Direct Execution

OSTEP Study week2

PT. 허 완

Warning

이 내용은 **UNIX system** 기반으로
설명되어 있습니다.

목차

01. Direct Execution

Direct Execution
정의 및 동작과정

Direct Execution
문제점

02. Limited Direct Execution

Limited Direct Execution
1. Restricted Operations (제한된 명령)
2. Switching Between process (process 간 전환)

01. Direct Execution

CPU virtualization

- **Time Sharing을 통해 CPU virtualization 달성 가능**

- 고려해야 할 사항 (Challenge)

  1. **performance** (overhead 감소)

  2. **control** (run process efficiently, resources ...)

Direct Execution

- CPU virtualization을 달성하기 위한 기법

- Run the program directly on cpu

  -> 프로그램을 CPU에서 직접 동작하는 방식

Direct Execution 동작 과정

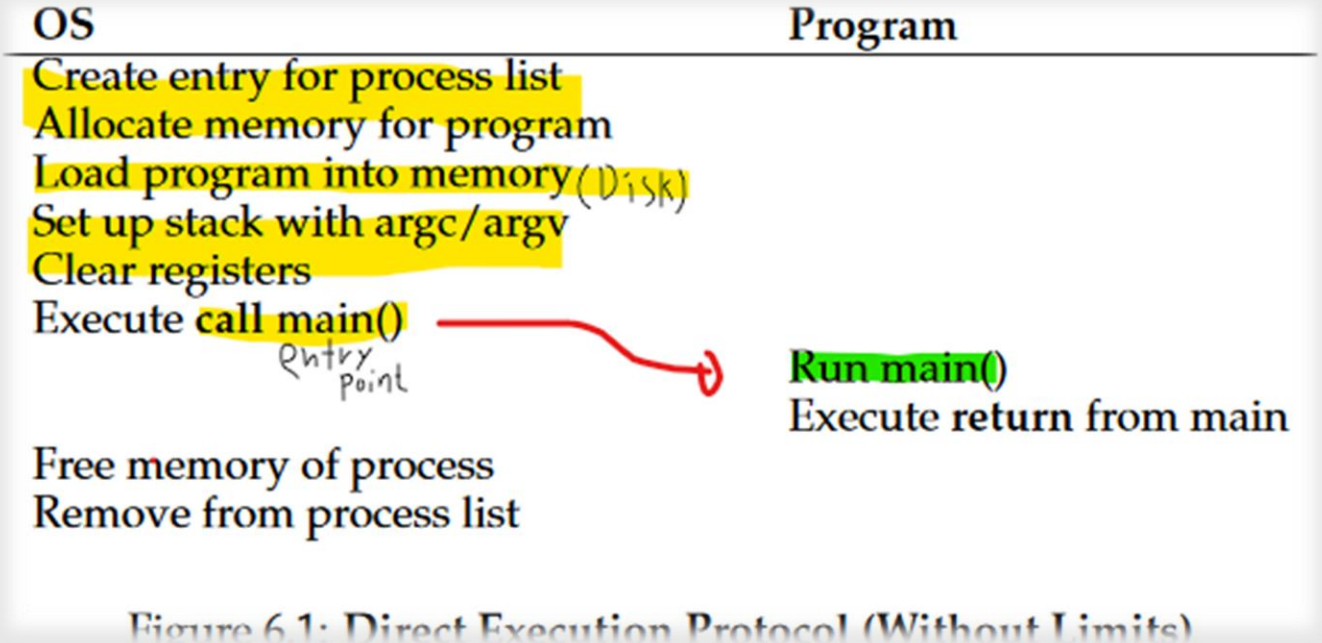


Figure 6.1: Direct Execution Protocol (Without Limits)

Direct Execution의 문제점

- OS가 program이 놀고 있는지, 의도하지 않은 동작을 하는지 알 수 없음

- 어떻게 실행하는 process를 중단하고, 다른 proces로 교체할지
  (time sharing 기법)

02.
Limited Direct Execution

Limited Direct Execution

- **Direct Execution 문제점 해결 방안** (to CPU virtualization)

  1. **Restricted Operations** (제한된 명령)

  2. **Switching between Process** (process 간 전환)

Restricted Operations

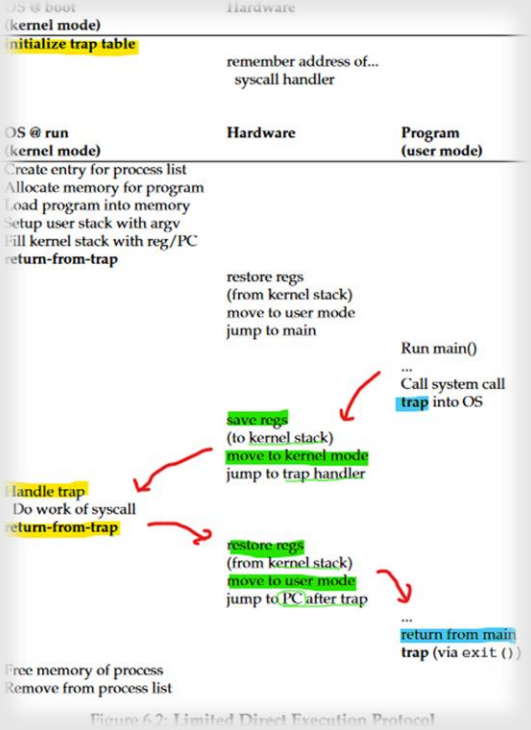
- **Process에세 file 접근 권한을 주면 process는 disk 어느 곳이든 읽고 쓰는게 가능 -> protection X**

