CaOS

(Calcium Operating System)

태스크 관리

2007.8.6 작성 시작

2007.9.3 v0.05 문서 작성

김기오

www.asmlove.co.kr

태스크 관리

1. 유저 모드 태스크 관리 개요

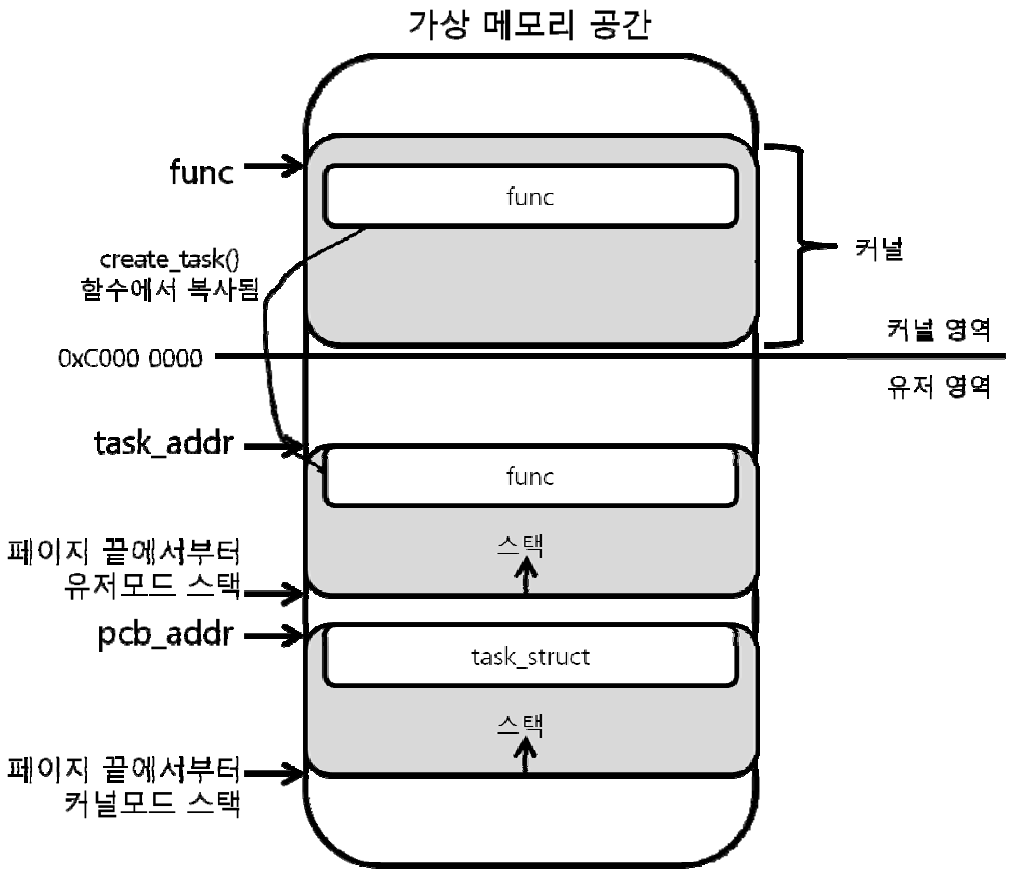
여러 개의 유저 모드 태스크를 실행하기 위해서 현재 시스템에 실행 중인 태스크의 정보를 관리하는 자료구조가 필요하다. 또 각각의 태스크는 프로그램 코드가 저장된 프로그램 영역과 태스크 관리 자료구조가 저장된 태스크 관리 영역으로 구성된다.
프로그램 영역은 유저 모드에서 실행되고, 유저 모드 스택을 포함한다. 태스크 관리 영역은 커널 모드에서 커널이 태스크의 정보를 관리하는데 사용되므로 커널모드에서 접근되고, 커널 모드 스택을 포함한다.

