Chapter-2

운영 체제

정내훈

2022년 가을학기
게임공학과
한국공학대학교

제 1 편
가상화

CPU 가상화

- 운영체제는 여러 개의 가상 CPU가 존재한다는 환상을 제공한다.

- 시분할(Time Sharing) : 프로세스 하나를 실행하고, 멈추고, 다른 프로세스를 실행하고.. 의 반복
  - 공짜는 아님 : 프로세스가 많으면 느려짐

프로세스

프로세스는 실행중인 프로그램이다..

- 구성 요소
  - 메모리 (주소 공간)
    - 명령어
    - 데이터
  - 레지스터
    - 프로그램 카운터 (PC)
    - 스택 포인터 (SP)
    - 일반 레지스터

프로세스 API

- 현대 운영체제가 제공하는 API
  - 생성 (Create)
    - 폴더에서 아이콘 클릭
  - 제거(Destroy)
    - ALT+F4
  - 대기 (Wait)
    - ???????
  - 각종 제어(Miscellaneous Control)
    - ???????
  - 상태 (Status)
    - 작업관리자

프로세스 API

- 현대 운영체제가 제공하는 API
  - 생성 (Create)
    - 폴더에서 아이콘 클릭
  - 제거(Destroy)
    - ALT+F4
  - 대기 (Wait)
    - ???????
  - 각종 제어(Miscellaneous Control)
    - ???????
  - 상태 (Status)
    - 작업관리자

우리는 사용자가 아니라 프로그램 제작자다!!!

프로세스 API

- 현대 운영체제가 제공하는 API
  - 생성 (Create)
    - 프로그램 실행을 위한 새 프로세스 생성
  - 제거(Destroy)
    - 프로세스의 소멸
  - 대기 (Wait)
    - 다른 프로세스의 멈춤을 기다림
  - 각종 제어(Miscellaneous Control)
    - 일시 정지 같은 추가 기능
  - 상태 (Status)
    - 프로세스의 정보를 얻어내기

자세한 프로세스 생성 (1/2)

