

설계와 구현을 통한 임베디드 OS의 이해와 응용 ⑨

우선순위 기반 스케줄링 작성

지난 호에 우리는 태스크 루틴과 하드웨어 인터럽트 처리 루틴 간, 또한 태스크 루틴과 태스크 루틴 간에 공유 영역에 대한 경쟁 상태가 발생하는 상황을 보았다. 이번 호에는 이러한 문제들에 대한 일반적인 해결책을 알아보고 필요한 루틴을 추가해 보자. 또 라운드 로빈 방식과 우선순위 방식의 스케줄링을 구현하기 위해 필요한 일반 큐와 우선순위 큐를 구현해 보기로 하자.

글 : 서민우 / 새롬전자

mwseo@e-serome.co.kr / www.e-serome.co.kr

지난 호에 우리는 라운드 로빈 방식과 우선순위 방식의 스케줄링을 구현하기 위해 필요한 일반 큐와 우선순위 큐를 구현해 보았다. 이번 호에서는 이 두 가지 큐를 이용하여 라운드 로빈 방식과 우선순위 방식의 스케줄링을 구현해보기로 하자.

먼저 라운드 로빈 방식의 스케줄링을 구현해보기로 하자.

다음은 task.c 파일의 내용이다.

```
task.c
#include <task.h>
#include <null.h>
#include <prior-queue.h>

process_state process[16];
process_state * current;
process_state * prev, * next;
unsigned int pid = 0;

priorQ pQ[2];
priorQ * active;
```

```
priorQ * expired;

void init_task()
{
    current = &process[0];
    next = current;

    process[0].context[0+2] = 0;
    process[0].context[13+2] = 0x408000;
    process[0].context[1] = 0x400000;
    process[0].context[0] = 0x5f;
    process[0].pid = 0;
    process[0].time_remain = 10;
    process[0].time_slice = 10;
    process[0].need_resched = 0;
    process[0].prior = 0;
    process[0].next = NULL;

    (int i;
```



서민우(mwseo@e-serome.co.kr)

* 연재순서

설계와 구현을 통한 임베디드 OS의 이해와 응용

- 1회 Overview
- 2회 개발환경 구성 및 부트로더 작성
- 3회 cuteOS의 시작
- 4회 MMU와 캐시 설정
- 5회 주요 루틴 작성하기
- 6회 하드웨어 인터럽트 처리 및 태스크 추가 루틴
- 7회 문맥 전환 및 스케줄링 루틴 작성
- 8회 동기화 문제의 해결과 우선 순위 큐
- 9회 우선 순위 기반 스케줄링 작성**
- 10회 IPC의 설계 및 구현

리눅스 커널, RTOS 커널의 구조에 관심을 가지고 연구하고 있으며, 리눅스나 RTOS 디바이스 드라이버 개발을 주업으로 하고 있다.

본 기사를 통하여 MMU 기능이 있는 cuteOS라는 마이크로 커널을 구현하였다. 과거에 32비트 마이크로 프로세서를 설계하고 VHDL을 이용하여 구현한 경험이 있다. 현재 (주)새롭전자에서 영상칩을 제어하기 위해 펌웨어, 리눅스 디바이스 드라이버, USB WDM 디바이스 드라이버 구현과 관련된 일들을 하고 있다.

```
for(i=1;i<16;i++) {
    process[i].context[0+2] = i;
    process[i].context[13+2] = 0x408000;
    process[i].context[1] = 0x400000;
    process[i].context[0] = 0x5f;
    process[i].pid = i;
    process[i].time_remain = i;
    process[i].time_slice = i;
    process[i].need_resched = 0;
    process[i].prior = 0;
    process[i].next = NULL;
}

initpriorQ(&pQ[0]);
initpriorQ(&pQ[1]);

active = &pQ[0];
expired = &pQ[1];

for(int i;
for(i=1;i<16;i++) enpriorQ(active, &process[i]);
}
```

```
}
...
void schedule()
{
    current = depriorQ(active);
    if(current == NULL) {
        priorQ * tmp;
        tmp = active;
        active = expired;
        expired = tmp;
        current = depriorQ(active);
    }

    next = current;

    flush_cache_tlb();
    if(current->pid==0) mmu_map_l2pt(MASTERL1PT
+4, TASK0L2PTBASE);
    else if(current->pid==1) mmu_map_l2pt
(MASTERL1PT+4, TASK1L2PTBASE);
    else mmu_map_l2pt(MASTERL1PT+4, TASKL
2PTBASE(current->pid));
}
```

task.c 파일에서는 먼저 task.h, null.h, prior-queue.h 파일을 include 시켜주었다. task.h 파일은 process_state 타입의 변수를 참조하기 위해 사용하였다. null.h 파일에는 NULL 매크로가 정의되어 있다. prior-queue.h 파일은 priorQ 타입의 변수를 참조하기 위해서 사용하였다.