현재 태스크를 생성하는 함수 create_thread에서는 2개의 페이지를 할당받아서 하나는 태스크로 실행되는 함수 코드를 복사하고 하나는 TCB와 커널 모드 스택으로 사용한다.

2. 태크스 테이블과 태스크 생성

현재 실행중인 태스크들의 정보를 관리하기 위해 태스크 테이블을 만든다. 이 테이블은 태스크 관리 영역의 주소 값을 배열로 저장한 것이다. 태스크의 pid가 배열의 인덱스가 되므로, init 태스크는 pid가 0이고, 첫번째 엔트리를 읽으면 init 태스크의 task_struct 구조체 주소를 얻을 수 있다. 태스크 테이블에 항목을 넣거나 빼기 위한 함수 add_task_table, del_task_table를 만든다. (추후 링크드 리스트를 위한 매크로를 정리한 후 add_task_list, del_task_list로 변경해서 태스크 배열이 아니라 태스크 리스트로 관리한다.)

TSS 영역은 현재 태스크가 커널 모드로 진입했을 때 사용할 커널 모드 스택의 포인터를 저장하는데 사용한다. 원래는 프로세서마다 하나씩 개별적인 TSS 영역을 가져야 하지만 현재 프로세서를 1개만 사용하도록 제한하고 있으므로 1개의 TSS 영역만 만든다. setup.asm 파일에서 물리 메모리 0x90000 에 TSS 영역의 시작 주소를 저장하고, process.c에서 TSS의 주소를 읽어서 TSS에 접근한다.



커널에는 유저 태스크로 실행될 init(), user1(), user2() 등의 함수가 있다. create_task() 함수는 이런 태스크들을 유저 영역 메모리의 한 페이지 프레임으로 복사한다. 그리고 그 다음 페이지 프레임에 task_struct 정보를 저장한다.

3. process.h

```
#ifndef __PROCESS_H__
#define __PROCESS_H__

#include "processor.h"
#include "memory.h"

#include "q.h"
```

태스크의 상태를 나타내는 상수 값, 태스크 스위칭에서 태스크가 실행될 때는

PROCESS_RUNNING 상태로 표시하고 태스크가 잠시 중단될 때는 PROCESS_INTERRUPTIBLE 상태로 표시한다. 태크스가 종료되면 PROCESS_STOPPED이다.

```
/* declare process's statements... */
#define  PROCESS_RUNNING                    1
#define  PROCESS_INTERRUPTIBLE      2
#define  PROCESS_STOPPED                    4
```

5개까지 태스크를 만들 수 있다.

```
/* maximum task number */
#define MAX_TASK 5
```

커널 모드 스택과 TCB를 같은 페이지 안에 저장한다. TCB는 페이지의 시작 부분에 저장되고 커널 모드 스택은 페이지의 끝 부분에서 시작되므로 겹칠 일이 없다.

```
/* Each task has a kernel-mode stack of one page size */
#define  PROCESS_STACK_SIZE       PAGE_SIZE
```

```
/* INIT task PID is 0 */
#define  INIT_PID 0
```

TSS의 주소는 물리 메모리 0x2000 으로 고정되어 있다.

```
#define TSS_ADDR                    0xc0002000
```

각 태스크마다 128개의 키보드 입력을 받을 수 있다.

```
#define KEY_INPUT_MAX 128
```

TCB 구조체에는 상태 정보, 프로세스 ID, 태스크 이름 등등 태스크를 실행하고 관리하기 위한 정보가 저장된다.

```
/* process table... */
typedef struct _task_struct
{
        int         status;
        int         pid;
        char name[16];
        char key_input[KEY_INPUT_MAX];
        int key_offset;

        unsigned int priority;
```

```
        unsigned int counter;

        unsigned int sleep;

        unsigned int task_addr;
```

모든 태스크가 동일한 가상 메모리에서 실행되도록 하기 위해서 고유의 페이지 디렉토리와 테이블을 할당해준다. 태스크가 저장된 물리 메모리가 달라도 가상 메모리 주소는 같게 된다.

```
        unsigned long cr3;
```