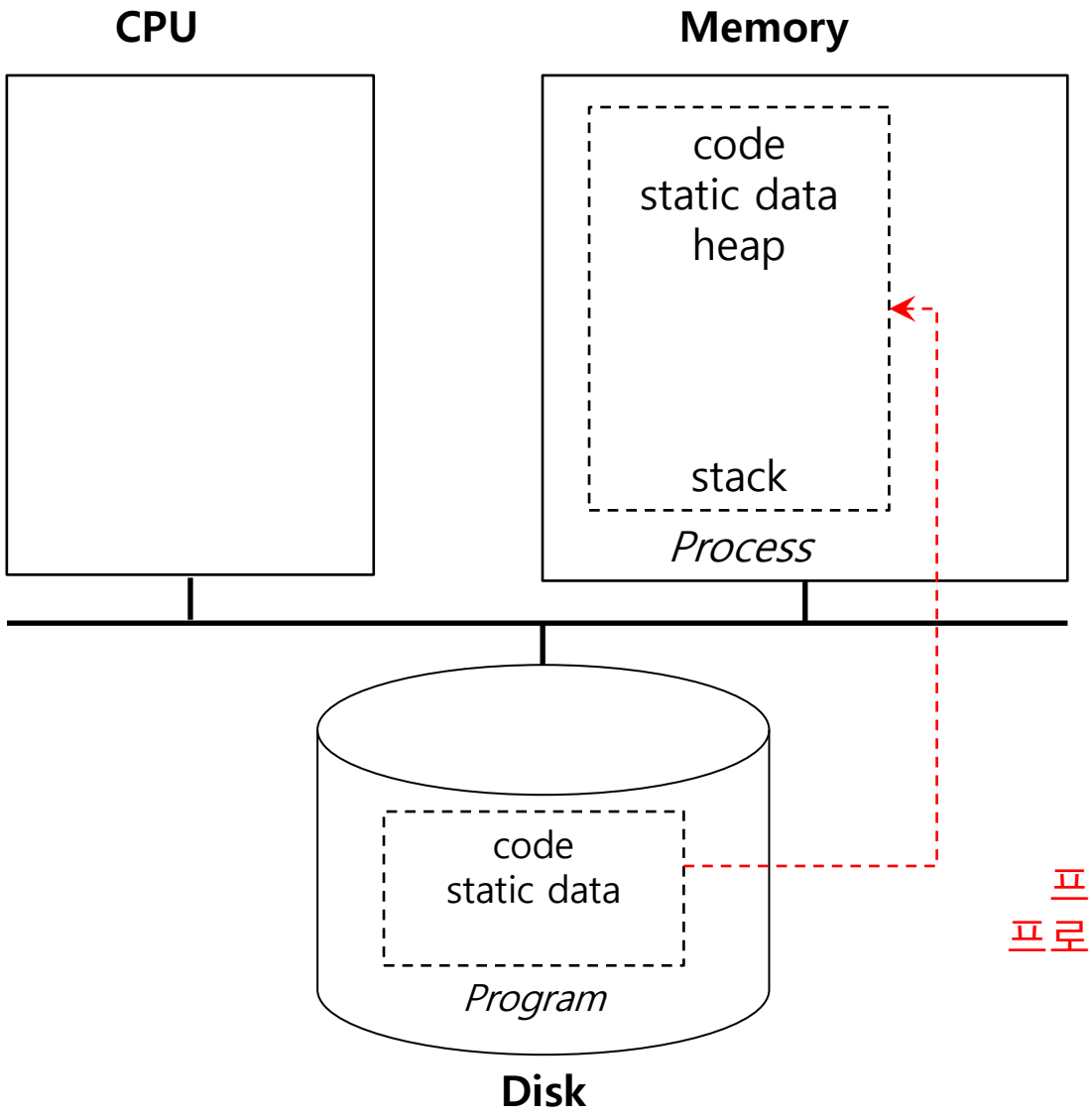
- 프로그램 파일을 메모리에 탑재(load)한다.
  - 프로세스 주소공간에 탑재
  - 프로그램 파일 자체가 실행 가능한 포맷으로 작성되어 있음
  - (게으른(Lazy)적재 : 로딩할 때 등록만 해놓고 실제 DISK에서 읽는 것은 CPU가 그 데이터가 를 필요로 할 때 수행한다.)

- 실행 스택(run-time stack) 할당
  - 지역변수, 함수 매개변수, 복귀 주소 저장용 공간
  - main()의 argc와 argv를 스택에 저장

자세한 프로세스 생성 (2/2)

- 힙(heap)할당
  - malloc이나 new가 사용하는 메모리 할당 공간 확보
- 여러가지 초기화
  - Input/Output 초기화 : stdin, stdout, stderr 파일 설정
- main() 으로 jump
  - 실행의 시작.

탑재 (Loading)

