Limited Direct Execution

Operating System (OS)

안인규

강의자료는 KAIST 원유집 교수님 OS 강의 자료를 참고했습니다.

How to efficiently virtualize the CPU with control?

- OS는 time sharing을 통해 physical CPU를 공유해야 함.
- Issue
  - **Performance**: 과도한 시스템 오버헤드를 추가하지 않고 가상화를 구현하는 방법?
  - **Control**: CPU에 대한 제어를 유지하면서 프로세스를 효율적으로 실행하는 방법?

Direct Execution

- Just run the program directly on the CPU.

OS	Program
1. Create entry for process list 2. Allocate memory for program 3. Load program into memory 4. Set up stack with `argc` / `argv` 5. Clear registers 6. Execute call `main()`	
	7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

제한 (limit) 없이 프로그램을 실행하면,
운영체제는 아무것도 제어할 수 없다!

Recap: Process Creation

1. **프로그램 코드를 메모리에 로드(load)**하여 프로세스의 주소 공간에 배치
   - 프로그램은 처음에 실행 가능한 형식(executable format)으로 디스크에 저장
   - 초기(또는 단순한) OS는 로딩 과정이 즉시(eagerly) 수행되었으며, 즉, 프로그램 실행하기 전에 한번에 모든 코드와 데이터 로드
   - 현대 OS는 지연(lazily) 방식으로 로딩을 수행; 즉, 프로그램 실행 중 필요한 코드나 데이터를 그때그때 로드→Paging & Swapping

2. 프로그램의 **run-time stack**을 위한 메모리를 할당
   - Stack은 *local variables*, *function parameters*, and *return address*을 저장하기 위해 활용
   - Initialize the stack with arguments
     - `argc` and the `argv` array of `main()` function

Process Creation

3. 프로그램의 **heap** 생성
   - Heap은 동적 메모리 할당 (dynamic memory allocation)을 위해 사용
   - Program request such space by calling `malloc()` and free it by calling `free()`.

4. OS는 입출력 (I/O) 설정도 수행
   - 각 process는 기본적으로 세 개의 file descriptors(standard input, standard output, and standard error)를 가진다 (Unix 시스템에서) → Persistence

5. **Start the program** running at the entry point, namely `main()`.
   - The OS *transfers control* of the CPU to the newly-created process. ???

Problem

- Issue 1
  - User can do wrong thing.
  - What if?

```
int *i ;
i = 0  ;
*i = 1 ;
```

- Issue 2
  - Getting the control back from CPU is not easy.
  - What if?

```
i = -1 ;
while (i <0)
    do something;
```

Problem1: Restricted Operation

- 만약 Process가 다음과 같은 제한된 작업을 수행하려고 한다면?
  - 디스크에 대한 I/O 요청을 수행
  - CPU나 메모리 같은 추가적인 시스템 자원에 접근

- **Solution 1:** 모든 권한을 부여
- **Solution 2**: Protected control transfer 사용
  - User mode: Application은 하드웨어 자원에 대한 전체 접근 권한이 없음
  - Kernel mode: OS는 시스템의 모든 자원에 접근 가능

System Call

- 커널이 사용자 프로그램에 다음과 같은 특정 핵심 기능을 신중하게 노출하도록 허용
  - 파일 시스템에 접근
  - 프로세스 생성 및 삭제
  - 다른 프로세스와의 통신
  - 추가 메모리 할당

Key Concept of System Call

- Trap?
  - User code → Kernel mode
- Trap이 발생하는 원인?
  - Interrupt (external interrupt)
  - Exception (internal interrupt)
  - **System call (Software trap)**

Key Concept of System Call

- **Trap** instruction

  `int n`

  - 특권 레벨 (privilege level)을 커널 모드로 상승
  - Kernel stack에 Register 값을 저장 (`eip, cs, eflags, esp, ss`)
  - Kernel에서 목적지를 찾음
  - 해당 목적지로 점프
  - 시스템 콜을 위한 인터럽트 번호: `0x64` @ xv6, `0x30` @Pintos
    - e.g. `int 0x64`
- **Return-from-trap** instruction
  - 호출한 user program으로 복귀
  - 특권 레벨 (privilege level)을 user model로 낮춤

What code to run in trap?