컨텍스트를 저장하는 데이터 구조를 따로 만들어준다. TSS와 같은 모양을 가진다.

```
        thread_struct thread;

} task_struct;



void start_init(void);
void init_process(void);

void setup_task(void);
int add_task(task_struct *new);
int del_task(int);

void init_cpu_tss(void);
int create_task(void (*)(void), char *);

void set_foreground(task_struct *);


extern task_struct *fore_task;



#endif
```

4. process.c

```
#include "process.h"
#include "console.h"
#include "segment.h"
#include "types.h"
#include "gdt.h"


#include "page.h"
#include "memory.h"


디버그 메시지를 출력하지 않는다.
//#define DEBUG 1


//
// setup.asm에서 저장한 TSS 영역의 주소
//
tss_struct *cpu_tss;

//
// 각 태크스들의 task_struct 주소를 배열로 관리
//
task_struct *task_table[MAX_TASK];


//
// number of created tasks
unsigned int task_count;

커널이 부팅할 때 임시로 사용하는 페이지 디렉토리, 유저모드 태스크가 생성되면서 유저모드 태스크의 페이지 디렉토리로 복사될 뿐, 커널에 의해 직접 사용되지는 않는다.
extern pgd_t *swapper_pg_dir;

현재 foreground 태스크로 실행되는 태스크의 포인터
```

```
task_struct *fore_task;
```

태스크 관리 테이블을 초기화하기 위해서 모든 엔트리를 NULL 값으로 클리어한다.

```
//
// clear table entries
//
void setup_task(void)
{
	int i;
	for (i=0; i<MAX_TASK;i++) {
		task_table[i] = NULL;
	}

	task_count = 0;
	fore_task = NULL;

}
```

태스크를 생성하면 태스크 테이블에 TCB의 포인터를 저장하고 테이블 엔트리를 태스크 ID로 반환한다.

```
//
// find empty entry and store new
// , return pid (process ID)
//
int add_task(task_struct *new)
{
	int i = 0;

	for (i=0; i<MAX_TASK;i++) {
		if (task_table[i] == NULL) {
			task_table[i] = new;
			return i;
		}
	}

	return -1;

}
```

태스크를 삭제할 때는 포인터를 지운다.

```
//
// remove id(th) entry
//
int del_task(int id)
{
        if (task_table[id] == NULL)
                return -1;
        task_table[id] = NULL;
        return 1;
}
```

foreground로 실행되는 태스크만 키보드 입력을 받을 수 있다.

```
void set_foreground(task_struct *ts)
{
        fore_task = ts;
}
```

태스크를 만들기 위해서는 태스크로 실행될 함수와 태스크 이름이 필요하다.

```
//
// func : task function pointer
// name : task name
//
int create_task(void (*func)(void), char *name)
{
        int l;
        char *src;          // task function
        char *dst;          // user memory space to execute task
        task_struct *tcb;   // task control block

        pgd_t *pgd;         // page dir
        pte_t *pte;         // page table
        unsigned long addr;


        caos_printf("Create task [%s]...", name);
```

한 페이지 물리메모리를 할당받아서 페이지 디렉토리로 사용한다. 기본적으로 모든 데이터는 가상 주소로 접근해야 한다. 필요할 때에만 물리주소로 변환해서 사용해야 한다.

```
        // 1 page memory for a page dir, a task has one page DIR
        pgd = (pgd_t *)get_free_pages(0, 0);
        if (pgd == NULL)
                goto err_pgd;

#ifdef DEBUG
        caos_printf("pgd->%x ", (unsigned long)pgd);
#endif
```

한 페이지 물리메모리를 할당받아서 페이지 테이블로 사용한다.

```
        // 1 page memory for a page table
        // When task is created, only one table is setup.
        pte = (pte_t *)get_free_pages(0, 0);
        if (pte == NULL)
                goto err_pte;

#ifdef DEBUG
        caos_printf("pte->%x ", (unsigned long)pte);
#endif
```