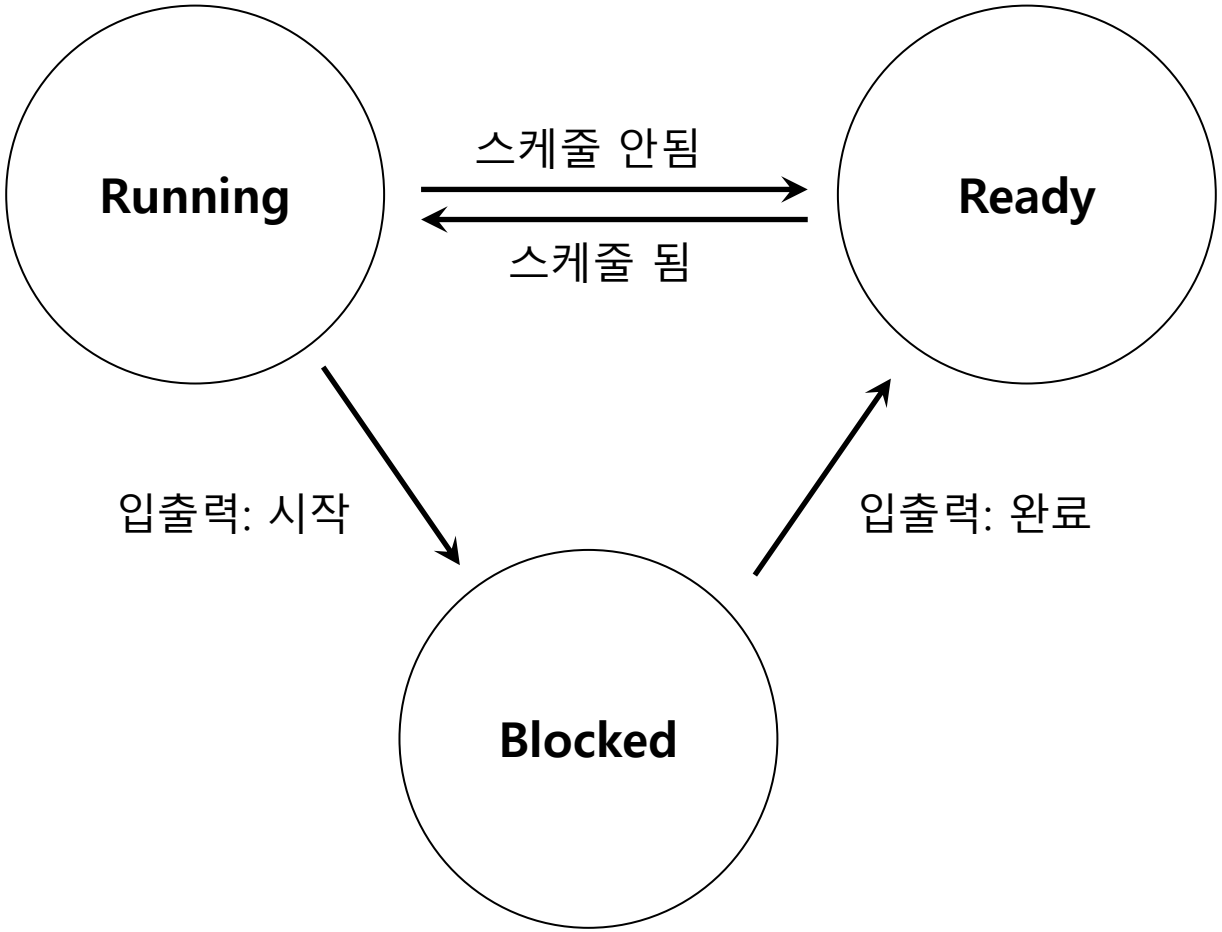


Loading:
디스크에 있는
프로그램을 꺼내서
프로세스의 주소공간에
복사한다.

