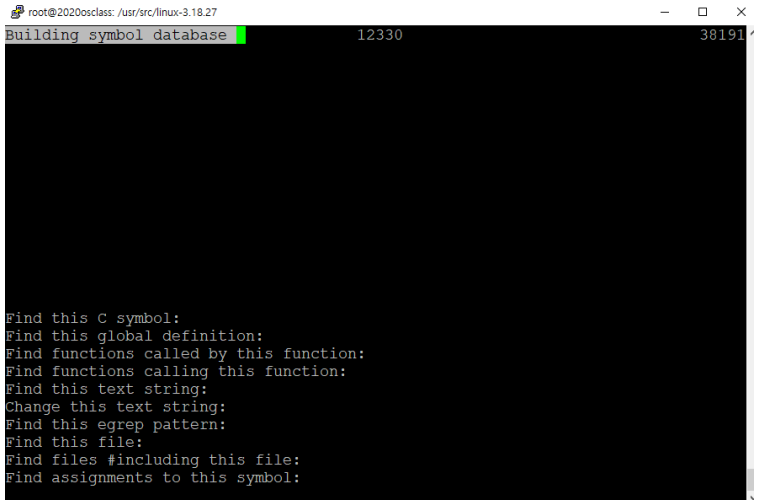
Operating System
[Lab 09]

목차

준비

- **cscope – 대량 소스 분석 툴**
  - Sudo apt-get install cscope
- **cscope 사용을 위한 소스 코드 데이터베이스 생성**
  - cscope –R
  - cscope 종료: Ctrl+D
  - 찾은 후에는 vi에디터로 동작

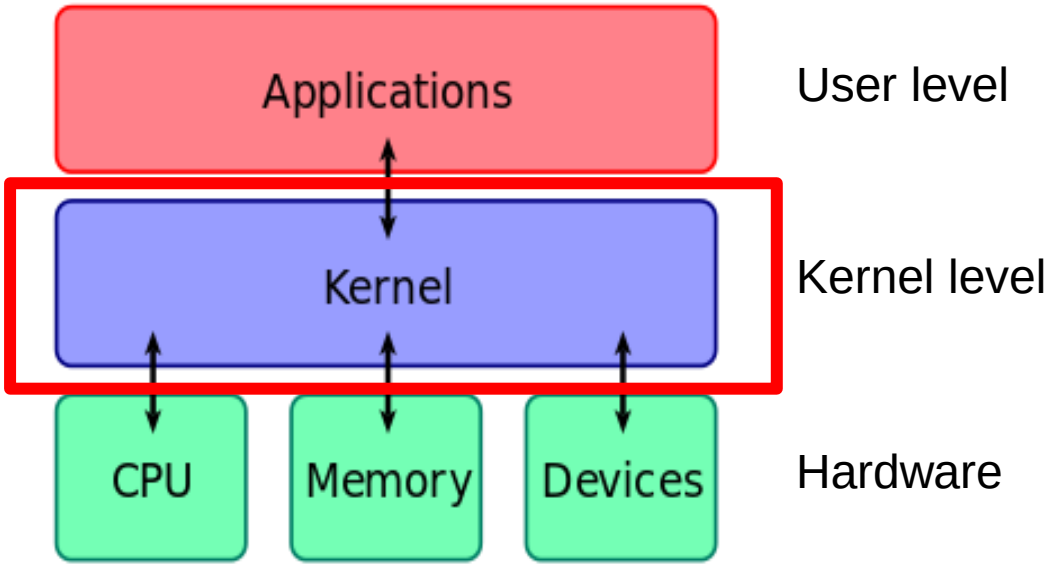



```
root@2020osclass:/usr/src/linux-3.18.27# cscope -R
root@2020osclass:/usr/src/linux-3.18.27# ls
arch          Documentation  Kbuild        mm              samples     usr
block         drivers        Kconfig       modules.builtin scripts     virt
compile.sh    firmware       kernel        modules.order   security    vmlinux
COPYING       fs             lib           Module.symvers  sound       vmlinux.o
CREDITS       include        MAINTAINERS   net             System.map
crypto        init           makedb.sh     README          tags
cscope.out    ipc            Makefile      REPORTING-BUGS  tools
```

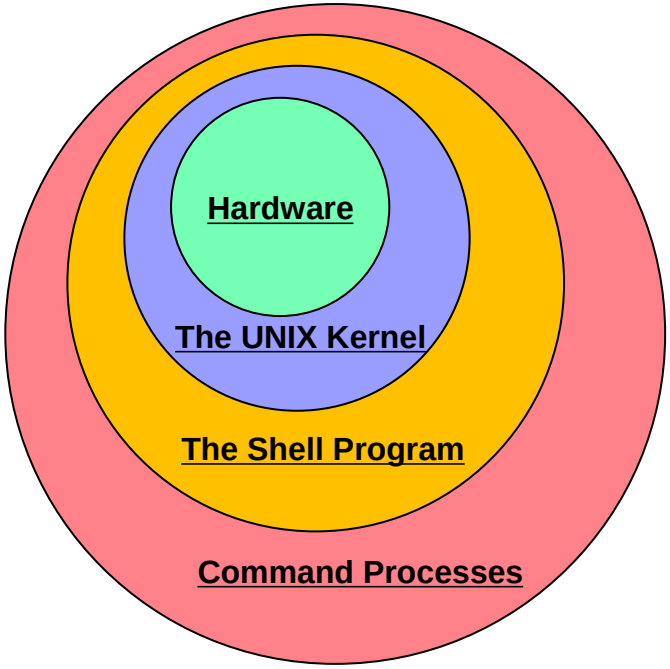
1. 리눅스 커널 소개

리눅스 커널 소개 – 정의

➢ 커널이란?