TCB를 만들기 위한 한 페이지 메모리를 할당받는다.

```
        // add task_struct to task table
        tcb = (task_struct *)get_free_pages(0,0);
        if (tcb == NULL)
                goto err_tcb;

#ifdef DEBUG
        caos_printf("tcb->%x ", (unsigned long)tcb);
#endif
```

태스크가 저장될 물리 메모리를 할당받는다.

```
        // store entry point of task
        tcb->task_addr = get_free_pages(0,0);
        if (tcb->task_addr == (unsigned long)NULL)
                goto err_task;
```

```
#ifdef DEBUG
        caos_printf("task->%x ₩n", (unsigned long)tcb->task_addr);
#endif
```

생성한 TCB를 태스크 테이블에 저장하고 태스크 ID를 할당받는다.

```
        tcb->pid = add_task(tcb);

        // There is no empty entry
        if (tcb->pid < 0) {
                caos_printf("Too many tasks₩n");
                goto err_task;
        }

#ifdef DEBUG
        caos_printf("PID=%d₩n", tcb->pid);
#endif
```

태스크로 실행될 함수를 새로 할당받은 메모리에 복사한다.

```
        //
        // Copy function to user space so that the func executes in user mode.
        // In user mode, function in kernel space cannot run.
        //
        src = (char *)func;
        dst = (char *)tcb->task_addr;
        for (l=0 ; l < PAGE_SIZE; l++) // copy one page
                *dst++ = *src++;

        //
        // initialize TCB
        //

        tcb->status = PROCESS_RUNNING;


        // build page dir/table
        // DIR=0x100, TABLE=0x0 -> 0x4000 0000
        // User task is stored at virtual address 0x4000 0000
#ifdef DEBUG
```

```
        caos_printf("setup paging..");
#endif
```

TCB에 페이지 디렉토리의 주소를 저장한다.

```
        tcb->cr3 = __pa((unsigned long)pgd);        // physical memory
```

유저 모드 태스크를 항상 0x4000 0000 번지에서 실행되도록 하기 위해서 페이지 디렉토리의 0x100번 엔트리에 페이지 디렉토리의 주소를 저장한다. 접근 권한은 0x7로 설정한다.

```
        addr = (unsigned long)__pa(pte) + (_PAGE_RW|_PAGE_PRESENT|_PAGE_USER);
        set_pgd(pgd+0x100, __pgd(addr)); // 0x100th entry
```

0번 페이지 테이블 엔트리에 태스크가 저장된 페이지의 물리 주소를 저장한다. 태스크가 한 페이지만을 사용하므로 하나의 엔트리만 설정하면 된다.

```
        addr = (unsigned long)__pa(tcb->task_addr) +(_PAGE_RW|_PAGE_PRESENT|_PAGE_USER);
        set_pte(pte, __pte(addr));                  // 0x0th entry
```

커널이 사용하고 있던 swapper_pg_dir의 내용을 유저 태스크의 페이지 디렉토리에 복사한다. swapper_pg_dir의 0번 엔트리와 0x300~0x3FF 엔트리가 사용되고 있으므로 이 엔트리 값들을 모두 복사한다. 만약 태스크가 인터럽트나 예외에 의해 실행이 중단되고 커널 모드로 바뀌어도 페이지 디렉토리를 바꿀 필요가 없다. 0x300번 이후의 엔트리에 커널의 페이지 디렉토리가 복사되어 있으므로 페이지 디렉토리를 바꾸지 않아도 커널이 실행되는데 문제가 없다.

```
        //*******************************************************//
        //
        // User task share page DIR [0], [300]~[3FF] with kernel.
        // When user task is created,
        // DIR[0][300~3FF] must be copied.
        //
        //*******************************************************//
        pgd[0x0].pgd = swapper_pg_dir[0x0].pgd;

        for (i=0x300; i<0x400; i++) {
                pgd[i].pgd = swapper_pg_dir[i].pgd;
#ifdef DEBUG
                //if (pgd[i].pgd != 0)
                        //caos_printf("pgd[%x]=%x  swapper[%x]=%x₩n", i, pgd[i].pgd, i,
swapper_pg_dir[i].pgd);
#endif
        }
```

```
#ifdef DEBUG
        caos_printf("CR3=%x PGD[0x100]=%x PTE[0x0]=%x₩n", tcb->cr3, pgd[0x100].pgd, pte[0].pte_low);
#endif

#ifdef DEBUG
        pte = (pte_t *)0xc0004000;
        caos_printf("kernel pte[0x36]=%x₩n", pte[0x36].pte_low);
#endif
```