pQ 배열 변수는 두 개의 우선순위 큐를 관리하기 위하여 선언하였다. pQ[0], pQ[1] 우선순위 큐는 모두 ready queue이다. active 포인터 변수는 time slice를 가지고 있는 태스크들이 대기하고 있는 우선순위 큐를 가리킨다.

active 포인터 변수는 pQ[0] 또는 pQ[1] 변수를 가리킬 수 있다. expired 포인터 변수는 time slice를 다 쓴 후 새롭게 time slice를 할당 받은 태스크들이 대기하고 있는 우선순위 큐를 가리킨다. expired 포인터 변수도 pQ[0] 또는 pQ[1] 변수를 가리킬 수 있다.

init_task 함수에서는 각 태스크의 pid, time_remain, time_slice, need_resched, prior, next 변수를 초기화하고 있다. task0의 경우는 time_slice의 값으로 10을 할당하였고, 나머지 task의 경우는 pid 값에 따라 time slice를 다르게 주었다. 즉, pid 값이 적을수록 time slice도 적게 주었다. 그리고 라운드 로빈 방식의 스케줄링을 확인하기 위하여 모든 태스크의 prior 값을 0으로 초기화하여 우선순위를 같게 하였다.

다음은 pQ[0], pQ[1] 우선순위 큐를 초기화하고, active, expired 포인터 변수는 각각 pQ[0], pQ[1] 우선순위 큐를 가리키도록 초기화한다. 그리고 1~15번 태스크를 active 포인터 변수가 가리키는 pQ[0] 우선순위 큐에 넣는다.

schedule 함수에서는 다음에 수행할 태스크를 active 포인터 변수가 가리키는 우선순위 큐에서 꺼낸다. 만약에 active 포인터 변수가 가리키는 우선순위 큐에 태스크가 남아 있지 않다면 active 포인터 변수는 expired 포인터 변수가 현재 가리키고 있는 우선순위 큐를 가리키게 하고, expired 포인터 변수는 바로 전에 active 포인터 변수가 가리키던 우선순위 큐를 가리키게 한다. 그리고 pid 값에 따라 가상 주소를 바꾼다.

다음은 irqhandler.c 파일의 내용이다.

```
irqhandler.c
#include <timer.h>
#include <task.h>
#include <prior-queue.h>

extern process_state * current;
extern priorQ * expired;
unsigned int jiffies = 0;

void eventsIRQHandler()
{
    unsigned int rintoffset;
    unsigned int rintpnd;

    rintoffset = rINTOFFSET;
    rintpnd = rINTPND;

    jiffies ++;
    current->time_remain --;
    if(current->time_remain <= 0) {
        current->time_remain = current-
>time_slice;

        enpriorQ(expired, current);
        current->need_resched = 1;
    }

    rSRCPND |= rintpnd;
    rINTPND |= rintpnd;

    if(current->need_resched == 1) {
        current->need_resched = 0;
        schedule();
    }
}
```

이상에서 `irqhandler.c` 파일의 내용을 살펴보았다.