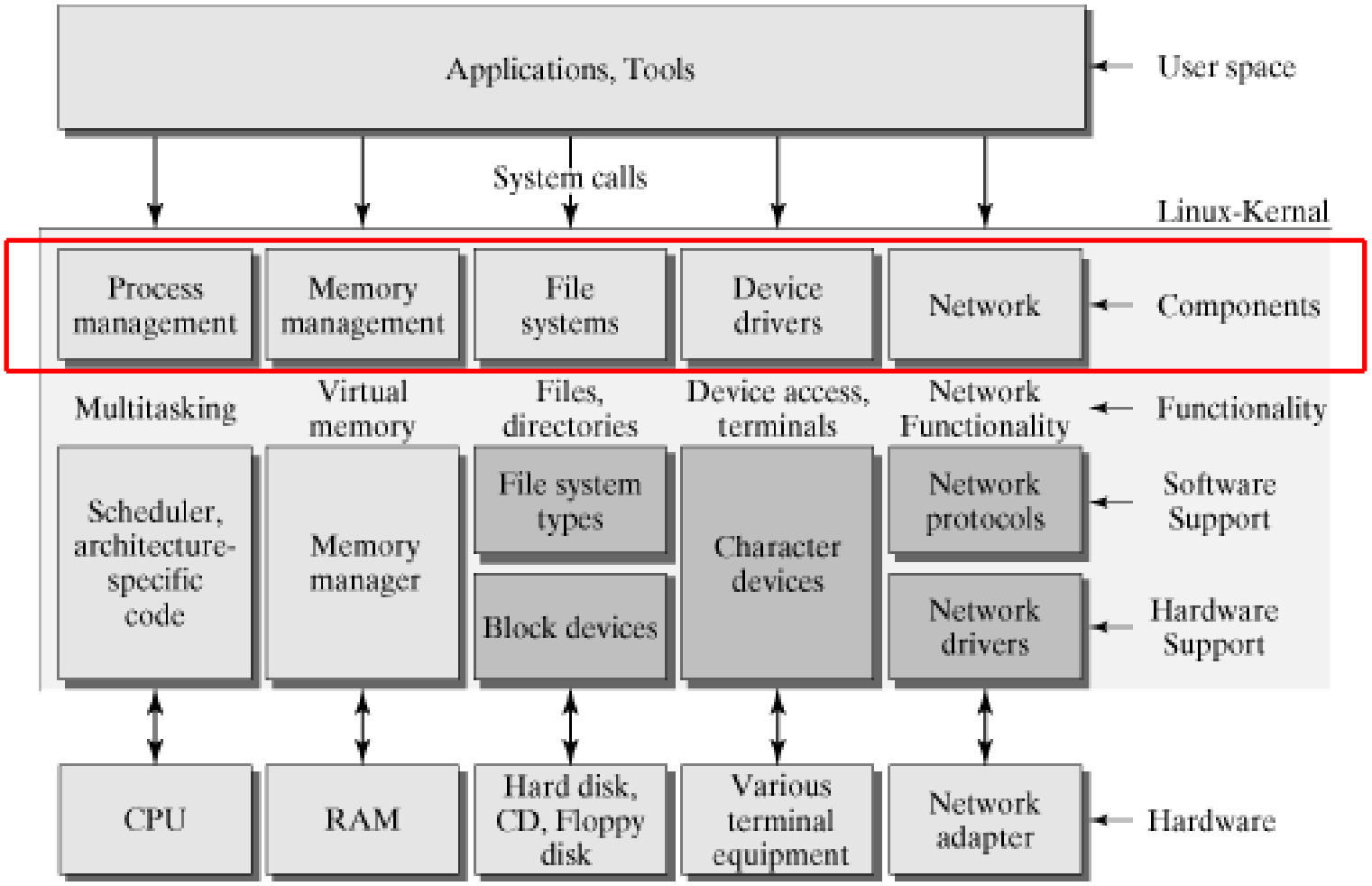
<운영체제 구성요소>



<리눅스 OS의 구성 계층도 >

- 커널이란 운영 체제의 다른 부분 및 응용 프로그램 수행에 필요한 여러 서비스 제공

리눅스 커널 소개 – 구조



<운영체제의 구성요소>

리눅스 커널 소개 – 커널 소스 코드 구성

android	cscope.out.in	Kbuild	mm	scripts
arch	cscope.out.po	Kconfig	modules.builtin	security
block	Documentation	kernel	modules.order	sound
build.log	drivers	lib	Module.symvers	System.map
COPYING	firmware	linaro	net	tools
CREDITS	fs	log.txt	output	usr
crypto	include	MAINTAINERS	README	virt
cscope.files	init	Makefile	REPORTING-BUGS	vmlinux
cscope.out	ipc	make.sh	samples	vmlinux.o

<리눅스 3.17.27의 소스 폴더의 루트 폴더 구조>

- **arch: 아키텍처 종속적인 파일 포함 (cpu 종류에 따라 arm, x86등을 지원)**
  - Interrupt, context switch, 장치 구성, 초기화 코드 등이 위치
- **include: 커널 헤더 파일들이 위치**
  - 아키텍처 독립적인 헤더: include/linux
  - 하드웨어 종속적인 헤더: include/asm-***
- **kernel: 리눅스 커널의 중심적인 디렉토리이며 아키텍처 독립적인 커널 관리 코드들이 위치**
  - 하드웨어 종속적인 커널 관리 코드는 arch/arm/kernel 디렉토리에 존재
  - 스케줄러, 시그널, 시간 관리, fork/exit과 같은 Task 관련 시스템 호출 처리 등과 관련된 코드들이 위치

