Introduction to Operating Systems Project #2: Multitasking and Context Switching

04/28/2022

Prof. Seongsoo Hong
sshong@redwood.snu.ac.kr
SNU RTOSLab
Dept. of Electrical and Computer Engineering
Seoul National University

Agenda

I. **Goal of Project #2**
II. Project Description
III. Learn to Use GNU AS
IV. Project Submission

과제 목적

- 과제 목적
  - OS 상에서 커널의 핵심 기능 중 하나인 멀티태스킹이 동작하는 원리를 이해
  - 멀티태스킹 메커니즘을 구현
    - 문맥 전환을 담당하는 eOS API들을 구현

Agenda

I. Goal of Project #2
II. **Project Description**
III. Learn to Use GNU AS
IV. Project Submission

과제 내용 (1)

- 구현해야 할 API 목록
  - `hal/linux/context.c`
    1. _os_context_t 구조체 – 문맥의 내용을 정의
    2. _os_save_context() 함수 – 문맥의 저장
    3. _os_restore_context() 함수 – 저장된 문맥의 복구
    4. _os_create_context() 함수 – 새로운 문맥의 생성
  - `core/eos.h`
    5. eos_tcb_t 구조체 – 태스크 컨트롤 블록
  - `core/task.c`
    6. eos_create_task() 함수 – 태스크의 생성
    7. eos_schedule() 함수 – 태스크 스케줄링

과제 내용 (2)

1. _os_context_t 구조체
   - 문맥 전환 시 저장해야 할 CPU 레지스터의 종류와 순서를 결정
   - 뒤에 나오는 _os_save_context()와 _os_restore_context()의 설명 그림 참고

과제 내용 (3)

2. _os_save_context() 함수

형	addr_t _os_save_context(void)
정의된 위치	core/eos_internal.h
구현된 위치	hal/linux/context.c
간단한 설명	현재 문맥을 스택에 저장하고 그 스택 포인터를 리턴
입력	없음
출력	문맥을 저장한 직후: 문맥이 저장된 스택의 스택 포인터 문맥이 복귀된 직후: NULL

과제 내용 (4)

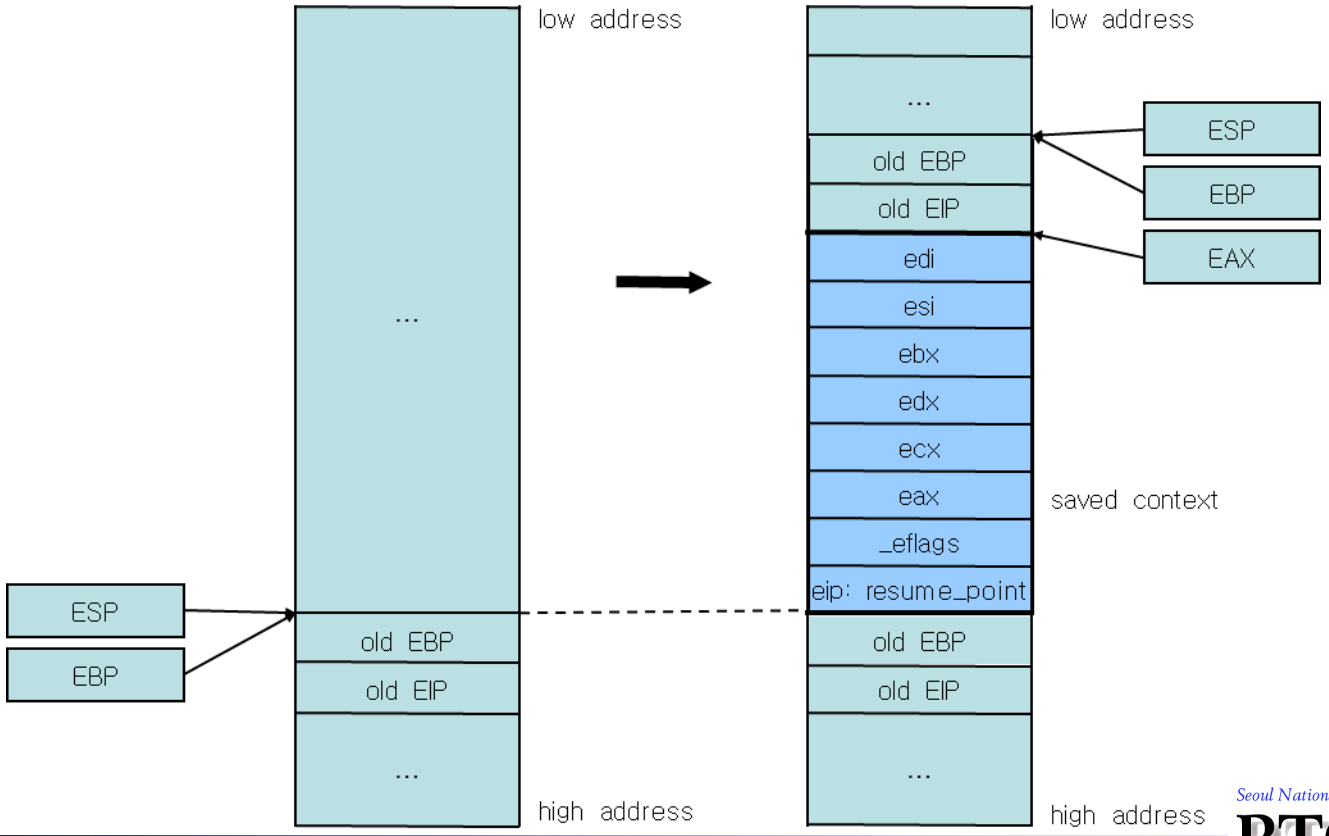
2. _os_save_context() 함수 (cont.)