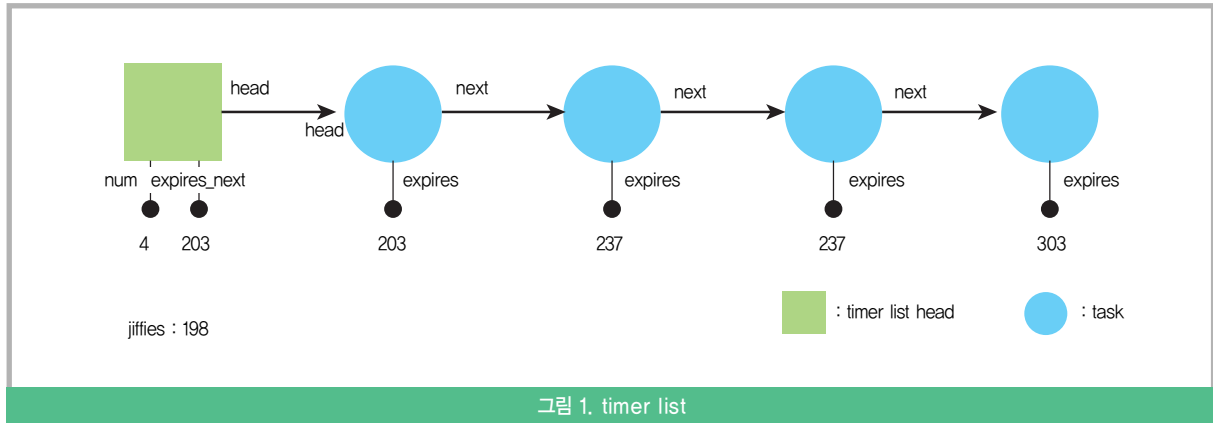
```
task0.c
#include <system.h>

int main(unsigned int arg)
{
    for(;;) {
        int i;
        for(i=0;i<0x10000;i++);
        local_irq_disable();
        uart_putc(a'+arg);
        local_irq_enable();
    }
}
```

task1.c 파일의 내용도 task0.c 파일의 내용과 같으며, 테스트 1~15까지 차례대로 문자 'b' ~ 'p'를 화면에 출력한다.

[illegible]

timer_list 구조체를 include 디렉토리 내에 timer-list.h 파일에 다음과 같이 정의한다.



```
include/timer-list.h
typedef struct _timer_list {
    process_state * head;
    unsigned int num;
    unsigned int expires_next;
} timer_list;

void init_timer_list(timer_list *);
void insert_timer_list(timer_list *, process_state *);
process_state * remove_timer_list(timer_list *, unsigned
int);
```

timer_list 구조체 내의 head 변수는 timer list에 들어온 태스크 중에서, 앞으로 올 시간 중에 현재 시간과 가장 가까운 시간에 timer list로부터 빠져 나갈 태스크를 가리킨다. num 변수는 timer list 내에 있는 태스크의 개수를 나타낸다. expires_next 변수는 head 변수가 가리키는 태스크의 expires 값과 같으며, timer list에 태스크가 없을 경우 (-1) 값을 갖는다.

process_state 구조체에 expires 멤버 변수를 다음과 같이 추가한다.

```
include/task.h
typedef struct _process_state {
    unsigned int context[18];
    unsigned int pid;
    unsigned int time_remain;
    unsigned int time_slice;
    unsigned int need_resched;
    unsigned int prior;
    struct _process_state * next;
    unsigned int expires;
} process_state;
```

expires 변수는 태스크가 timer list에 있을 경우 timer list로부터 빠져 나갈 시간을 나타낸다.

그림 1에서 jiffies는 현재 시간을 1/1000 단위로 나타내며 현재 시간은 198이다. 이 때 timer list에는 네 개의 태스크가 있으며, jiffies 값이 203이 되면 expires 값이 203인 태스크가 timer list로부터 가장 먼저 빠져 나간다. 그래서 expires_next 변수의 값이 203으로 되어 있다. 그 다음에 jiffies 값이 237이 되면 두 개의 태스크가, 303이 되면 마지막 태스크가 timer list로부터 빠져나간다.

다음은 timer-list.c 파일의 내용이다.

```

timer-list.c
#include <task.h>
#include <timer-list.h>
#include <null.h>

void init_timer_list(timer_list * ptlist)
{
    ptlist->head = NULL;
    ptlist->num = 0;
    ptlist->expires_next = -1;
}

void insert_timer_list(timer_list * ptlist, process_state *
proc)
{
    if(ptlist->num == 0) {
        ptlist->head = proc;
        ptlist->expires_next = proc-
>expires;
    } else if(proc->expires < ptlist->expires_next) {
        proc->next = ptlist->head;
        ptlist->head = proc;
        ptlist->expires_next = proc->expires;
    } else {
        int i;
        process_state * tmp = ptlist->head;
        while(1) {
            if(tmp->next == NULL) {
                tmp->next = proc;
                break;
            }
            if(proc->expires <= tmp->next-
>expires) {
                proc->next = tmp->next;