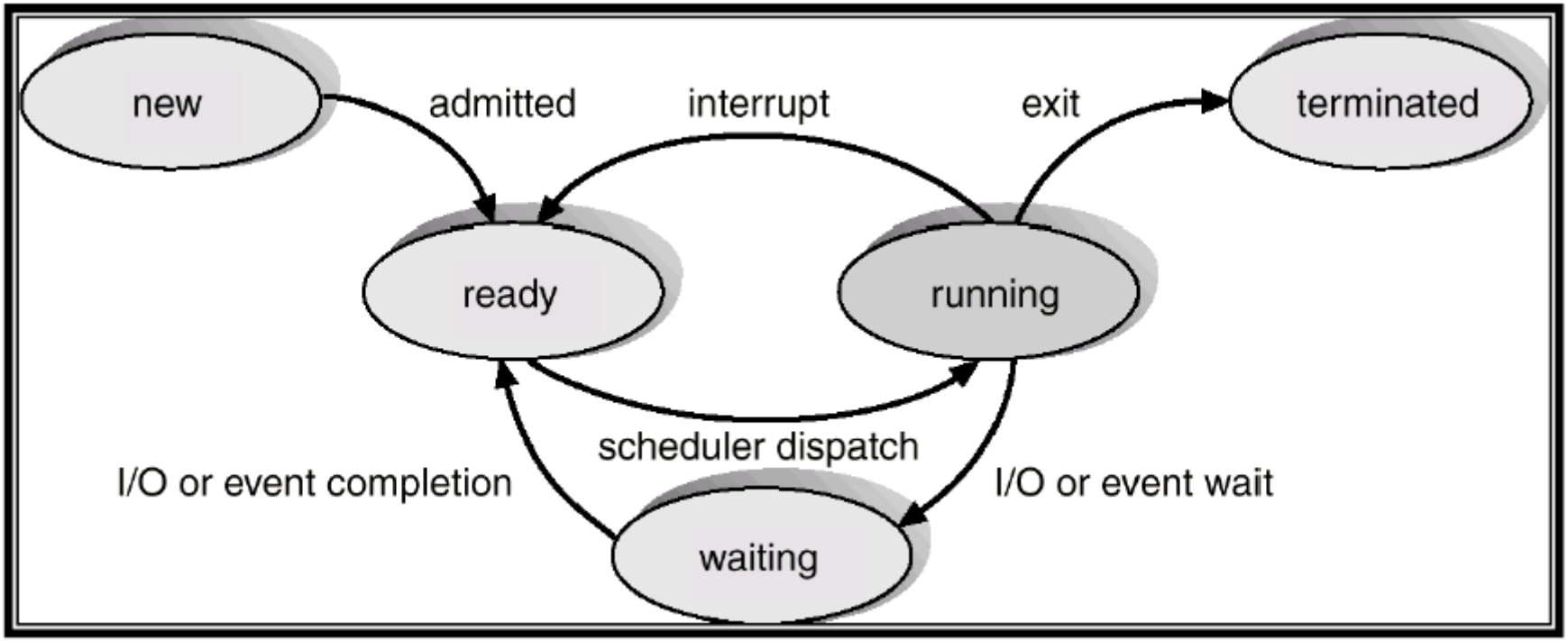
2-1. 리눅스 스케줄러

소개

- 다음에 실행될 프로세스를 선택하는 모듈을 스케줄러라고 함
  - 스케줄러에게 선택된 후 얼마나 실행될지(CPU Time)는 스케줄러에 달림
- 리눅스에는 그룹 스케줄러가 존재하며 이를 스케줄러 클래스라고 함
  - **Linux-3.18.27 코드 기준으로 Stop-task, Deadline, RT, CFS, Idle-task 스케줄러가 존재**
    - Stop-task와 Idle-task는 커널만이 자체적으로 사용 가능
    - Deadline, RT,CFS는 사용자가 선택할 수 있음
  - **CFS 스케줄러는 리눅스 커널 2.6.23 부터 새로 추가된 스케줄러**
  - **리눅스의 스케줄러는 제일 변화가 많은 부분으로 버전마다 차이가 많음**
- 스케줄러 정책이 존재하여 좀 더 세부적으로 스케줄러를 동작 시킴
  - SCHED_DEADLINE: EDF+CBS(Constant Bandwidth Server) 방식.
  - Linux Kernel Deadline Scheduling Policy
  - **RT 스케줄러에 포함된 스케줄러 정책**
    - SCHED_RR: 같은 우선 순위를 가지는 프로세스 사이에선 default 0.1초 단위로 round-robin 방식
    - SCHED_FIFO: FIFO 방식
  - **CFS 스케줄러에 포함된 스케줄러 정책**
    - SCHED_NORMAL: CFS 스케줄러 기본 정책
    - SCHED_BATCH: yield를 회피하여 현재 태스크가 최대한 처리를 할 수 있게 함
    - SCHED_IDLE: CFS 스케줄러에서 가장 낮은 우선순위로 동작하게 함

프로세스 상태 다이어그램

- 프로세스의 상태 표현
  - OS 동작에 따라 프로세스가 실행되고, 프로세스의 상태가 변함
  - Ready와 running은 리눅스 커널에서는 Task_Running 상태로 취급

