
total_time = user + nice + system + idle + iowait + irq + softirq
busy_time = total_time - idle - iowait
cpu_usage% = (busy_time / total_time) * 100
```

→ 샘플링 주기 1초 이상으로 2번 읽어 차이로 계산.

`/proc/<PID>/stat`

```
$ cat /proc/1234/stat
```

필드	의미 (일부 발췌)
utime	사용자 모드에서 사용한 jiffies
stime	커널 모드에서 사용한 jiffies
cutime	자식 프로세스의 utime
cstime	자식 프로세스의 stime
starttime	프로세스 시작 시점 (jiffies)

CPU 사용 시간 = utime + stime

→ 이를 시간 단위로 변환 시:

```
cpu_time_sec = (utime + stime) / HZ;
```

→ HZ는 시스템마다 다름 (일반적으로 100 또는 1000).

5️⃣ 정리

도구	사용 목적
`top`	시스템 전체 / 실시간 CPU 모니터링
`ps`	특정 시점에서 프로세스 상태 조회
`/proc/stat`	전체 CPU 사용 시간 확인 → 직접 계산 가능
`/proc/<PID>/stat`	개별 프로세스의 누적 CPU 사용 시간 확인 가능

실습 아이디어

- **top 사용법 익히기** → `1`, `P`, `H` 키 사용 실험.
- **ps로 특정 프로세스 주기적 모니터링** → `watch ps -p <PID> -o %cpu,time,cmd`
- **/proc/stat 기반 CPU 사용률 직접 계산 프로그램 구현하기** (자주 성능 분석 도구에서 사용됨).
- **/proc/`<PID>`/stat 기반 프로세스 CPU 사용 시간 모니터링 프로그램 구현하기** → profiling 시 유용.

결론

- 리눅스에서는 `top`, `ps`, `/proc` 를 조합해서 **정확하고 다양한 시각으로 CPU 사용률을 모니터링** 할 수 있다.
- 고성능 시스템에서는 `/proc` 기반의 **정밀 샘플링 도구(top, htop, perf 등)** 를 통해 실시간 분석이 가능하다.
- 성능 튜닝 / 병목 분석 시 **CPU 점유율 변화 패턴**을 정확히 보는 것이 매우 중요하다.

📎 실습

`nice()` 효과 확인 실험

목표

- 프로세스가 `nice()` **를 통해 nice 값을 높이면** → CPU 점유율이 낮아지는지 확인.
- 프로세스가 **기본 nice 값(0)일 때와 nice 값을 높인 경우** → CPU 사용률 차이를 비교.

실험 구성

준비

- 동일한 CPU intensive(계산 중심) 프로그램을 여러 개 실행.
- 각 프로세스에 **서로 다른 nice 값 적용**.
- `top` 또는 `ps` 를 이용해 CPU 점유율 비교.

1️⃣ 실험용 CPU intensive 프로그램 (cpu_loop.c)

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

int main(int argc, char *argv[]) {
    int n = 0;
    if (argc > 1) {
        n = atoi(argv[1]);
    }

    // nice 값 증가
    if (n != 0) {
        int ret = nice(n);
        if (ret == -1) {
            perror("nice");
        } else {
            printf("Set nice increment to %d, new nice: %d\n", n, ret);
        }
    }

    // CPU intensive loop
    volatile unsigned long long counter = 0;
    while (1) {
        counter++;
    }

    return 0;
}
```

컴파일

```
$ gcc -o cpu_loop cpu_loop.c
```

2️⃣ 실험 실행 방법

터미널 1: 기본 nice 값 (0)

```
$ ./cpu_loop 0
```

터미널 2: nice +5 적용

```
$ ./cpu_loop 5
```

터미널 3: nice +10 적용

```
$ ./cpu_loop 10
```

---

3 관찰 방법

```
$ top
```

→ `top`에서 각 프로세스의:

항목	의미
%CPU	해당 프로세스 CPU 점유율
NI	nice 값 (기본 0, 높을수록 낮은 우선순위)

예상 결과:

프로세스	nice 값	예상 %CPU (상대적)
cpu_loop 1	0	높음 (최우선)
cpu_loop 2	+5	중간
cpu_loop 3	+10	낮음

→ nice 값이 높아질수록 **vruntime 증가 속도가 빨라짐** → CPU 사용률이 상대적으로 줄어듦.