- 해결 방안

  1. **user mode & kernel mode** 분리

  2. **user proces가 privileged operation을 사용 가능한 syscall** 제공

System Call동작 과정 (with LDE Protocol)



- **Trap table 설정 이유는??**
  -> Kernel의 어떠한 곳이든 jump 가능하면, 독단적인 code 실행이 가능하여, 보안에 위협이 됨

- **LDE의 2가지 phase**

  1. OS boot 시 trap table initialize 및 주소 저장

  2. OS run 시 여러가지 동작들 (trap, syscall ...)

Switching Between Process

- **Switching Between Process** (process 간 전환)
  1. **A Cooperative Approach: Wait For System**
  2. **A Non-Cooperative Approach: The OS Takes Control**
  3. **Saving and Restoring**

A Cooperative Approach

- A Cooperaative Approach: Wait For System

  1. OS가 porcess를 신뢰
  2. 오랫동안 실행되는 process가 syscall, trap등을 사용하여 cpu 포기 결정

- 문제점

  -> infinite loop이 발생하고 syscall을 호출하지 않는 process 해결 불가

A Non-Cooperative Approach

- **A Non-Cooperative Approach: The OS Takes Control**

  1. **timer interrupt** 도입을 통해 앞의 문제 해결
  2. **process boot** 시 **OS**는 **timer** 실행

Saving and Restoring

- **Saving and Restoring**

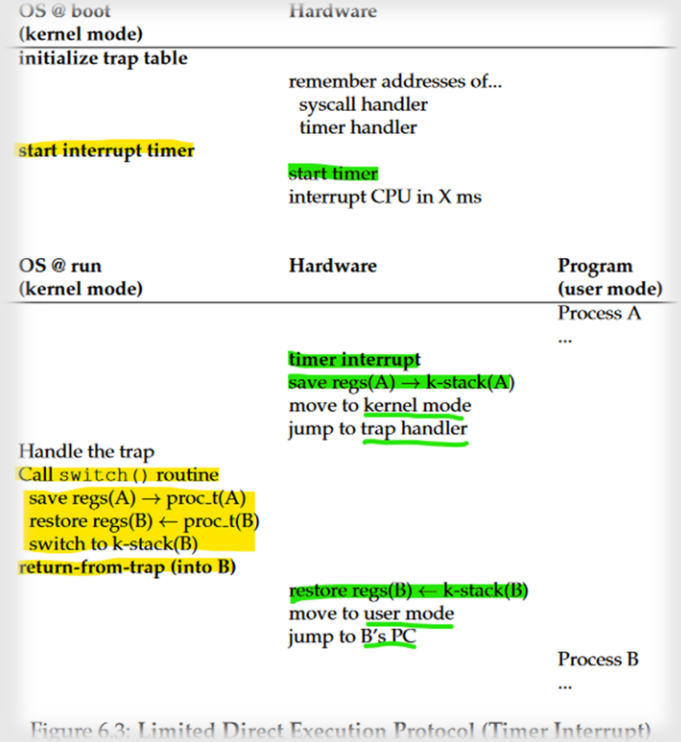
  1. **OS**의 프로세스 전환은 **Scheduler**에 의해 발생
  2. **process** 전환시 **Context switch** 발생

Context Switch

- **Context :** 프로세스/스레드의 **상태** (cpu, memory)

- **OS**가 **현재 실행 프로세스에 레지스터 값을 저장** (in kernel stack) 하고,
  곧 실행할 프로세스 레지스터 값을 복원 (from kernel stack) 하는 것

Context Switch 동작 과정 (with LDE Protocol)



OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of... syscall handler timer handler	
start interrupt timer		
	start timer	
	interrupt CPU in X ms	
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)

- 위 과정에서 2번의 register saves/restores 발생

  1. timer interrupt 발생 시

  2. OS가 process를 전환할 시

Ps. Context Switch Code

```
# void swtch(struct context *old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax  # put old ptr into eax
  popl 0(%eax)        # save the old IP
  movl %esp, 4(%eax)  # and stack
  movl %ebx, 8(%eax)  # and other registers
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)

  # Load new registers
  movl 4(%esp), %eax  # put new ptr into eax
  movl 28(%eax), %ebp # restore other registers
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp  # stack is switched here
  pushl 0(%eax)       # return addr put in place
  ret                 # finally return into new ctxt
```

Figure 6.4: The xv6 Context Switch Code

Default Question

- 만약 system call 동작 도중 timer interrupt가 발생한다면?
  (Concurrency prob)
  -> CPU가 하나의 interrupt를 처리중이라면, 다른 interrupt는 CPU에 전달되지 않음
  -> lock기법을 통해 동시에 접근하는 것을 방지

- LDE에서 'Limited'가 붙은 이유는?
  → 프로세스 하나가 끝날때까지 계속 진행하면 효율이 떨어지기 때문에 프로세스에 제한을 두었기 때문

Questions