- 인라인 어셈블리를 이용하여 개별 레지스터 저장 (GNU AS 사용법 참조)
- 수행 흐름
  1. 저장해야 할 레지스터들(문맥)을 스택에 저장
     - EIP는 문맥이 복구되었을 때 수행을 재개할 위치의 주소
  2. 리턴할 스택 포인터를 EAX에 저장
  3. 저장한 문맥을 보호하기 위해 old EIP와 old EBP를 복사하고 EBP 변경
  4. 리턴
- Hint
  - 프로젝트 1에서의 C 서브루틴을 확실히 알아야 함
    - leave, ret, call instruction이 구체적으로 뭘 하는지 이해해야 함

과제 내용 (5)

2. _os_save_context() 함수 (cont.)

- 시작 직후와 리턴 직전의 스택 모습



과제 내용 (6)

3. _os_restore_context() 함수

형	void _os_restore_context(addr_t sp)
정의된 위치	core/eos_internal.h
구현된 위치	hal/linux/context.c
간단한 설명	저장된 문맥의 내용으로 복귀
입력	sp: 복구할 문맥이 저장된 스택의 꼭대기 주소
출력	없음

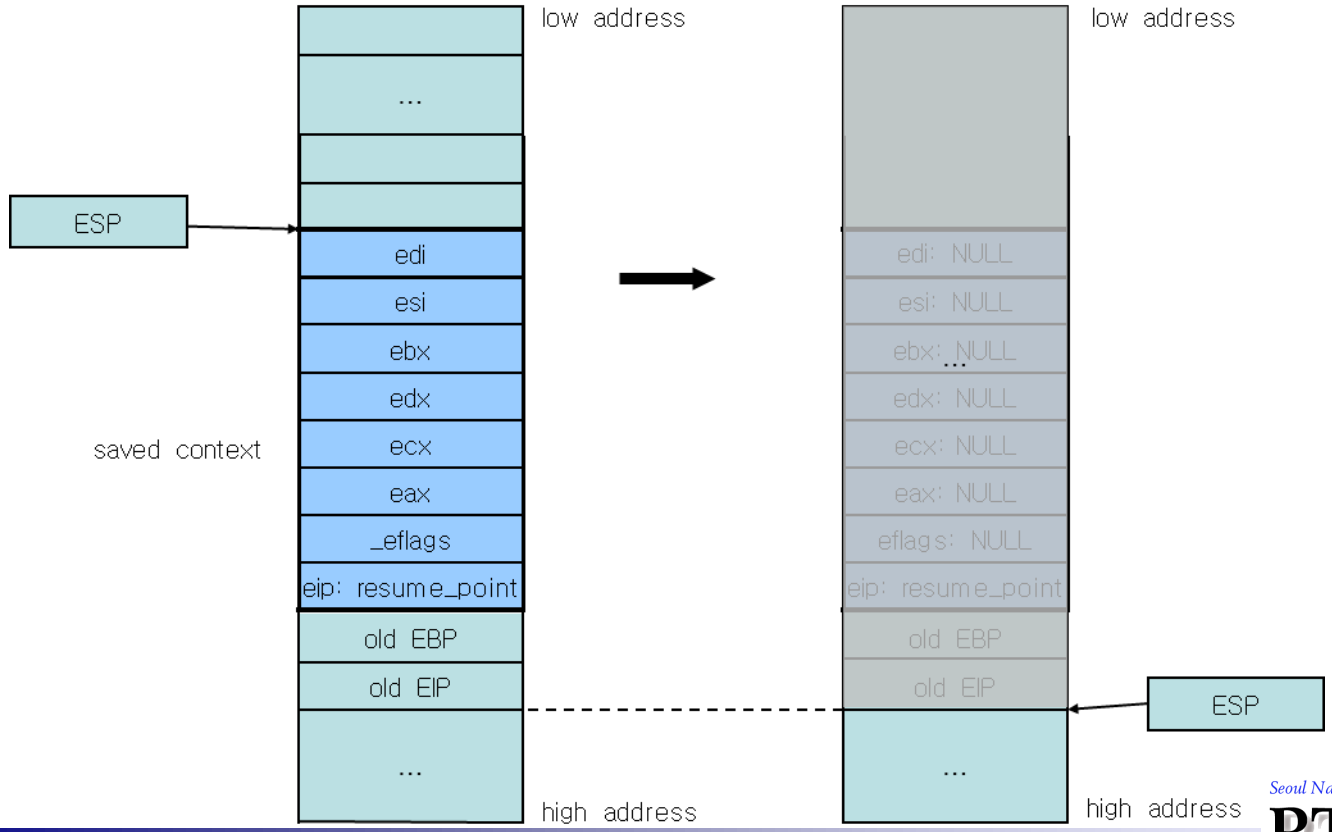
과제 내용 (7)

3. _os_restore_context() 함수 (cont.)
   - 인라인 어셈블리를 이용하여 개별 레지스터 복구 (GNU AS 사용법 참조)
   - 수행 흐름
     1. 스택 포인터를 문맥이 저장된 위치로 바꿈
     2. 저장된 레지스터의 값들을 차례로 복구
     3. resume_point로 control flow가 바뀜
     4. _os_save_context() 함수에서 리턴하면서 old EBP와 old EIP를 복구

과제 내용 (8)

3. _os_restore_context() 함수 (cont.)

- 시작 직후와 리턴 직전의 스택 모습



과제 내용 (9)

4. _os_create_context() 함수

형	addr_t _os_create_context(addr_t stack_base, size_t stack_size, void (*entry)(void *arg), void *arg);
정의된 위치	core/eos_internal.h
구현된 위치	hal/linux/context.c
간단한 설명	주어진 스택에 새로운 문맥을 만듬
입력	**stack_base:** 스택 영역의 시작 주소 (low address) **stack_size:** 스택 영역의 크기 (단위: byte) **entry:** 새로 만들어진 문맥이 복귀할 경우 가장 먼저 수행되어야 하는 명령어의 주소 **arg:** entry 함수가 수행될 때 전달될 인자 값
출력	생성된 문맥의 시작 주소

과제 내용 (10)