이제 유저 태스크의 컨텍스트를 설정하고 태스크 스위칭을 준비한다.

```
        // initialize point of task code, user-stack, kernel-stack
```

태스크가 실행될 가상 주소를 설정하고 유저 모드로 동작할 때 사용할 스택도 설정한다. 유저 모드 스택은 태스크가 저장된 페이지의 끝에서 시작된다.

```
        tcb->thread.eip = 0x40000000;
        tcb->thread.esp = 0x40000000+PAGE_SIZE;
```

커널 모드 스택은 인터럽트 핸들러나 스케줄러가 실행될 때 사용할 스택이다. 따라서 크기가 작아도 되므로 TCB와 동일한 페이지를 사용한다.

```
        tcb->thread.esp0 = (unsigned long)tcb+PAGE_SIZE;

        tcb->thread.cs = __USER_CS;
        tcb->thread.eflags = 0x200;         // interrupt enable
        tcb->thread.ss = __USER_DS;
        tcb->thread.ds = __USER_DS;
        tcb->thread.es = __USER_DS;
        tcb->thread.fs = __USER_DS;
        tcb->thread.gs = __USER_DS;

        for (l=0; name[l] != '₩0'; l++)
                tcb->name[l] = name[l];

        caos_printf("%s (PID=%d) OK₩n", tcb->name, tcb->pid);

        task_count++;

#ifdef DEBUG
        caos_printf("Task count=%d₩n", task_count);
```

```
#endif

여기까지 문제없이 실행되었으면 태스크 ID를 반환하고 종료한다.
        return tcb->pid;

        caos_printf("fail₩n");

문제가 생겼으면 할당된 메모리를 해제시키고 종료한다.
err_task:
        free_pages((unsigned long)tcb->task_addr, 0);

err_tcb:
        free_pages((unsigned long)tcb, 0);

err_pte:
        free_pages((unsigned long)pte, 0);

err_pgd:
        free_pages((unsigned long)pgd, 0);

        return -1;

}


//
// TSS segment must be initialized to run user-mode tasks
//
void init_cpu_tss(void)
{

        // TSS for CPU
        cpu_tss = (tss_struct *)TSS_ADDR;


        caos_printf("CPU TSS is at->[%x]₩n", (unsigned int)cpu_tss);

        // Kernel Stack segment = 0x10
        cpu_tss->ss0 = __KERNEL_DS;
```

LTR 레지스터에 TSS의 세그먼트 셀렉터 값을 저장하면 TSS 설정이 끝난다.

```
        // TSS selector is 0x20 (5th entry in GDT)
        asm volatile (
                "cli                                                    ₩n₩t"
                "mov $0x20, %%ax        ₩n₩t"
                "ltr %%ax                       ₩n₩t"
                "sti                            ₩n₩t"
                ::
        );
}
```