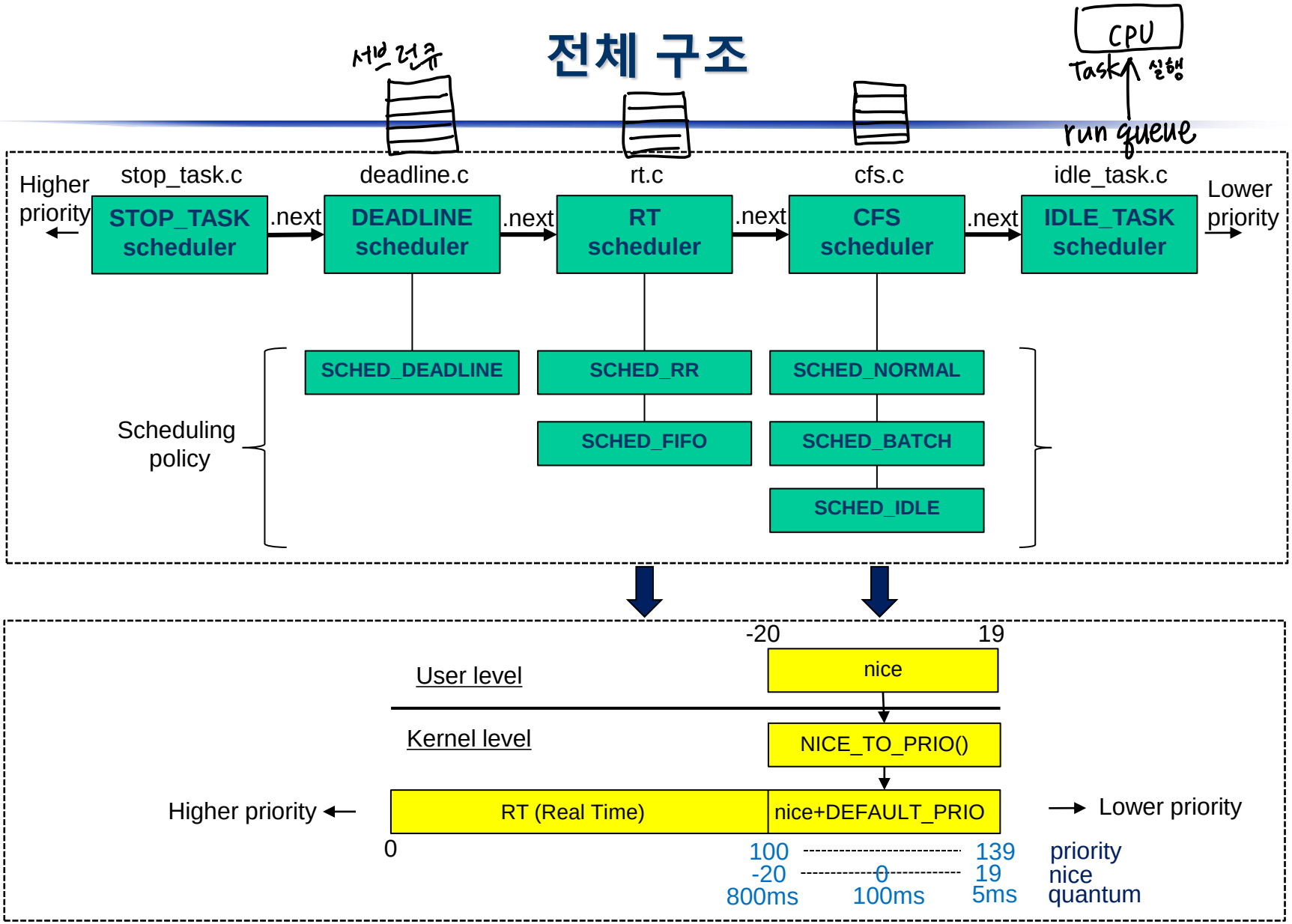


<Diagram of Process State>

schedule() 함수

- **schedule() 함수가 실행이 되면 스케줄링이 이루어짐**
  - **schedule() 함수의 종류**
    - **asmlinkage __visible void __sched schedule(void)**
    - asmlinkage __visible void __sched notrace preempt_schedule(void)
    - asmlinkage __visible void __sched notrace preempt_schedule_context(void)
    - asmlinkage __visible void __sched preempt_schedule_irq(void)
    - static void __cond_resched(void)
  - **schedule() 함수가 실행이 되면 내부적으로 __schedule() 함수가 실행됨**
  - **첫번째 schedule() 함수를 중점적으로 생각하면 됨**
- **schedule() 함수 호출**
  - **시스템 호출(system call) 발생 시**
    - 시스템 콜에서 schedule() 함수 호출
  - **문맥 교환(context switch) 발생 시**
    - 비자발적 문맥 교환이 발생하면 schedule() 함수 실행
      - 타임 슬라이스(Time Slice) 소진 시 문맥 교환 발생
      - 인터럽트(interrupt) 발생 시 문맥 교환 발생
    - 자발적 문맥 교환이 발생하면 schedule() 함수 실행
      - 프로세스가 sleep 또는 exit할 때
      - 프로세스의 상태가 wait 상태일 때

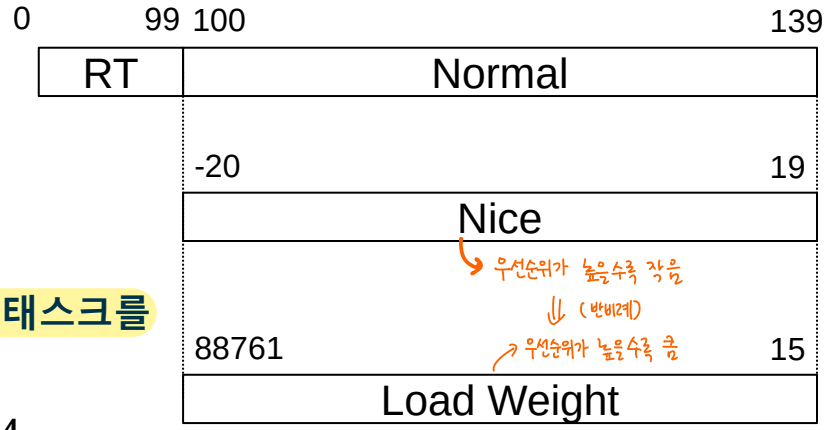
전체 구조


CPU
Task 실행
run queue
Higher priority
stop_task.c
deadline.c
rt.c
cfs.c
idle_task.c
Lower priority
STOP_TASK scheduler
.next
DEADLINE scheduler
.next
RT scheduler
.next
CFS scheduler
.next
IDLE_TASK scheduler
SCHED_DEADLINE
SCHED_RR
SCHED_NORMAL
SCHED_FIFO
SCHED_BATCH
SCHED_IDLE
Scheduling policy
-20
19
User level
nice
Kernel level
NICE_TO_PRIO()
Higher priority
RT (Real Time)
nice+DEFAULT_PRIO
Lower priority
0
100
139
priority
-20
0
19
nice
800ms
100ms
5ms
quantum

CFS Scheduler

- **Completely Fair Scheduler**
  - 목표: 태스크에게 CPU시간을 공정하게 분배하는 것
  - 공정하게 분배 ≠ 시간을 균등하게 분배
  - 공정한 분배를 위해 우선순위를 기반으로 결정되는 Load Weight라는 개념 사용
- **Load Weight**
  - 우선순위와 매칭되는 Nice값으로 결정

$$load\ weight = \frac{1024}{1.25^{nice}}$$



- **Virtual Runtime**
  - CFS 스케줄러는 Virtual Runtime이 가장 작은 태스크를 선택하여 실행함

$$virtual\ runtime = real\ runtime\ * \frac{1024}{load\ weight}$$

  - 누적되는 값으로, load weight가 크면 천천히 값이 증가하고, 작다면 빠르게 증가

CFS Scheduler

- virtual runtime 비교
  - Load weight 비율이 1 : 2 : 3인 3개의 태스크
  - Virtual runtime의 비율은 load weight 비율이 역으로 반영된 1 : 0.5 : 0.33