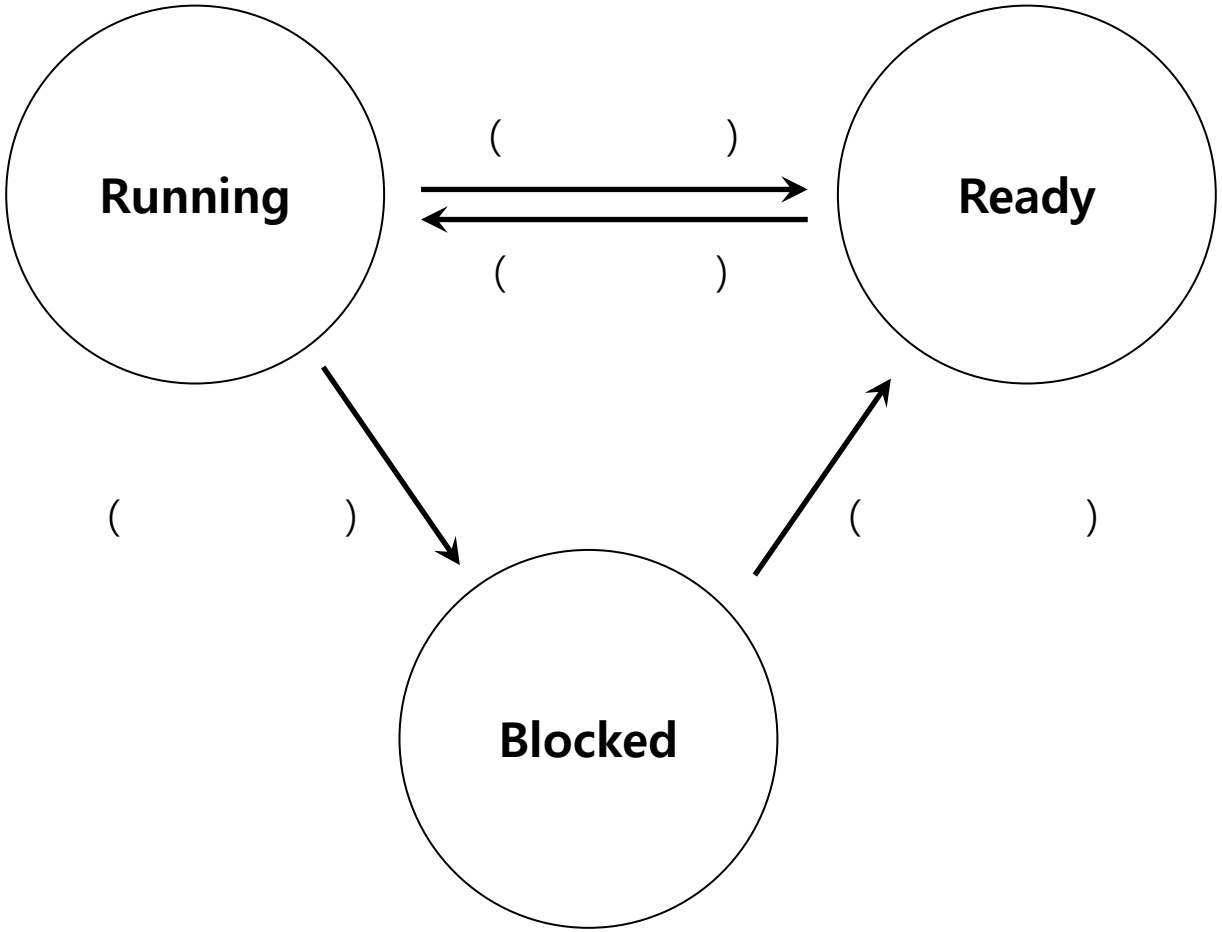
프로세스의 상태

- 프로세스는 상태를 갖는다
  - 실행(Running)
    - CPU가 실행 중
  - 준비(Ready, 실행 대기)
    - 실행할 준비가 다 되어 있음. CPU 사용 차례를 기다리고 있음.
  - 대기(Blocked)
    - 다른 사건이 발생하기를 기다리고 있음
    - 사건 : 입출력 완료, 메시지 도착, 약속 시간

