프로젝트 2 보고서 2014-17952 정재철

• 정의한 구조체

1. \_os\_context\_t

컨텍스트의 구성을 알려주는 참고용 구조체이다. 이번 프로젝트에서 직접 사용되지는 않았다. 프로젝트 안내문에 나온 레지스터에 더하여 ebp 레지스터를 추가했는데, 낮은 주소부터 edi, esi, ebp, ebx, edx, ecx, eax, eflags, eip 순서이다.

1. eos\_tcb\_t

태스크 컨트롤 블락이다. 어떤 태스크의 상태(state), 스택 포인터(sp), 스택의 시작 주소(sblock\_start), 스택의 사이즈(sblock\_size), 함수 포인터(entry), 함수 인자(arg), 프로세스 큐에 들어가기 위한 노드(node)로 구성되어있다. 특징은 노드를 다이나믹하게 할당하는 게 번거롭고 한 노드와 한 컨트롤 블락이 1대 1 관계로 되어 있기 때문에 노드 포인터를 사용하지 않고 정적 변수로서 가지고 있다.

• 구현한 함수들

1. \_os\_create\_context (hal/linux/context.c)

컨텍스트 구조체 설명에 나와있는 것과 동일하게 스택 탑부터 값들을 푸쉬했다. 리턴값은 스택 탑에 있는 edi의 주소가 된다.

1. \_os\_restore\_context (hal/linux/context.c)

\_\_asm\_\_ \_\_volatile\_\_(

"mov %0, %%esp\n"

"pop %%edi\n"

"pop %%esi\n"

"pop %%ebp\n"

"pop %%ebx\n"

"pop %%edx\n"

"pop %%ecx\n"

"pop %%eax\n"

"pop \_eflags\n"

"ret\n"

::"r"(sp)

);

복원해야 할 컨텍스트 탑을 가리키는 주소가 인자로 주어진다. esp에 이를 넣고 전부 pop 하면 현재 레지스터에 이전 컨텍스트가 모두 복원된다. 참고로 eflags는 전역 변수인 \_eflags를 사용하고, eip는 ret 명령어를 이용하여 복원한다. (ret가 pop eip; jmp eip와 같은 기능을 한다.)

1. \_os\_save\_context (hal/linux/context.c)

\_\_asm\_\_ \_\_volatile\_\_(

"push $resume\_point\n"

"push \_eflags\n"

"push $0\n"

"push %%ecx\n"

"push %%edx\n"

"push %%ebx\n"

"push %%ebp\n"

"push %%esi\n"

"push %%edi\n"

"mov %%esp, %%eax\n"

"push 4(%%ebp)\n"

"push 0(%%ebp)\n"

"mov %%esp, %%ebp\n"

"resume\_point:\n"

"leave\n"

"ret\n"

::

);

현재 컨텍스트를 스택에 푸쉬하는 함수이다. 함수가 정상적으로 리턴되기 위해서 old ebp와 eip를 푸쉬하고 ebp를 esp로 바꾸는 과정을 거친다. 컨텍스트가 복구된 후에(스케줄의 마지막 단계) 이 함수가 지정한 resume\_point로 돌아오게 되는데, 이 때 프로세스가 스케줄 함수를 호출할 때 저장한 eip(복구 후 가야 할 주소)를 얻기 위해서는 스케줄 함수에서 바로 리턴해야 하기 때문에 eax 자리에 0을 넣음으로서 리턴값을 마음대로 컨트롤 할 수 있다. (4번째 줄)

1. eos\_create\_task (core/task.c)

eos\_tcb\_t에서 선언한 모든 변수에 값을 저장해준다. 레디 큐에 이 태스크를 추가하기 위해서 <core/common.c>에 구현된 \_os\_add\_node\_tail 함수를 사용했다.

(참고) \_os\_add\_node\_tail (core/common.c)

if (\*head) {

new\_node -> previous = (\*head) -> previous;

(new\_node -> previous) -> next = new\_node;

new\_node -> next = (\*head);

(new\_node -> next ) -> previous = new\_node;

} else {

(\*head) = new\_node;

new\_node -> previous = new\_node;

new\_node -> next = new\_node;

}

이 함수는 레디 큐의 마지막 노드에 새로운 노드를 추가하는데, 레디 큐가 Circular하고 헤드의 previous가 마지막 노드를 가리키고 마지막 노드의 next가 헤드를 가리키게 만든다.

1. eos\_schedule (core/task.c)

스케줄 함수의 플로우가 이번 프로젝트에서 가장 중요한 부분이다.

우선 현재 실행되는 태스크가 있는지 확인한다. 만약 있으면, 컨텍스트를 저장하고 스택 포인터를 얻는다. 현재 실행되는 태스크를 레디 상태로 만들고 레디 큐에 넣는다.

그리고 다음에 실행할 태스크를 정한다. 이 때 우선순위만 고려한다. 이 태스크를 현재 실행할 태스크로 지정하고 러닝 상태로 만든다. 그리고 레디 큐에서 제거하기 위하여 <core/common.c>에 구현이 주어진 \_os\_remove\_node 함수를 사용했다.

(참고) \_os\_remove\_node (core/common.c)

if (node->next == NULL || node->previous == NULL) return 0;

if (node -> next == node) {

(\*head) = NULL;

} else {

(node -> previous) -> next = node -> next;

(node -> next) -> previous = node -> previous;

if (node == \*head) {

\*head = node -> next;

}

}

node->next = NULL;

node->previous = NULL;

return 1;

따라서 노드를 레디 큐에서 제거함을 뜻한다. 제거된 노드의 특징은 next와 previous가 모두 NULL이라는 것이다.

태스크의 컨텍스트를 복구하는 과정

\_os\_restore\_context 함수를 호출하면 세이브 했던 컨텍스트로 레지스터가 바뀌게 된다. 이 때 resume\_point 주소로 컨트롤 플로우가 바뀌는데, 이는 곧바로 leave, ret로 이어진다. 따라서 \_os\_save\_context가 맨 처음 호출된 다음 부분(sp가 0이면 eos\_schedule 함수를 리턴하는 줄)이 실행되는 것이다.

* 테스트 프로그램 수행 결과

무한 루프 돌던 함수를 1000번 반복하고 종료하도록 수정 한 후 얻은 화면이다.