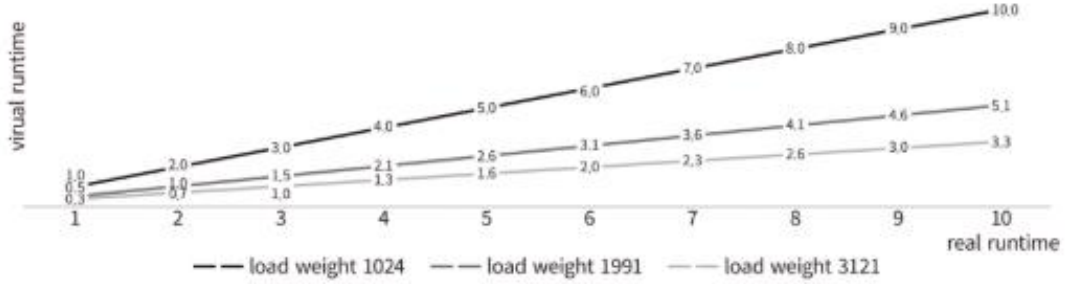


그림 6-5 load weight의 크기에 따라 달라지는 virtual runtime의 추이

  - 동일한 런타임을 소모해도 virtual runtime이 달라짐

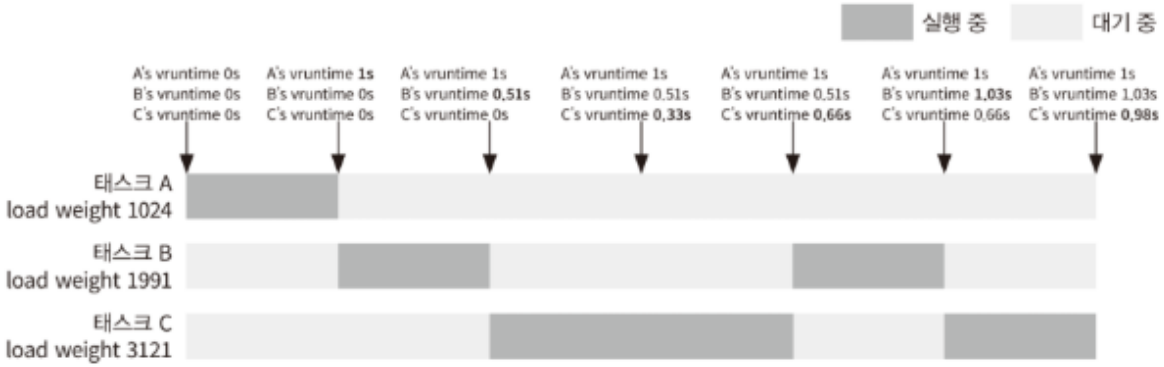


그림 6-6 서로 다른 load weight에 따라 달라지는 태스크의 실행시간

그림 출처: 코드로 알아보는 ARM 리눅스 커널

CPU의 런큐(Run Queue)와 서브 런큐

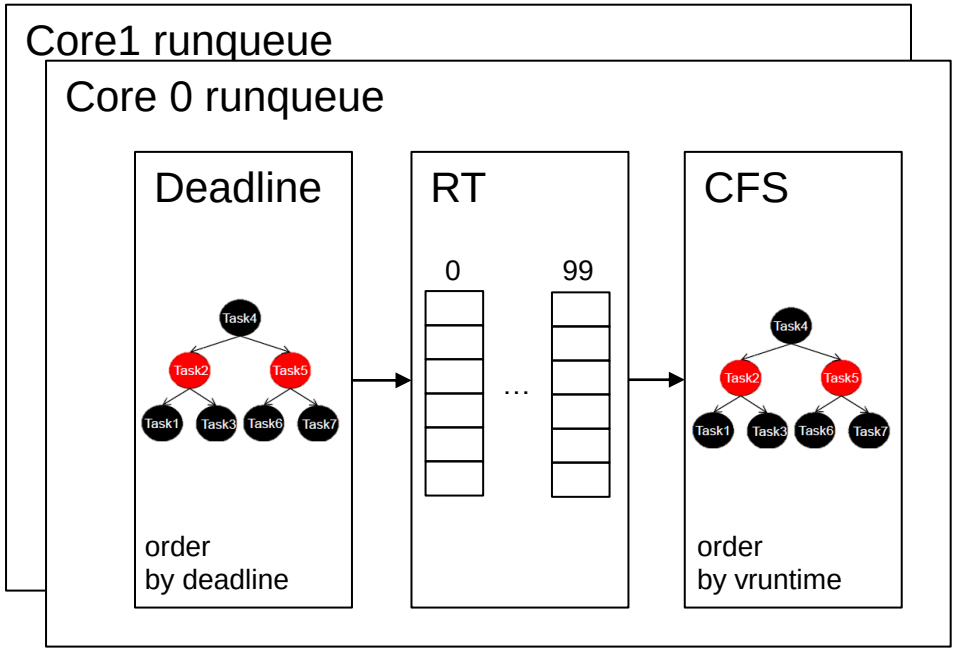
- **CPU의 런큐란 전체 스케줄러의 메타 정보와 서브런큐를 담는 자료구조**
  - **각 CPU마다 런큐(rq)를 가지고 있으며 태스크는 한 CPU의 런큐에만 존재**
  - **struct rq 자료구조를 사용**
    - cfs_rq, rt_rq, dl_rq: 스케줄러의 런큐이며 서브 런큐라고도 불림
    - *curr: 현재 cpu에서 실행되고 있는 태스크의 task_struct 주소

위치: linux-3.18.27/kernel/sched/sched.h

```
518  struct rq {
526          unsigned int nr_running;
548          struct cfs_rq cfs;
549          struct rt_rq rt;
550          struct dl_rq dl;
                  :
567          struct task_struct *curr, *idle, *stop;
568          unsigned long next_balance;
569          struct mm_struct *prev_mm;
570
571          u64 clock;
572          u64 clock_task;
```

task 개수

<struct rq>



CPU의 런큐(Run Queue)와 서브 런큐

- **CPU의 런큐란 전체 스케줄러의 메타 정보와 서브런큐를 담는 자료구조**
  - **태스크의 상태(state)가 변경되면 자신에게 할당된 스케줄러(sched_class 값)를 확인하고 해당 스케줄러와 관련된 서브 런큐에서 enqueue/dequeue를 진행**
    - 태스크가 어떤 상태(state)에서 RUNNING 상태로 전이되면 서브 런큐로 enqueue가 됨
    - 태스크가 RUNNING 상태에서 어떤 상태(state)로 전이되면 서브 런큐에서 dequeue가 됨

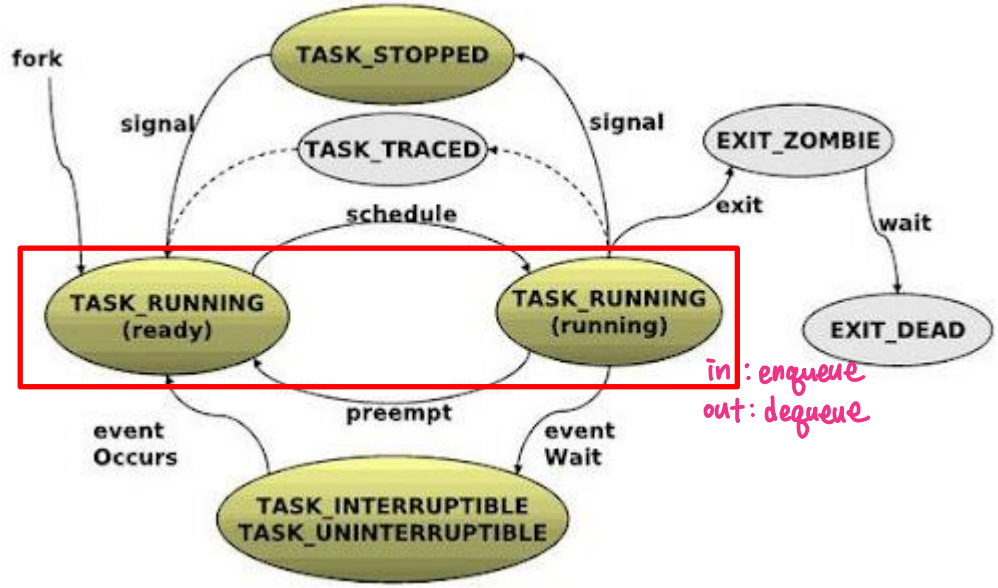


그림 출처: https://jinkyu.tistory.com/87

sched_entity 자료구조

- task_struct 자료구조에서 스케줄링을 위한 멤버변수들을 모아 놓은 자료구조
  - 리눅스 스케줄러의 sched_entity 종류
    - cfs 스케줄러: sched_entity
    - rt 스케줄러: sched_rt_entity
    - Deadline 스케줄러: sched_dl_entity
  - 서브 런큐는 원소로 sched_entity 자료구조를 가짐
    - 태스크가 RUNNING 상태로 전이되어, 서브 런큐에 진입을 하면 자신이 가진 sched_entity 자료구조를 서브런큐에 enqueue함
    - 태스크가 RUNNING 상태를 벗어나면, 서브 런큐에서 벗어나면서 enqueue되어있던 자료구조를 서브런큐에서 dequeue를 함
  - 스케줄러는 서브 런큐에 들어있는 sched_entity들을 비교하여 다음에 실행할 태스크를 선출함
    - 해당 과정은 pick_next_task 함수를 통해 이루어짐