init.c 파일에 정의된 init() 함수를 시작한다. 커널 설정이 끝나고 가장 먼저 실행되는 태스크가 init 태스크이다.

```
//
// execute user-mode task, init
//
void start_init(void)
{
        task_struct *init_pcb = task_table[0];
        unsigned int reg;



/*
 * This code is for init task creating debugging
 *
#ifdef DEBUG
        thread_struct *ts;
        unsigned long cr3;



        caos_printf("init_pcb->%x reg->%x₩n", init_pcb, reg);

        ts = (thread_struct *)reg;
        caos_printf("eip-%x cs-%x elfags-%x esp0-%x esp-%x₩n", ts->eip, ts->cs, ts->eflags, ts->esp0, ts->esp);

        caos_printf("cr3=%x₩n", task_table[0]->cr3);
```

```
        cr3 = task_table[0]->cr3;

        caos_printf("cpu_tss->%x init_pcb->%x swapper_pg_dir->%x cr3->%x reg->%x₩n",
                        (unsigned long)cpu_tss, (unsigned long)init_pcb, (unsigned
long)swapper_pg_dir,
                        (unsigned long)cr3, (unsigned long)reg);

        caos_printf("cpu_tss->esp0=%x cpu_tss->cr3=%x init_pcb->thread.esp0=%x₩n",
                        cpu_tss->esp0, cpu_tss->cr3, init_pcb->thread.esp0);
#endif
*/

        //
        // set foreground task
        //
        set_foreground(init_pcb);
```

init 태스크의 컨텍스트가 저장된 주소를 계산한다.

```
        reg = (unsigned int)&(init_pcb->thread);

        asm volatile (
                "cli                    ₩n₩t"
                /*
                 * Before task start, TSS must be initialized.
                 * Therefore kernel stack pointer is store at cpu_tss->esp0.
                */
```

커널 스택 포인터를 TSS에 복사한다.

```
                "mov %2, %%eax          ₩n₩t"
                "mov %%eax, %0          ₩n₩t"

                // (back-up kernel page DIR <- old ver.)
                // Do not change cr3 with kernel page DIR
                // Use the same page DIR with user mode task,
                // because DIR[0][300~3FF] is shared, it is not necessary.
```

init 태스크가 사용하는 페이지 디렉토리의 주소를 TSS 영역에 복사하고 cr3 레지스터에도 저장한다.