---

4 분석

- **CFS 기반 스케줄러는 엄밀히 비선형적 CPU weight** 를 적용하므로 정확히 선형 비율은 아님.
- nice 값 변화에 따른 **CPU 점유율 변화 패턴** 을 눈으로 확인할 수 있음.

참고: weight 계산은 다음 공식 사용:

```
weight = 1024 / (1.25 ^ nice_value)
```

→ nice +1 당 약 10% 가량 weight 감소.

---

5 실험 주의사항

- CPU core 수가 **1개일수록 효과가 더 뚜렷하게 나타남**.
- 다중 core 시스템에서는 `top`에서 **개별 프로세스의 %CPU** 를 보고 비교.
- 실시간 프로세스(SCHED_FIFO, SCHED_RR)에는 `nice()` 적용되지 않음 → CFS 프로세스용 실험임.

---

6 결론

이 실험을 통해:

- `nice()` → **CPU 우선순위에 직접적인 영향을 준다는 것**을 확인할 수 있음.
- nice 값 높을수록 → **다른 프로세스에게 CPU 사용 기회 양보** → 점유율 감소.

→ 시스템 튜닝 시 **백그라운드 작업에 높은 nice 값 적용**이 좋은 실무 팁이다.

---

7 정리 표

기능	도구
nice 값 설정	`nice()` 함수 또는 `nice` 명령어
실시간 관찰	`top`
정적 관찰	`ps aux`

`clock_gettime()`으로 태스크 시간 측정

개요

`clock_gettime()` 은 리눅스에서 고해상도 시간(High-resolution time)을 가져오는 대표적인 함수다.
→ 시스템의 여러 **Clock source** 중 하나를 선택해서 **정확한 시간 측정** 가능.

주요 용도

- **실행 시간 측정** (Benchmarking)
- **응답 시간 측정** (Latency 측정)
- **성능 튜닝 시 모듈별 소요 시간 측정**

---

1 함수 원형

```
#include <time.h>

int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

매개변수

매개변수	설명
clk_id	사용할 시계 종류 (clock source)
tp	결과가 저장될 `struct timespec` 포인터

반환값

- 성공 시 `0`
- 실패 시 `-1` + `errno` 설정

---

2 `struct timespec`

```
struct timespec {
    time_t tv_sec;        // 초 (seconds)
    long   tv_nsec;       // 나노초 (nanoseconds)
};
```

- 총 시간 = `tv_sec + tv_nsec * 10^-9` 초 단위로 사용 가능.

---

3 주요 Clock Source

Clock ID	의미
CLOCK_REALTIME	시스템 전체의 실시간 clock (UTC 기준, settimeofday 등으로 변경 가능)
CLOCK_MONOTONIC	부팅 이후 경과 시간 (단조 clock, 항상 증가) → **시간 측정에 가장 적합**
CLOCK_PROCESS_CPUTIME_ID	호출 프로세스가 사용한 CPU 시간
CLOCK_THREAD_CPUTIME_ID	호출 스레드가 사용한 CPU 시간

→ 태스크(프로세스/스레드) 실행 시간 측정 → CLOCK_MONOTONIC 또는 CLOCK_PROCESS_CPUTIME_ID 추천

---

4 기본 예제: 태스크 실행 시간 측정

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>

void busy_work() {
    volatile unsigned long long counter = 0;
    for (unsigned long long i = 0; i < 1000000000ULL; i++) {
        counter++;
    }
}

double timespec_to_sec(struct timespec *ts) {
    return ts->tv_sec + ts->tv_nsec / 1e9;
}

int main() {
    struct timespec start, end;

```

```
    // 측정 시작
    clock_gettime(CLOCK_MONOTONIC, &start);

    // 측정 대상 작업
    busy_work();

    // 측정 종료
    clock_gettime(CLOCK_MONOTONIC, &end);

    double start_sec = timespec_to_sec(&start);
    double end_sec = timespec_to_sec(&end);
    double elapsed = end_sec - start_sec;

    printf("Elapsed time: %.6f seconds\n", elapsed);

    return 0;
}
```

실행 예시

```
Elapsed time: 1.234567 seconds
```

---

5 실전 적용 패턴

고성능 타이밍 측정 패턴

```
struct timespec t0, t1;

clock_gettime(CLOCK_MONOTONIC, &t0);
// ... 측정 대상 코드 ...
clock_gettime(CLOCK_MONOTONIC, &t1);

double elapsed_sec = (t1.tv_sec - t0.tv_sec) + (t1.tv_nsec - t0.tv_nsec) / 1e9;
printf("Elapsed: %.9f sec\n", elapsed_sec);
```

왜 CLOCK_MONOTONIC?