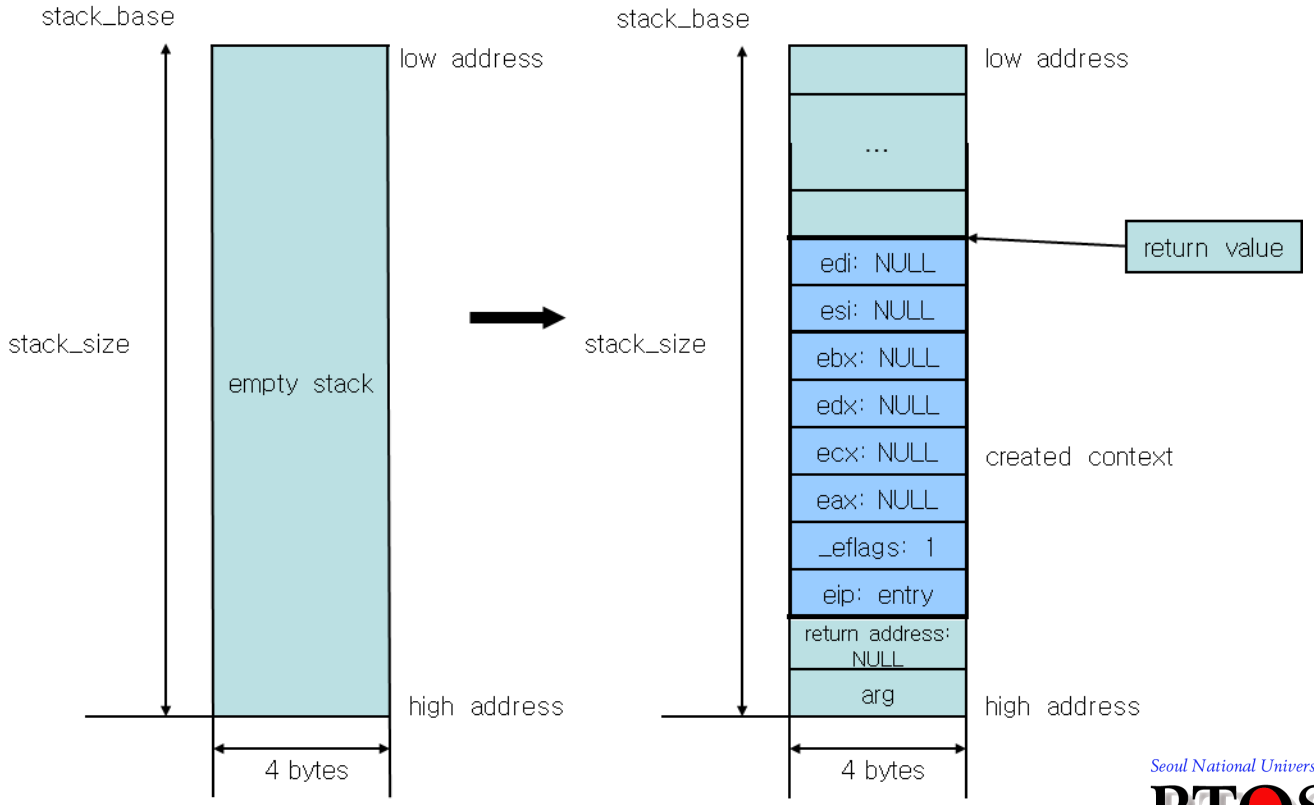
4. _os_create_context() 함수 (cont.)

- 인라인 어셈블리 사용할 필요 없음 (포인터 이용)
- 수행 흐름
  1. entry 함수에 전달될 매개변수(arg)와 entry 함수가 리턴할 주소(return address)를 저장
     - EOS에서 entry 함수는 리턴하지 않으므로 return address는 NULL
  2. entry 함수가 시작될 때의 레지스터들(문맥)을 스택에 저장
     - _eflags와 eip를 제외한 레지스터들은 don't care이므로 NULL
     - _eflags는 1
     - eip는 entry 함수 주소
  3. 문맥이 저장된 위치를 리턴하여 함수 종료

과제 내용 (11)

4. _os_create_context() 함수 (cont.)

- 수행 전후의 스택 모습



과제 내용 (12)

5. eos_tcb_t 구조체
   - 커널이 관리하는 태스크의 정보들을 가지고 있는 구조체
     - 태스크 state, 우선순위, 주기, 스택 포인터 등
   - 구현에 필요한 정보들을 추가할 것
     - 프로젝트가 진행됨에 따라 필요한 정보가 늘어남

과제 내용 (13)

6. eos_create_task() 함수

형	int32u_t eos_create_task(eos_tcb_t *task, addr_t sblock_start, size_t sblock_size, void (*entry)(void *arg), void *arg, int32u_t priority)
정의된 위치	core/task.c
간단한 설명	태스크를 생성하고 초기화
입력	task: 이 태스크의 정보를 가지고 있는 TCB 구조체 sblock_start: 태스크에게 할당된 스택의 시작 주소 sblock_size: 스택의 크기 entry: 태스크로 실행될 함수의 주소 arg: entry 함수에 넘겨질 인수 priority: 태스크의 우선순위
출력	0

과제 내용 (14)