```

```

                tmp->next = proc;
                break;
            }
            tmp = tmp->next;
        }
        ptlist->num ++;
    }

process_state * remove_timer_list(timer_list * ptlist,
unsigned int jiffies)
{
    process_state * tmp = NULL;

    if(ptlist->num == 0) return (process_state *)(-
1);

    if(ptlist->expires_next <= jiffies) {
        tmp = ptlist->head;
        ptlist->head = ptlist->head->next;
        if(ptlist->head != NULL)
            ptlist->expires_next =
ptlist->head->expires;
        else ptlist->expires_next = -1;

        ptlist->num --;
        tmp->next = NULL;
    }

    return tmp;
}

```

init_timer_list 함수는 timer list를 초기화하는 역할을 한다. insert_timer_list 함수는 태스크를 timer list에 넣는 역할을 한다. 이 때 태스크의 expires 값에 따라 timer list에서 태스크가

놓일 위치가 달라진다. `remove_timer_list` 함수는 인자로 넘어가는 jiffies 값에 따라 timer list로부터 태스크를 꺼내는 역할을 한다.

이상에서 우선순위 방식의 스케줄링을 구현하기 위해 timer list 기능에 대해 알아보았다.

그러면 지금부터 우선순위 방식의 스케줄링을 구현해보자.

다음은 `task.c` 파일의 내용이다.

```
task.c
#include <task.h>
#include <null.h>
#include <prior-queue.h>
#include <timer-list.h>

process_state process[16];
process_state * current;
process_state * prev, * next;
unsigned int pid = 0;

priorQ pQ[2];
priorQ * active;
priorQ * expired;

timer_list tlist;

void init_task()
{
    int i;

    current = &process[0];
    next = current;

    process[0].context[0+2] = 0;
    process[0].context[13+2] = 0x408000;
```

```
process[0].context[1] = 0x400000;
process[0].context[0] = 0x5f;
process[0].pid = 0;
process[0].time_remain = 10;
process[0].time_slice = 10;
process[0].need_resched = 0;
process[0].prior = 15;
process[0].next = NULL;
process[0].expires = 0;

process[1].context[0+2] = 1;
process[1].context[13+2] = 0x408000;
process[1].context[1] = 0x400000;
process[1].context[0] = 0x5f;
process[1].pid = 1;
process[1].time_remain = 10;
process[1].time_slice = 10;
process[1].need_resched = 0;
process[1].prior = 0;
process[1].next = NULL;
process[1].expires = 0;

for(i=2;i<16;i++) {
    process[i].context[0+2] = i;
    process[i].context[13+2] = 0x408000;
    process[i].context[1] = 0x400000;
    process[i].context[0] = 0x5f;
    process[i].pid = i;
    process[i].time_remain = i;
    process[i].time_slice = i;
    process[i].need_resched = 0;
    process[i].prior = i-1;
    process[i].next = NULL;
    process[i].expires = 0;
}
```

```

initpriorQ(&pQ[0]);
initpriorQ(&pQ[1]);

active = &pQ[0];
expired = &pQ[1];

for(i=1;i<16;i++) enpriorQ(active, &process[i]);

init_timer_list(&tlist);
}
...
void schedule()
{
    current = depriorQ(active);
    next = current;

    flush_cache_tlb();
    if(current->pid == 0) mmu_map_l2pt(MASTER
L1PT+4, TASK0L2PTBASE);
    else if(current->pid==1) mmu_map_l2pt
(MASTERL1PT+4, TASK1L2PTBASE);
    else mmu_map_l2pt(MASTERL1PT+4, TASKL
2PTBASE(current->pid));
}

```

task.c 파일에서는 먼저 timer-list.h 파일을 추가로 include 시켜주었다. timer-list.h 파일은 timer_list 타입의 변수를 참조하기 위해 사용하였다.

timer list로 tlist 변수를 두었다. init_task 함수에서는 0번 태스크의 우선순위를 가장 낮은 15로 주었다. 1번 태스크의 경우는 우선순위를 가장 높은 0으로 주었다. 그리고 태스크의 번호가 작은 순으로 우선순위를 주었다. 예를 들어 2번 태스크는 우선순위가 1이며, 15번 태스크는 우선순위가 14이다. 모든 태스크의 expires 변수는 0으로 초기화하였다. 마지막으로 tlist

변수를 초기화하였다.

schedule 함수의 내용은 이전에 이미 설명하였다.