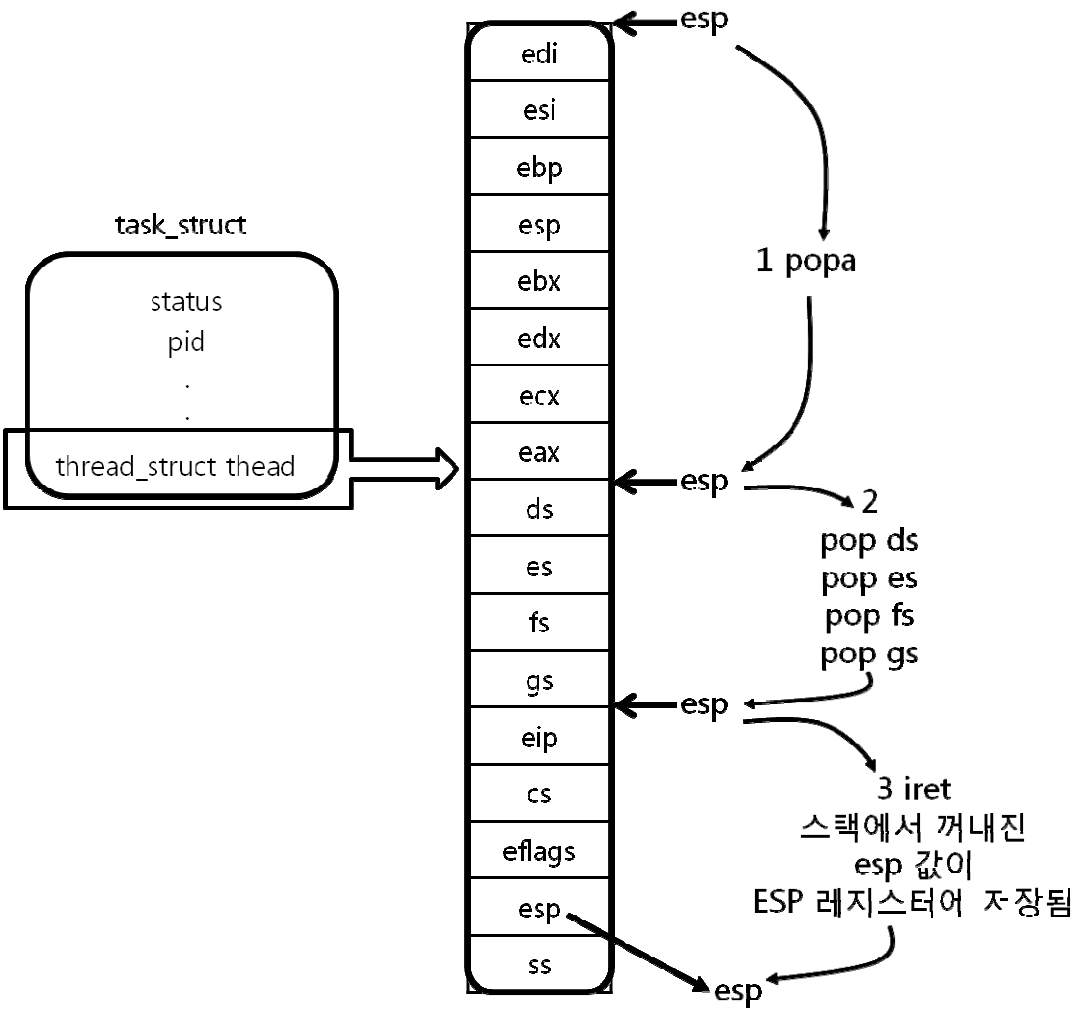
```
                "mov %4, %%eax          ₩n₩t"
                "mov %%eax, %1          ₩n₩t"
```

```
                // store user page DIR
                "mov %4, %%eax          ₩n₩t"
                "mov %%eax, %%cr3               ₩n₩t"
```

init 태스크의 컨텍스트가 저장된 영역을 스택으로 설정하고 컨텍스트를 꺼낸다. iret 명령을 실행하면서 init 태스크가 실행된다.

```
                "mov %5, %%esp          ₩n₩t"
                "popa                           ₩n₩t"
                "pop %%ds                               ₩n₩t"
                "pop %%es                               ₩n₩t"
                "pop %%fs                               ₩n₩t"
                "pop %%gs                               ₩n₩t"
                "sti                                            ₩n₩t"
                "iret                                           ₩n₩t"
                : "=m"(cpu_tss->esp0), "=m"(cpu_tss->cr3)
                :   "m"(init_pcb->thread.esp0),   "m"(swapper_pg_dir),   "m"(task_table[0]->cr3),
"m"(reg)
        );
}
```

thread_struct 구조체는 아래 그림과 같은 순서로 태스크의 컨텍스트가 저장되어 있다. popa 명령어를 실행하면 eax~edi 까지 레지스터 값이 꺼내지고 그 다음 세그먼트 레지스터를 꺼낸다. 그리고 iret 명령을 실행하면 태스크의 첫번째 명령의 세그먼트와 오프셋 주소와 스택 포인터와 스택 세그먼트 레지스터 값이 꺼내진다. 그러면 스택 포인터는 thread_struct 를 가르키는 것이 아니라 create_thread에서 설정된 유저 모드 스택 값으로 설정된다.


task_struct
status
pid
.
.
thread_struct thead
edi
esi
ebp
esp
ebx
edx
ecx
eax
ds
es
fs
gs
eip
cs
eflags
esp
ss
esp
1 popa
esp
2
pop ds
pop es
pop fs
pop gs
esp
3 iret
스택에서 꺼내진
esp 값이
ESP 레지스터에 저장됨
esp