CLOCK	측정에 적합?	이유
CLOCK_REALTIME	X	시스템 시간 변경(settimeofday 등)에 영향을 받음
CLOCK_MONOTONIC	O	절대 시간 기준, 시스템 시간 변경 영향 없음 → 안정적 측정
CLOCK_PROCESS_CPUTIME_ID	O	CPU 사용 시간만 측정 (sleep 등 I/O는 제외됨)
CLOCK_THREAD_CPUTIME_ID	O	단일 스레드 CPU 사용 시간 측정

---

6 고급: 프로세스/스레드 CPU 사용 시간 측정

```
struct timespec cpu_start, cpu_end;
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &cpu_start);

// 작업 수행

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &cpu_end);

double cpu_elapsed = (cpu_end.tv_sec - cpu_start.tv_sec) + (cpu_end.tv_nsec - cpu_start.tv_nsec) / 1e9;
printf("Process CPU time used: %.6f seconds\n", cpu_elapsed);
```

→ `CLOCK_PROCESS_CPUTIME_ID` 사용 시 → **실제 CPU에서 수행된 시간만 측정** → sleep(), block 등 제외.

---

7 정리

기능	방법
태스크 전체 경과 시간 측정	`CLOCK_MONOTONIC` 사용
프로세스 CPU 사용 시간 측정	`CLOCK_PROCESS_CPUTIME_ID` 사용
스레드 CPU 사용 시간 측정	`CLOCK_THREAD_CPUTIME_ID` 사용
고해상도 시간 단위	`struct timespec` 사용 → 나노초 단위 지원

---

8 실습 아이디어

- **busy work vs sleep 포함** → `CLOCK_MONOTONIC` / `CLOCK_PROCESS_CPUTIME_ID` 비교 실험
- **다중 스레드 작업 시** → `CLOCK_THREAD_CPUTIME_ID` 로 각 스레드 CPU 사용 시간 측정
- **고성능 모듈의 before/after 구간 측정** → `clock_gettime()` 도입으로 성능 튜닝.

결론

- `clock_gettime()` 은 **리눅스 고해상도 시간 측정의 표준 도구**다.
- `CLOCK_MONOTONIC` 기반 측정 → Benchmarking 에 가장 적합.
- **성능 분석, 실시간 프로세스 튜닝** 등에 널리 활용됨.

다양한 우선순위로 프로세스 실행 실험

목표

- 프로세스를 **서로 다른 우선순위**로 실행한 뒤 CPU 사용률 차이를 관찰.
- **CFS 기반 프로세스** → nice 값 변경 실험.
- **실시간 프로세스** → SCHED_FIFO / SCHED_RR 적용 실험.
- `top`, `ps`, `/proc` 통해 결과 분석.

1️⃣ 실험 구성

준비

- 동일한 **CPU 부하 발생 프로그램(cpu_loop)** 을 사용.
- 여러 프로세스를 서로 다른 우선순위로 실행:
  - nice 값 → CFS 기반
  - SCHED_FIFO / SCHED_RR → 실시간 우선순위

측정 도구

- `top` → 실시간 CPU 점유율 관찰
- `ps` → %CPU 및 정책 확인
- `/proc/<PID>/stat` → 직접 확인 가능

2️⃣ 실험용 프로그램 (cpu_loop_rt.c)

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sched.h>
#include <errno.h>

void busy_work() {
    volatile unsigned long long counter = 0;
    while (1) {
        counter++;
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <mode> [priority|nice_inc]\n", argv[0]);
        printf("mode: normal | fifo | rr\n");
        return 1;
```