다음은 task1.c 파일의 내용이다.

```

task1.c
#include <system.h>

int main(unsigned int arg)
{
    int i = arg;

    for(;;) {
        uart_putc('a'+arg);
        i--;
        if(i<=0) {
            i = arg;
            alarm(16-arg);
        }
    }
}

```

task1.c 파일의 내용은 1~15번 태스크가 수행하는 루틴이다. 각 태스크는 main 함수에서 무한 루프를 돌면서 각 태스크에 해당하는 문자를 태스크의 번호만큼 찍고, alarm 매크로를 이용하여 각 태스크에 해당하는 시간만큼 timer list에 대기한다. timer list에서 빠져 나온 후에는 앞의 동작을 반복한다. 예를 들어 1번 태스크의 경우는 'b' 문자를 한 번 찍고 (15/1000)시간 동안 timer list에서 대기한 후, timer list에서 빠져나와 같은 동작을 반복한다. 15번 태스크의 경우는 'o' 문자를 15번 찍고 (1/1000)시간 동안 timer list에서 대기한 후, timer list에서 빠져나와 같은 동작을 반복한다. 여기서는 우선순위가 높을수록 timer list에 대기하는 시간이 더 많도록 하였다.

alarm은 매크로로써 include 디렉토리 내의 system.h 파일에 다음과 같이 정의하였다.

```
include/system.h
...
#define alarm(n)          \
({                          \
    unsigned long time=n;  \
    __asm__ __volatile__(  \
        "mov r1,%0\n"     \
        "swi 0x80"        \
        ::"r"(time):"r1"); \
    })
```

alarm 매크로에서는 timer list에서 대기하고자 하는 시간을 r1 레지스터를 통해 넘겨준 후 시스템 콜을 호출한다. 이 때 swi 명령어의 comment 필드에 0x80을 써 주었다.

r1 레지스터를 통해 넘어온 값을 eventsSWIHandler 함수에 넘겨주기 위하여 exceptions.S 파일의 coreSWIHandler 함수에 아래와 같이 <mov r1,r1> 부분을 추가해준다.

```
exceptions.S
...
.globl coreSWIHandler
coreSWIHandler:
    ldr    r13,=current
    ldr    r13,[r13]    @ r13->current
    add    r13,r13,#8
    stmia  r13,{r0-r14}^ @ save user registers
    mrs    r0, spsr
    stmdb  r13,{r0,r14}  @ save the rest
    ldr    r13,=svc_stack

    ldr    r10,[r13,#-4]
```

```
bic    r10,r10,#0xff000000
mov    r0,r10
mov    r1,r1
bl     eventsSWIHandler

    ldr    r13,=next
    ldr    r13,[r13]    @ r13->next
    add    r13,r13,#8
    ldmdb r13,{r0,r14}
    msr    spsr_cxsf,r0
    ldmia  r13,{r0-r14}^ @ load user registers
    movs  pc,lr         @ return next task
...
```

또, coreSWIHandler 함수에서 r1 레지스터를 통해 넘겨준 값을 받고 처리하기 위하여 swihandler.c 파일의 eventsSWIHandler 함수를 다음과 같이 수정해준다.

```
swihandler.c
#include <task.h>
#include <timer-list.h>

extern process_state * current;
extern timer_list tlist;

extern unsigned int jiffies;

void eventsSWIHandler(unsigned int syscallnum,
unsigned int arg1)
{
    if(syscallnum == 0x80) {
        unsigned int alarm_time = arg1;
        current->expires = jiffies+alarm_
time;
```

```

        insert_timer_list(&tlist, current);
        current->need_resched = 1;
    }

    if(current->need_resched == 1) {
        current->need_resched = 0;
        schedule();
    }
}

```

eventsSWIHandler 함수에서는 두 번째 인자 arg1 변수를 통하여 timer list에서 태스크가 빠져 나올 시간을 받을 수 있도록 하였다. 그리고 첫 번째 인자로 넘어온 syscallnum의 값이 0x80일 경우 현재 태스크의 expires 멤버 변수 값을 현재 시간인 jiffies 값에 두 번째 인자 arg1 변수를 통해 넘어온 값을 alarm_time 지역 변수로 받아 그 값을 더해 준 후, 현재 태스크를 timer list에 넣고 태스크 스케줄링을 요청한다. 그리고 태스크 스케줄링의 요청이 있을 경우 schedule 함수를 통하여 스케줄링을 수행한다.