위치: linux-3.18.27/include/linux/sched.h

```
1236  struct task_struct {
1237          volatile long state;    /* -1 unrunnat
1252          int on_rq;
1256          const struct sched_class *sched_class;
1257          struct sched_entity se;
1258          struct sched_rt_entity rt;
1259  #ifdef CONFIG_CGROUP_SCHED
1260          struct task_group *sched_task_group;
1261  #endif
1262          struct sched_dl_entity dl;
```

스케줄러들을 할당

<struct task_struct>

위치: linux-3.18.27/include/linux/sched.h

```
1155  struct sched_rt_entity {
1156          struct list_head run_list;
1157          unsigned long timeout;
1158          unsigned long watchdog_stamp;
1164          /* rq on which this entity is (to be) queued: */
1165          struct rt_rq            *rt_rq;
1166          /* rq "owned" by this entity/group: */
1167          struct rt_rq            *my_q;
1168  #endif
1169  };
```

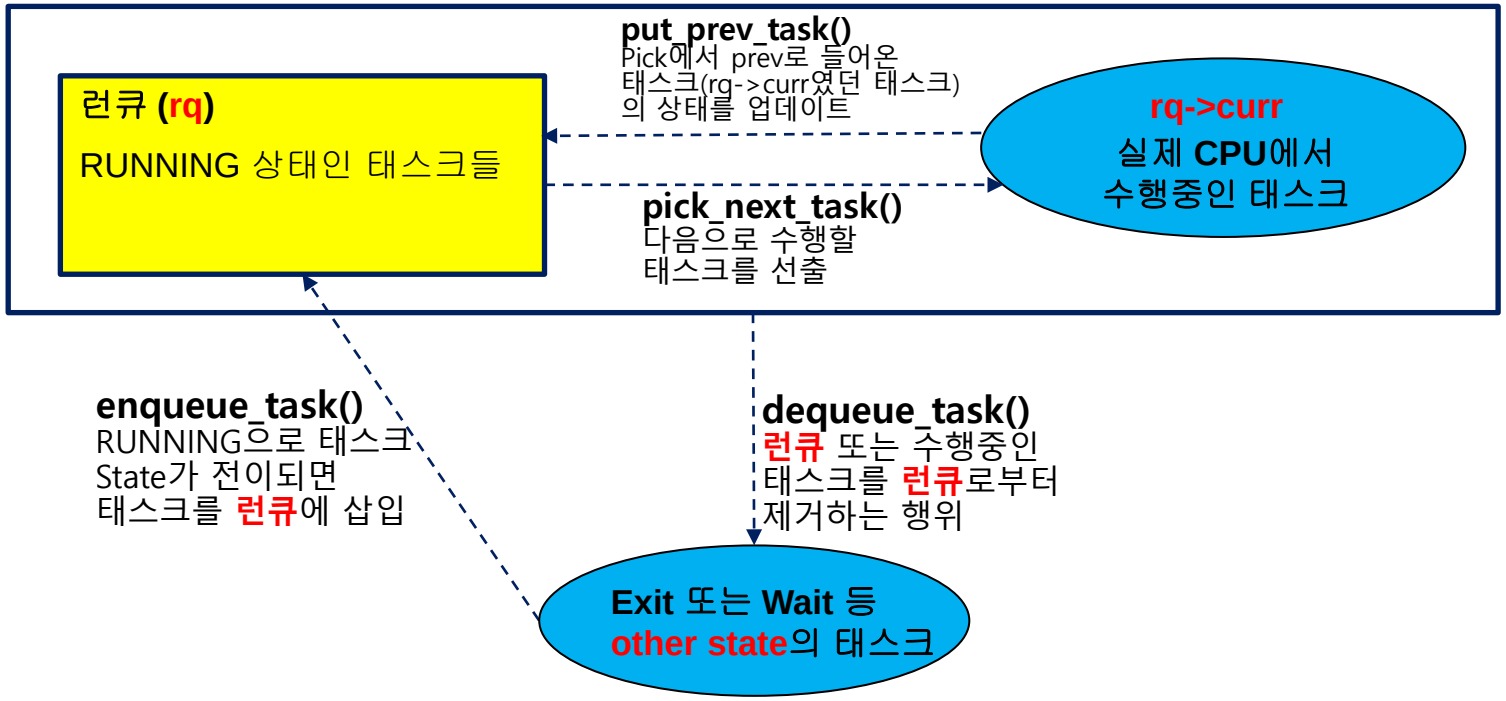
각 스케줄러마다 entity 존재

<struct sched_rt_entity>

2-2. 스케줄링 클래스

주요 스케줄링 과정

- **sched_class를 이용한 태스크 스케줄링**
  - **put_prev_task 및 pick_next_task는 세부 스케줄러의 구현방식에 따라 동작이 달라짐**
    - cfs의 경우 put_prev_task 및 pick_next_task에서는 서브런큐의 enqueue/dequeue 동작이 실제로 발생함
    - Deadline 및 rt의 경우 pushable 큐라고 하는 큐로 enqueue/dequeue만 일어날 뿐 실제 서브런큐는 건드리지 않음



< 태스크 스케줄링 >

sched_class 자료구조

➢ sched_class 자료구조

- 리눅스의 다양한 스케줄러 정책의 동작을 지원하는 인터페이스
- 모든 스케줄러는 sched_class를 자료형으로 가지는 변수를 가짐
- 전체 스케줄러 동작에서 sched_class wrapper 함수를 부르면 내부적으로는 세부 스케줄러의 sched_class의 함수를 부르게 됨

위치: linux-3.18.27/kernel/sched/sched.h

```
1088  struct sched_class {
1089          const struct sched_class *next;
1090
1091          void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
1092          void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
1106          struct task_struct * (*pick_next_task) (struct rq *rq,
1107                                                  struct task_struct *prev);
1108          void (*put_prev_task) (struct rq *rq, struct task_struct *p);
1109
1125          void (*set_curr_task) (struct rq *rq);
1126          void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
1138          void (*update_curr) (struct rq *rq);
1143  };
```

<struct sched_class>

위치: linux-3.18.27/kernel/sched/fair.c

```
7937  const struct sched_class fair_sched_class {
7938          .next                   = &idle_sched_class,
7939          .enqueue_task           = enqueue_task_fair,
7940          .dequeue_task           = dequeue_task_fair,
7941          .yield_task             = yield_task_fair,
7942          .yield_to_task          = yield_to_task_fair,
7943
7944          .check_preempt_curr     = check_preempt_wakeup,
```