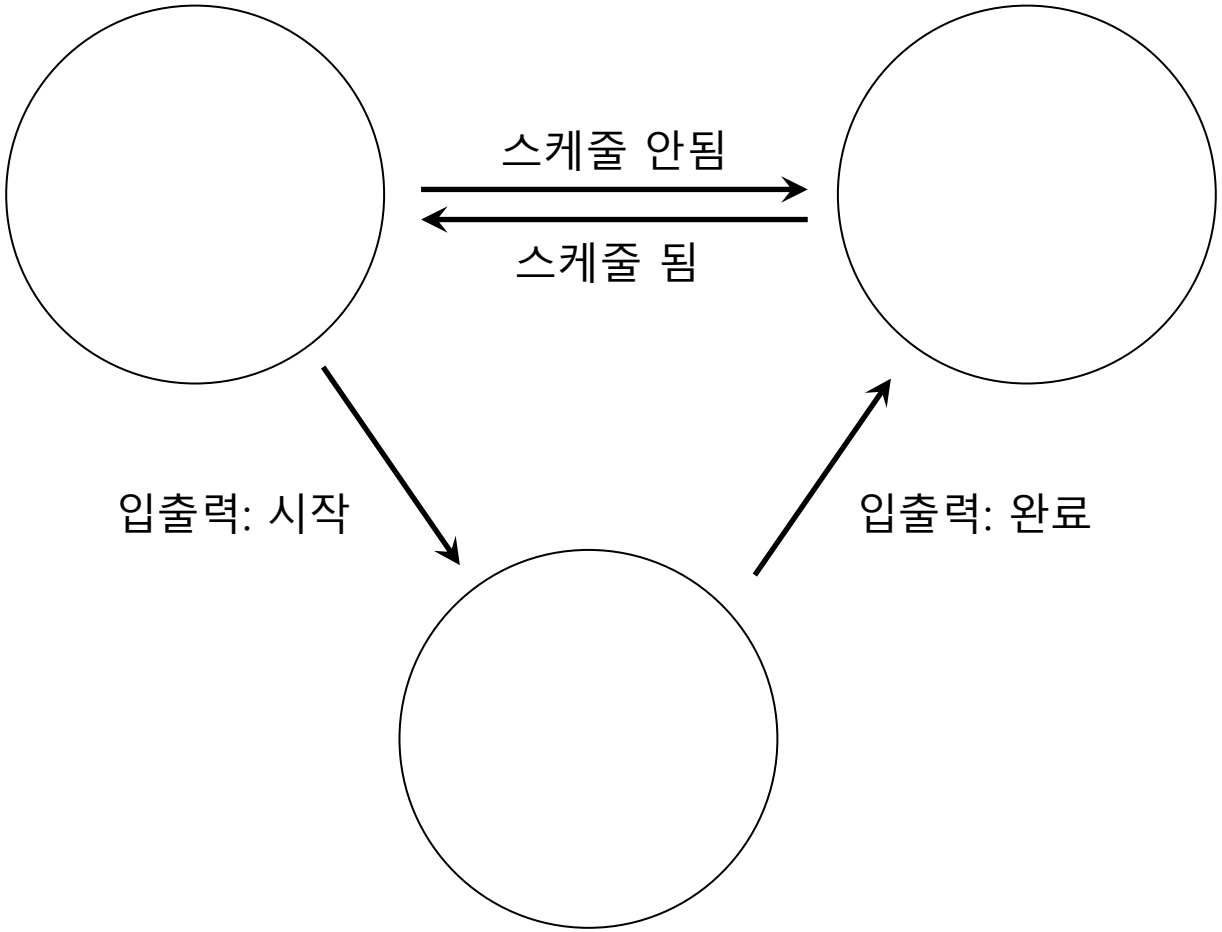
프로세스의 상태



프로세스의 상태



프로세스의 상태



프로세스의 상태

- CPU(=core)가 한 개일 때 상태 변환

시간	프로세스 0	프로세스 1	비고
0	실행	준비	
1	실행	준비	
2	실행	준비	프로세스 0 입출력 시작
3	대기	실행	
4	대기	실행	
5	대기	실행	프로세스 0 입출력 종료
6	준비	실행	
7	준비	실행	프로세스 1 종료
8	실행	-	
9	실행	-	

자료구조

- 운영체제가 관리하는 핵심 자료구조 들
  - **Process 리스트**
    - Ready processes
    - Blocked processes
    - Current running processes
  - **Register 문맥(context)**
    - **현재 프로세스의 실행 상태를 정의.**

- PCB(Process Control Block)
  - A C-structure that contains information about each process.