- trap handler: 각 interrupt number (trap number)에 대해 실행될 코드
- Trap table
  - Trap handler들의 주소를 저장하는 Table
  - 하드웨어는 부팅 시 OS에 trap table의 위치를 알림
  - Trap table의 위치를 설정하는 명령어는 privileged instruction 임
  - 즉, 해당 명령어는 user mode에서 실행할 수 없음

Limited Direction Execution

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of … syscall handler

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from -trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() … Call system **trap** into OS

Limited Direction Execution (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		... return from main trap (via `exit()`)
Free memory of process Remove from process list		

Problem 2: Switching Between Processes

- OS가 CPU의 제어권을 다시 획득하여 process를 전환(Switching)하는 방법은?
  - A cooperative Approach: **Wait for system calls**
  - A Non-Cooperative Approach: **The OS takes control**

A cooperative Approach: Wait for System calls

- Process는 `yield` 과 같은 **System call**을 실행하여 주기적으로 CPU 사용을 포기
  - The OS decides to run some other task.
  - Application also transfer control to the OS when they do something illegal.
    - Divide by zero
    - Try to access memory that it shouldn't be able to access
  - E.g., Early versions of the Macintosh OS, The old Xerox Alto system

Process가 무한 루프 (infinite loop)에 빠진다면?
→ Reboot the machine!

A Non-Cooperative Approach: OS Takes Control

- **A timer interrupt**
  - During the boot sequence, the OS start the timer.
  - The timer raise an interrupt every few milliseconds.
  - Interrupt가 발생한다면?
    - 현재 실행 중인 process 가 일시 정지 (Suspended)된다.
    - Process의 상태를 저장하여 나중에 다시 실행할 수 있도록 함
    - OS의 Interrupt handler가 실행된다.

A timer interrupt gives OS the ability to run again on a CPU (takes control)

Timer Interrupt

- Programming the timer interval
  - CPU가 주기적으로 interrupt를 발생하도록 설정
  - Single CPU: PIT (Programmable Interval Timer)
  - Multiple CPU: APIC(Advanced Programmable Interrupt Controller)
    - LAPIC: Local APIC per processor
    - IO APIC on system bus

```
// timer
while (1)
    i = INTERVAL ;
    while (--i > 0) ;
    raise interrupt ;
```

Saving and Restoring Context

- Scheduler makes a decision:
  - **현재 process** 를 계속 실행할지, 또는 **다른 process** 로 전환할지를 결정
  - 만약 다른 process 로 전환하기로 결정되면, OS는 Context Switch를 실행

Context Switch

- Context Switch 수행 과정
  - 현재 실행 중인 프로세스의 상태 저장
    - 현재 프로세스의 CPU Register 값 (PC, ESP, General purpose registers 등)을 Kernel Stack에 저장
    - 현재 프로세스의 PCB (Process Control Block)에 해당 정보를 업데이트
  - 새로운 프로세스의 상태 복원
    - OS가 다음 실행할 프로세스를 결정 (Scheduler)
    - 새로운 프로세스의 PCB에서 CPU 레지스터 값 복원
    - 해당 프로세스의 kernel Stack을 활성화 ()
  - CPU 제어권을 새로운 프로세스로 넘김
    - User mode로 복귀 → 새로운 프로세스 실행

Limited Direct Execution Protocol (Timer interrupt)

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address 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) to k-stack(A) move to kernel mode jump to trap handler	

Limited Direct Execution Protocol (Timer interrupt)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
Handle the trap Call switch() routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B …

The xv6 Context Switch Code

```
1  # void swtch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl swtch
6  swtch:
7          # Save old registers
8          movl 4(%esp), %eax          # put old ptr into eax
9          popl 0(%eax)                # save the old IP
10         movl %esp, 4(%eax)          # and stack
11         movl %ebx, 8(%eax)          # and other registers
12         movl %ecx, 12(%eax)
13         movl %edx, 16(%eax)
14         movl %esi, 20(%eax)
15         movl %edi, 24(%eax)
16         movl %ebp, 28(%eax)
17
18         # Load new registers
19         movl 4(%esp), %eax          # put new ptr into eax
20         movl 28(%eax), %ebp         # restore other registers
21         movl 24(%eax), %edi
22         movl 20(%eax), %esi
23         movl 16(%eax), %edx
24         movl 12(%eax), %ecx
25         movl 8(%eax), %ebx
26         movl 4(%eax), %esp          # stack is switched here
27         pushl 0(%eax)               # return addr put in place
28         ret                         # finally return into new ctxt
```

Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations:
  - **Disable interrupts** during interrupt processing
  - Use a number of sophisticated **locking** schemes to protect concurrent access to internal data structures.