<fair_sched_class 변수>

sched_class 주요 함수

- **sched_class의 주요 함수 세부 설명**
  - **상태 전이한 (other state ←→ RUNNABLE) 태스크를 서브런큐에 '삽입 또는 제거'하는 함수**
    - enqueue_task: 태스크를 서브런큐에 삽입(enqueue)하는 함수
    - dequeue_task: 서브런큐 안에 있는 태스크를 제거(dequeue)하는 함수
  - **태스크의 상태를 변경해주는 함수**
    - pick_next_task: 다음에 실행할 태스크를 서브런큐에서 선택하여 반환함
      - ✓cfs의 경우 서브런큐에서 dequeue도 진행
    - put_prev_task: pick에서 prev로 들어온 태스크(rq->curr였던 태스크)의 상태를 업데이트
      - ✓cfs의 경우 서브런큐에 enqueue도 진행
    - set_curr_task: cpu의 런큐나 서브런큐의 curr를 설정할 때 쓰임
    - task_tick: 주기적으로 task_tick이 호출이 되는데, 이 때 스케줄러에 따라 현재 수행중인 태스크의 상태를 업데이트하고 스케줄링 할지를 판단하기도 함
      - ✓예: 타임 슬라이스를 모두 소진할 경우 다른 태스크로 선점이 될 수 있음
    - update_curr: 태스크의 상태를 업데이트하는 함수

enqueue_task

➢ enqueue_task

- ‘생성’되거나 ‘대기’ 상태의 태스크가 ‘실행가능’ 상태로 전환되면 스케줄러의 서브런큐에 삽입하는 함수
- 호출된 함수는 각 스케줄러의 sched_class를 참조하여 실제 함수를 수행
- 서브런큐에 삽입된 태스크는 다음에 실행될 태스크로 선택될 수 있게 됨

```
static void enqueue_task(struct rq *rq, struct task_struct *p, int flags)
{
        update_rq_clock(rq);
        sched_info_queued(rq, p);
        p->sched_class->enqueue_task(rq, p, flags);
}
```

or

```
static void
enqueue_task_stop(struct rq *rq, struct task_struct *p, int flags)
{
        add_nr_running(rq, 1);
```

or

```
static void
enqueue_task_fair(struct rq *rq, struct task_struct *p, int flags)
{
        struct cfs_rq *cfs_rq;
        struct sched_entity *se = &p->se;
```

or

```
static void
enqueue_task_rt(struct rq *rq, struct task_struct *p, int flags)
{
        struct sched_rt_entity *rt_se = &p->rt;
```

or

```
static void enqueue_task_dl(struct rq *rq, struct task_struct *p, int flags)
{
        struct task_struct *pi_task = rt_mutex_get_top_task(p);
        struct sched_dl_entity *pi_se = &p->dl;
                :
                :
```

<각 태스크가 속한 클래스별 함수로 동작>

dequeue_task

➢ dequeue_task

- 현재 실행 중인 태스크가 '종료' 또는 '실행 가능 않은 상태'로 전환되면 스케줄러의 서브 런큐에서 태스크를 삭제하는 함수

```
static void dequeue_task(struct rq *rq, struct task_struct *p, int flags)
{
        update_rq_clock(rq);
        sched_info_dequeued(rq, p);
        p->sched_class->dequeue_task(rq, p, flags);
}
```

```
static void
dequeue_task_stop(struct rq *rq, struct task_struct *p, int flags)
{
        sub_nr_running(rq, 1);
}
```

or

```
static void __dequeue_task_dl(struct rq *rq, struct task_struct *p, int flags)
{
        dequeue_dl_entity(&p->dl);
        dequeue_pushable_dl_task(rq, p);
}
```

or

```
static void dequeue_task_rt(struct rq *rq, struct task_struct *p, int flags)
{
        struct sched_rt_entity *rt_se = &p->rt;
                :
                :
```

or

```
static void dequeue_task_fair(struct rq *rq, struct task_struct *p, int flags)
{
        struct cfs_rq *cfs_rq;
        struct sched_entity *se = &p->se;
        int task_sleep = flags & DEQUEUE_SLEEP;
                :
                :
```

<각 태스크가 속한 클래스별 함수로 동작>

pick_next_task

- pick_next_task
  - 현재 실행 중인 태스크의 클래스 서브 런큐를 참조하여 다음 수행할 태스크를 선택
  - 태스크를 선택하는 방식은 해당 클래스의 스케줄러 정책에 따라 다름
  - 만약 실행 가능한 태스크가 없을 경우 다음 우선 순위의 클래스의 서브 런큐 참조

```
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev)
{
```

cpu의 런큐 안에 있는 태스크가 모두 cfs 일 때 실행되는 코드이므로 생략

```
        :
        :
again:
        for_each_class(class) {
                p = class->pick_next_task(rq, prev);
                if (p) {
                        if (unlikely(p == RETRY_TASK))
                                goto again;
                        return p;
                }
        }

        BUG(); /* the idle class will always have a ru
}
```

or

```
static struct task_struct *
pick_next_task_stop(struct rq *rq, struct task_struct *prev)
{
        struct task_struct *stop = rq->stop;
```

or

```
struct task_struct *pick_next_task_dl(struct rq *rq, struct task_struct *prev)
{
        struct sched_dl_entity *dl_se;
        struct task_struct *p;
        struct dl_rq *dl_rq;
```

or

```
static struct task_struct *
pick_next_task_rt(struct rq *rq, struct task_struct *prev)
{
        struct task_struct *p;
        struct rt_rq *rt_rq = &rq->rt;
```

```
static struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev)
{
        struct cfs_rq *cfs_rq = &rq->cfs;
        struct sched_entity *se;
```

<각 태스크가 속한 클래스별 함수로 동작>

set_curr_task

- set_curr_task
  - 런큐에 있는 태스크를 주어진 CPU에 실제 실행시키는 동작의 함수
  - 서브 런큐(set_curr_task_fair 등) 에서는, 태스크가 스케줄링 클래스를 바꾸거나 태스크 그룹을 바꿀 때 호출

run queue에 있는 task를 CPU에 실행시킨다

```
void set_curr_task(int cpu, struct task_struct *p)
{
        cpu_curr(cpu) = p;
}
```

```
#define cpu_curr(cpu)          (cpu_rq(cpu)->curr)
```