다음은 irqhandler.c 파일의 내용을 수정하자.

```

irqhandler.c
#include <timer.h>
#include <task.h>
#include <prior-queue.h>
#include <timer-list.h>
#include <null.h>

extern process_state * current;
extern priorQ * expired;
extern priorQ * active;

extern timer_list tlist;

unsigned int jiffies = 0;

```

```

void eventsIRQHandler()
{
    unsigned int rintoffset;
    unsigned int rintpnd;
    process_state * pps;

    rintoffset = rINTOFFSET;
    rintpnd = rINTPND;

    jiffies ++;
    current->time_remain --;
    if(current->time_remain <= 0) {
        current->time_remain = current->time_slice;
        current->need_resched = 1;
    }

    rSRCPND |= rintpnd;
    rINTPND |= rintpnd;

    while(1) {
        pps = remove_timer_list(&tlist, jiffies);

        if(pps == NULL) goto sched;
        if(pps == (process_state *)(-1)) goto sched;

        enpriorQ(active, pps);
        if(current->prior < pps->prior) current->
need_resched = 1;
    }

    sched:
        if(current->need_resched == 1) {
            current->need_resched = 0;
            enpriorQ(active, current);
            schedule();
        }
    }
}

```

irqhandler.c 파일에서는 먼저 timer_list 구조체의 타입을 알 수 있도록 timer-list.h 파일을 include해준다. 또 null.h 파일도 include해준다. 그리고 irqhandler.c 파일에서 active 포인터 변수와 tlist 변수를 참조할 수 있도록 extern 변수로 선언하였다.

eventsIRQHandler 함수는 이전 예제와 유사하나 자세히 살펴보면 그 내용이 부분적으로 바뀌어 있다. 크게 세 부분을 살펴보기로 하자. 먼저 맨 앞부분에서는 현재 태스크의 time_remain 변수 값을 하나 감소시킨 후 그 값이 0보다 작거나 같으면 time_remain 변수 값을 time_slice 값으로 초기화시킨 후 스케줄링을 요청한다. 그러면 함수의 마지막 부분에서 스케줄링 요청을 체크하여 스케줄링 요청이 있을 경우 현재 태스크를 active 큐에 넣고 schedule 함수를 호출하여 태스크 스케줄링을 수행한다. 이전 예제에서는 맨 앞부분에서 현재 태스크를 expired 큐에 넣는 부분이 있었는데 이 부분이 빠졌음에 유의하자. 중간 부분에서는 tlist가 가리키는 timer list에 태스크가 있는지 체크하여 태스크가 있을 경우 active 큐에 태스크를 넣은 후 tlist로부터 나온 태스크의 우선순위가 현재 태스크의 우선순위보다 클 경우 스케줄링을 요청한다. 이상에서 irqhandler.c 파일의 내용을 살펴보았다.

다음은 task0.c 파일의 내용이다.

```
task0.c
#include <system.h>

int main(unsigned int arg)
{
    for(;;);
}
```

0번 태스크는 무한 루프를 돌며 어떤 경우에도 wait queue나 timer list에 대기하지 않는다. 앞에서도 보았던 것처럼 0번

태스크의 우선순위는 가장 낮아야 한다.

마지막으로 Makefile의 .S.o suffix rule과 KERNEL_OBJ 변수의 내용을 각각 다음과 같이 수정해준다.

```
...
.S.o:
    arm-linux-gcc $< -c -include
...
KERNEL_OBJ = head.o main.o mmusetup.o uart.o
exceptions.o swihandler.o timer.o \
    irqhandler.o mmu.o task.o queue.o prior-
queue.o timer-list.o
...
```

이상 작성한 내용을 컴파일하여 확인해 보면 다음과 같은 결과를 볼 수 있다.

```
...
bbcccddeeeefffffgggggghhhhhhhbbiiiiiccc
jjjjjjjjkkkkkkkkkklllddddlllllbbmmmmmmmmmm
mmmmmmnnnccnnnnnnnnnnnnnooooddddoooooooooo
oppppppppppppppppppppppppppppppppppppppp
...
```

각각의 태스크가 할당 받은 time slice와 우선순위에 따라 우선순위 방식의 스케줄링이 수행되는 걸 볼 수 있다.

지금까지 우리는 일반 큐와 우선순위 큐를 이용하여 라운드 로빈 방식과 우선순위 방식의 스케줄링을 구현해보았다.

다음 호에서는 SLEEP_ON 함수와 WAKE_UP 함수를 구현해보기로 하자. 또, 이 함수들을 이용하여 뮤텍스, 세마포어를 구현해 보고, IPC 관련 루틴도 작성해보기로 하자. R_{Time}^{real}