6. eos_create_task() 함수 (cont.)
   - _os_create_context() 함수를 호출하여 새로운 문맥을 생성하고 tcb에 스택 포인터 기록

과제 내용 (15)

7. eos_schedule() 함수
   - 현재 수행 중인 태스크를 중단하고 새로운 태스크로 문맥 전환
   - 수행 흐름
     1. _os_save_context() 함수를 호출하여 현재 수행중인 태스크의 문맥을 저장하고 리턴 값에 따라 다음과 같이 처리
        1. 리턴 값이 NULL이 아닌 경우 (정상적으로 문맥을 저장한 경우)
           1. 리턴 값(스택 포인터)을 tcb에 기록하고 _os_restore_context() 함수를 호출하여 새로 수행시킬 태스크의 문맥을 복구
        2. 리턴 값이 NULL인 경우 (문맥이 복구된 직후)
           1. 함수를 종료시킴
     2. 현재 수행 중인 태스크가 없는 경우(최초 호출 시) 문맥을 저장할 필요 없이 복구만 하면 됨

Agenda

I. Goal of Project #2
II. Project Description
III. Learn to Use GNU AS
IV. Project Submission

GNU AS (GAS)

- GAS 인라인 어셈블리
  - C 소스코드 안에 내장된 어셈블리 코드
  - eOS의 문맥 전환 루틴, 인터럽트 처리 루틴에 사용됨

사용법 (1)

- 인라인 어셈블리 명령어 형식

__asm__ __volatile__ (<*asms*>:<*output*>:<*input*>:<*clobber*>);

  - <*asms*>
    - GAS 문법에 따르는 어셈블리 코드
    - 쌍따옴표로 둘러싸인 문자열로 표현됨
    - %n의 형태로 입력, 출력 인자들을 사용
      - 예를 들어 세 개의 인자를 사용하는 경우 각 인자는 순서대로 %0, %1, %2와 같이 표현됨
  - <*output*>
    - <*asms*>에서 사용된 출력 인자를 지정
      - 각 인자는 "=<*const*>"(<*var*>)의 형식으로 표현됨
      - 여러 개의 인자는 쉼표로 구분
      - <*const*>는 constraint로 출력 인자가 어느 곳에 위치하는지를 지정

사용법 (2)

- 인라인 어셈블리 명령어 형식 (cont.)
  - *<input>*
    - *<asms>*에서 사용된 입력 인자를 지정
      - *<output>*과 마찬가지로 각 인자는 "=*<const>*"(*<var>*)의 형식으로 표현되며, 여러 개의 인자를 쉼표로 구분
      - *<output>*과 동일한 constraint를 가짐
  - *<clobber>*
    - *<output>*, *<input>*에 나오진 않았지만 해당 어셈블리를 수행한 결과로 값이 바뀌는 레지스터들을 나타냄
    - 레지스터 이름을 쌍따옴표로 둘러싸인 문자열로 표현
    - 여러 개의 레지스터 이름은 쉼표로 구분
    - eOS에서는 이 필드를 사용할 일이 없으므로 자세한 설명은 reference를 참조

사용법 (3)

- 인라인 어셈블리 명령어 형식 (cont.)
  - Constraint의 종류
    - ‘m’: memory operand
      - 인자가 메모리에 위치해야 함
    - ‘r’: register operand
      - 인자가 범용 레지스터에 위치해야 함
    - ‘i’: immediate operand
      - 인자가 정수와 symbol 값(주소)이어야 함
    - ‘n’:
      - 값을 정확히 알고 있는 정수
    - 기타:
      - ‘q’, ‘b’, ‘c’, ‘f’, …

사용법 (4)

- 인라인 어셈블리 명령어 형식 (cont.)
  - 레지스터 이름 직접 사용
    - 레지스터 이름에 %%를 붙여서 직접 명시할 수 있음
    - 용례 1) 현재 stack pointer의 값을 읽어 옴

```
unsigned int sp;
__asm __volatile("movl %%esp, %0": "=m"(sp));
```

    - 용례 2) 스택 포인터를 0x10000번지로 변경

```
unsigned int sp = 0x10000;
__asm __volatile("movl %0, %%esp": : "m"(sp));
```