<각 태스크가 속한 클래스별 함수로 동작>

task_tick

task_tick

- Timer Tick 호출 함수
- 추가적인 동작이 있을 수 있음(할당된 타임 슬라이스 값을 소진했는지 여부 등)
- 할당된 시간을 모두 소진했을 경우 해당 태스크에 'TIF_NEED_RESCHED'이라는 플래그를 SET하여 스케줄링이 가능하도록 설정하며 이는 resched_curr 함수를 통해 이루어짐

```
void scheduler_tick(void)
{
        int cpu = smp_processor_id();
        struct rq *rq = cpu_rq(cpu);
        struct task_struct *curr = rq->curr;
                        :
                        :
        update_rq_clock(rq);
        curr->sched_class->task_tick(rq, curr, 0);
        update_cpu_load_active(rq);
```

or

```
static void task_tick_dl(struct rq *rq, struct task_struct *p, int queued)
{
        update_curr_dl(rq);
```

or

```
static void task_tick_rt(struct rq *rq, struct task_struct *p, int queued)
{
        struct sched_rt_entity *rt_se = &p->rt;
                                :
                                :
        resched_curr(rq);
```

```
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
{
        struct cfs_rq *cfs_rq;
        struct sched_entity *se = &curr->se;
                        if (queued) {
                                resched_curr(rq_of(cfs_rq));
                                return;
                                :
                                :
```

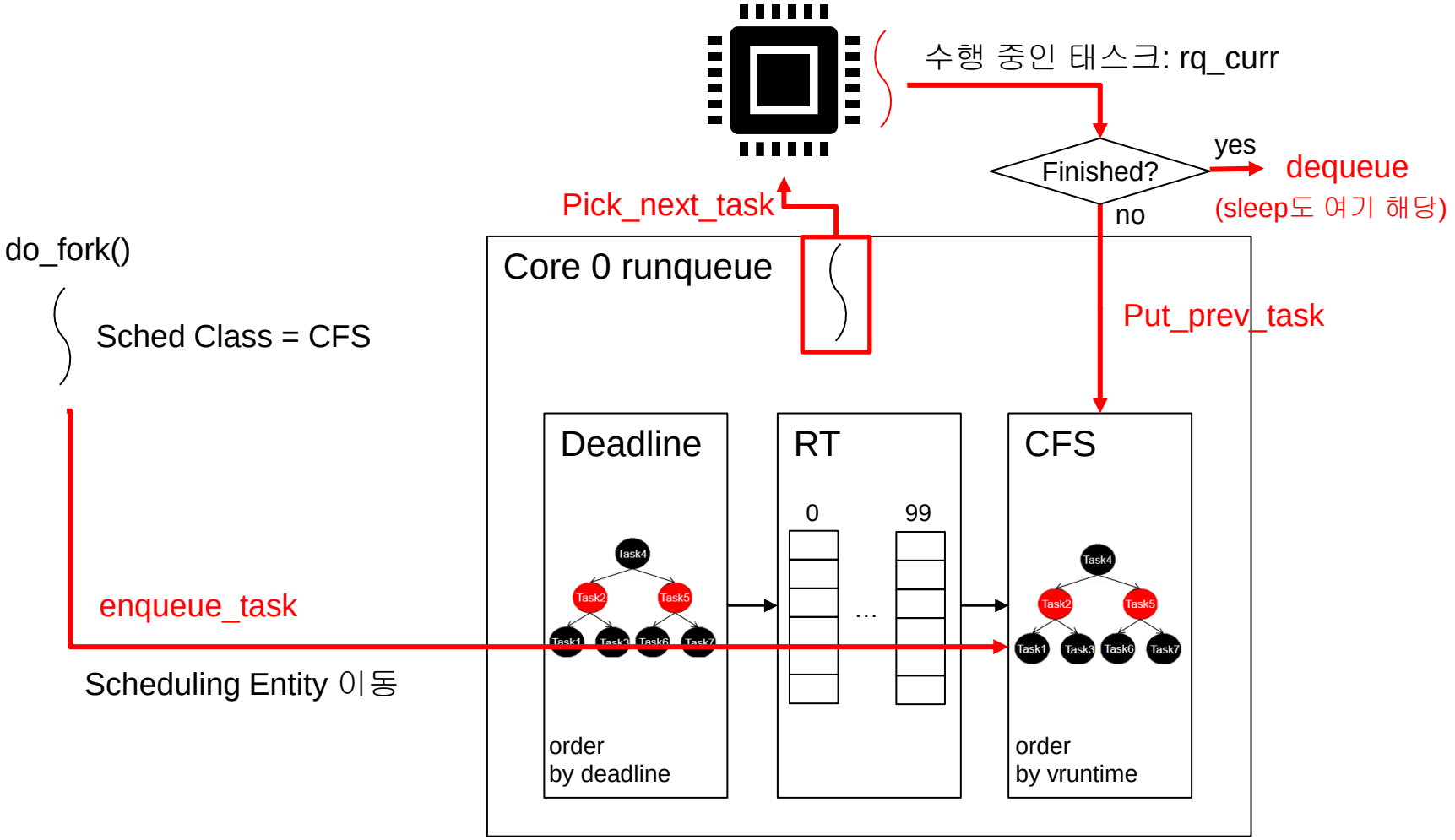
<각 태스크가 속한 클래스별 함수로 동작>

실습 1

- **Task가 Deadline 스케줄러를 따르고 있다고 가정한 상태에서 __schedule() 함수의 흐름도를 그림**
  - **__schedule 함수의 pick_next_task부터 __schedule 끝까지의 흐름**
  - **do_fork부터 p->sched_class->enqueue_task가 호출되기까지의 흐름**
  - **__schedule부터 p->sched_class->dequeue_task가 호출되기까지의 흐름**

변경사항*: 다음주차 실습(5월 11일) 과제와 같이 제출

리눅스 스케줄링 정리


수행 중인 태스크: rq_curr
Finished?
yes
dequeue
(sleep도 여기 해당)
no
Pick_next_task
Put_prev_task
do_fork()
Sched Class = CFS
Core 0 runqueue
enqueue_task
Scheduling Entity 이동
Deadline
RT
CFS
0
99
Task4
Task2
Task5
Task1
Task3
Task6
Task7
order
by deadline
order
by vruntime

리눅스 스케줄링 정리

<include/linux/sched.h>

```
struct sched_entity {
        struct load_weight load;
        …
        u64 vruntime;
        …
};

struct task_struct {
        …
        const struct sched_class *sched_class;
        struct sched_entity se;   /* cfs */
        struct sched_rt_entity rt;
        …
};
…
```

<kernel/sched/sched.h>

```
…
struct cfs_rq {
        …
        struct load_weight load;
        unsigned int nr_running, h_nr_running;
        …
        struct sched_entity *curr, *next, *lask, *skip;
        …
};
…
struct rq {
        …
        struct cfs_rq cfs;
        struct rt_rq rt;
        struct dl_rq dl;
        …
};
…
```