```
    }

    if (strcmp(argv[1], "normal") == 0) {
        if (argc >= 3) {
            int n = atoi(argv[2]);
            int ret = nice(n);
            if (ret == -1 && errno != 0) {
                perror("nice");
            } else {
                printf("Set nice increment to %d, new nice: %d\n", n, ret);
            }
        } else {
            printf("Running in normal CFS mode with default nice.\n");
        }
    } else if (strcmp(argv[1], "fifo") == 0 || strcmp(argv[1], "rr") == 0) {
        if (argc < 3) {
            printf("Need priority for real-time mode.\n");
            return 1;
        }

        struct sched_param param;
        param.sched_priority = atoi(argv[2]);
        int policy = (strcmp(argv[1], "fifo") == 0) ? SCHED_FIFO : SCHED_RR;

        if (sched_setscheduler(0, policy, &param) == -1) {
            perror("sched_setscheduler");
            return 1;
        }

        printf("Running with policy %s, priority %d\n",
               (policy == SCHED_FIFO) ? "FIFO" : "RR", param.sched_priority);
    } else {
        printf("Unknown mode %s\n", argv[1]);
        return 1;
    }

    // CPU-intensive work
    busy_work();

    return 0;
}
```

컴파일

```
$ gcc -o cpu_loop_rt cpu_loop_rt.c
```

3️⃣ 실험 실행 방법

3.1 CFS 기반 (nice 값 실험)

터미널 1

```
$ ./cpu_loop_rt normal 0
```

터미널 2

```
$ ./cpu_loop_rt normal 5
```

터미널 3

```
$ ./cpu_loop_rt normal 10
```

→ 기대: nice 값 높을수록 CPU 점유율 ↓

3.2 실시간 정책 실험

주의: 실시간 정책은 **root 권한 필요**:

```
$ sudo ./cpu_loop_rt fifo 20
```

```
$ sudo ./cpu_loop_rt rr 50
```

→ 기대:

- 실시간 프로세스는 **CFS 기반 프로세스를 선점** → 거의 100% CPU 점유.
- FIFO → 타임슬라이스 없이 실행 (Block 또는 yield 시만 스케줄 변경).
- RR → 타임슬라이스 단위로 교대.

4️⃣ 관찰 방법

```
$ top
```

필드 확인:

필드	의미
NI	nice 값 (CFS 프로세스만 표시)
%CPU	프로세스 CPU 점유율
PR	우선순위 (실시간 정책 적용 시 1~99 반영됨, CFS는 기본값 20 기준)

필드	의미
SCHED	`ps -o policy` 또는 `/proc/<PID>/stat` 통해 확인 가능

---

5 기대 결과

CFS 실험 (nice 값 적용)

nice 값	예상 %CPU
0	높음
+5	중간
+10	낮음

→ CFS 내부에서 **vruntime 증가 속도 차이** 발생 → CPU 점유율 감소.

실시간 정책 실험

Policy	예상 동작
SCHED_FIFO	CPU 독점 실행 (Block or yield 없으면 계속 실행)
SCHED_RR	고정 타임슬라이스 교대 실행

→ **SCHED_FIFO 99** 프로세스가 돌면 CFS 프로세스는 CPU 점유 거의 불가.

---

6 실험 확장 아이디어

- **nice 값 전체 sweep**: -20 ~ +19 → 그래프화 가능 (CPU 점유율 vs nice 값)
- 실시간 정책에서:
  - SCHED_FIFO와 SCHED_RR의 **latency / responsiveness 차이 분석**
  - `sched_yield()` 넣어서 FIFO 교대 실험
- **CPU core 바인딩 후 실험** (`sched_setaffinity()` 사용) → 코어별 우선순위 효과 확인 가능.

---

7 정리

정책	우선순위 적용 방법	점유율 변화 효과
CFS	`nice()`	상대적 CPU weight 변화
SCHED_FIFO	`sched_setscheduler()` + priority	고정 우선순위 → 선점성 강함
SCHED_RR	`sched_setscheduler()` + priority	고정 우선순위 + RR 타임슬라이스

→ 시스템 성능 튜닝 시:

- **백그라운드 작업** → 높은 nice 값 사용
- **실시간 요구 작업** → SCHED_FIFO / SCHED_RR 사용 (주의 깊게 설계 필요)