Agenda

I. Goal of Project #2
II. Project Description
III. Learn to Use GNU AS
IV. Project Submission

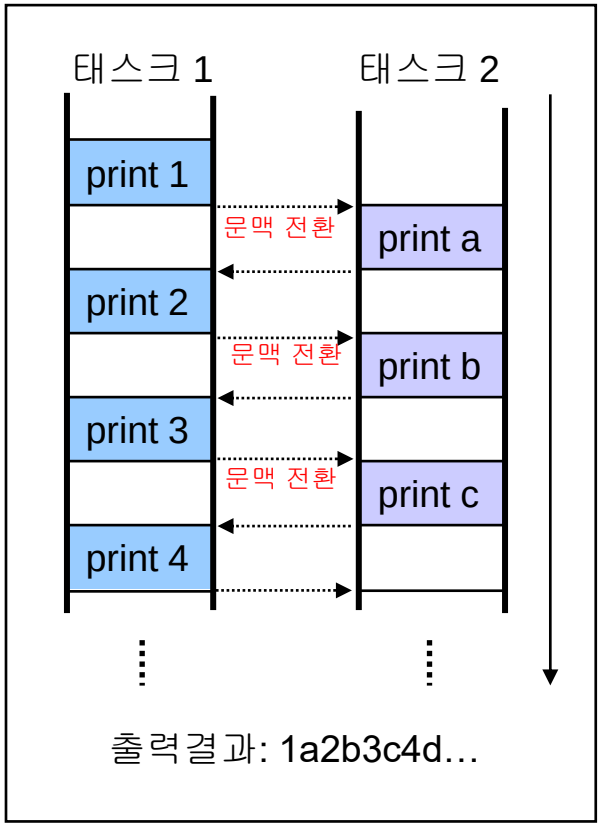
제출물 (1)

- 제출물
  - 수정한 EOS 코드
  - 보고서
    - 정의한 구조체에 대한 설명
    - 구현한 함수들에 대한 설명
    - 테스트 프로그램 수행 결과

- 주의사항
  - 주어진 함수 prototype을 변경하지 말 것
  - 가능한 새로운 함수나 글로벌 변수 추가를 하지 말 것
  - 보고서 분량은 5페이지 이내

제출물 (2)

- 테스트 프로그램
  - 태스크 1 생성
    - 1, 2, 3, 4의 숫자열을 무한히 출력
  - 태스크 2 생성
    - a, b, c, d의 문자열을 무한히 출력
  - 각 태스크는 한 문자를 출력한 후 eos_schedule() 함수를 호출하여 다른 태스크로 점유권을 넘김
  - 결국 1a2b3c4d…와 같은 문자열을 화면에 출력하게 된다



제출물 (3)

```
#include <core/eos.h>
#define STACK_SIZE 8096

int8u_t stack1[STACK_SIZE];    // stack for task1
int8u_t stack2[STACK_SIZE];    // stack for task2
eos_tcb_t tcb1;            // tcb for task1
eos_tcb_t tcb2;            // tcb for task2

/* task1 function - print number 1 to 20 repeatedly */
void print_number() {
    int i = 0;
    while(++i) {
        printf("%d", i);
        eos_schedule();                // 태스크 1 수행 중단, 태스크 2 수행 재개
        if (i == 20) { i = 0; }
    }
}

/* task2 function - print alphabet a to z repeatedly */
void print_alphabet() {
    int i = 96;
    while(++i) {
        printf("%c", i);
        eos_schedule();                // 태스크 2 수행 중단, 태스크 1 수행 재개
        if (i == 122) { i = 96; }
    }
}

void eos_user_main() {
        eos_create_task(&tcb1, stack1, STACK_SIZE, print_number, NULL, 0)                    // 태스크 1 생성
        eos_create_task(&tcb2, stack2, STACK_SIZE, print_alphabet, NULL, 0)                  // 태스크 2 생성
}
```

제출 기한과 방법

- 제출 기한
  - 5/12(목) 11:59 PM 전까지

- 제출 방법: ETL에 업로드
  - 보고서는 워드 파일 또는 PDF
  - 소스 코드는 make clean을 하여 정리 후 압축
    - hal/current를 삭제 후 압축하는 것을 권장
      - 압축 에러의 원인이 되는 경우가 있기 때문
      - symbolic link일 뿐이기에 삭제해도 make all을 할 때 다시 생성

Question or Comment?