예) The xv6 kernel Proc Structure (1/2)

```
// 프로세스를 중단하고 이후에 재개하기 위해
// xv6가 저장하고 복원하는 레지스터
struct context {
    int eip;    // Instruction pointer register
    int esp;    // Stack pointer register
    int ebx;    // Called the base register
    int ecx;    // Called the counter register
    int edx;    // Called the data register
    int esi;    // Source index register
    int edi;    // Destination index register
    int ebp;    // Stack base pointer register
};

// 가능한 프로세스 상태
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

예) The xv6 kernel Proc Structure (1/2) 64비트

```
// 프로세스를 중단하고 이후에 재개하기 위해
// xv6가 저장하고 복원하는 레지스터
struct context {
        long long rip; // Instruction pointer register
        long long rax; // Accumulator register
        long long rsp; // Stack pointer register
        long long rbx; // Called the base register
        long long rcx; // Called the counter register
        long long rdx; // Called the data register
        long long rsi; // Source index register
        long long rdi; // Destination index register
        long long rbp; // Stack base pointer register
        long long r8, r9, r10, r11, r12, r13, r14, r15;
};

// 가능한 프로세스 상태
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

예) The xv6 kernel Proc Structure (2/2)

```
// 레지스터 문맥과 상태를 포함하여
// 각 프로세스에 대하여 xv6가 추적하는 정보
struct proc {
    char *mem;                  // Start of process memory
    uint sz;                    // Size of process memory
    char *kstack;               // Bottom of kernel stack
                                // for this process
    enum proc_state state;      // Process state
    int pid;                    // Process ID
    struct proc *parent;        // Parent process
    void *chan;                 // If non-zero, sleeping on chan
    int killed;                 // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;          // Current directory
    struct context context;     // Switch here to run process
    struct trapframe *tf;       // Trap frame for the
                                // current interrupt
};
```

5. 막간: 프로세스 API

현실의 프로세스

- 프로세스는 누가 만드는가?
  - 운영체제가 만든다.
- 왜 프로세스를 만드는가?
  - 어떤 프로세스가 요청했기 때문에
- 최초의 프로세스는 누가 만드는가?
  - 그건 운영체제가 알아서 만든다.
- 최초의 프로세스는 무엇인가?
  - Windows : winload.exe
  - Linux : init

- 어떻게 프로세스 생성을 요청하는가?
  - 시스템 호출을 한다.
  - Windows : CreateProcess
  - Linux : fork, exec 또는 clone

The fork() System Call

- 생략
- 너무나도 낡은 방식. 지금은 posix_spawn() 사용
- Pipe와 Redirection을 쉽게 하기위한 메커니즘
  - 옛날 text 시절에 많이 사용
  - 지금도 사용하면 편함.

Fork해서 프로세스를 생성. 하지만 기존의 Parent Process 코드와 데이터를 계속 실행

자신의 stdin, stdout, stderr을 파일로 교체 (open 시스템 호출 사용)

변경된 file들을 가지고 새 프로그램으로 교체 (exec 시스템 호출 사용)

CreateProcess

- Windows에서 프로세스를 생성하는 시스템 호출

```
BOOL CreateProcessA(
  [in, optional]          LPCSTR                      lpApplicationName,
  [in, out, optional]     LPSTR                       lpCommandLine,
  [in, optional]          LPSECURITY_ATTRIBUTES       lpProcessAttributes,
  [in, optional]          LPSECURITY_ATTRIBUTES       lpThreadAttributes,
  [in]                    BOOL                        bInheritHandles,
  [in]                    DWORD                       dwCreationFlags,
  [in, optional]          LPVOID                      lpEnvironment,
  [in, optional]          LPCSTR                      lpCurrentDirectory,
  [in]                    LPSTARTUPINFOA              lpStartupInfo,
  [out]                   LPPROCESS_INFORMATION       lpProcessInformation
);
```

- Windows에서 프로세스의 종료를 기다리는 시스템 호출

```
DWORD WaitForSingleObject(
  [in] HANDLE hHandle,
  [in] DWORD  dwMilliseconds
);
```

숙제 #2

- 실행하면 노트패드(메모장, notepad.exe)를 2개 실행시키는 프로그램을 작성하라.
  - CreateProcess함수 사용
  - 노트패드가 둘 다 종료할 때 까지 기다렸다가 종료.
  - Windows에서 Visual Studio를 사용
- eclass로 제출

6. Mechanism: 제한적 직접 실행 원리

CPU를 어떻게 효율적으로 사용할까?

- 운영체제는 실제 CPU를 시분할(time sharing) 방식으로 나누어 사용한다.
- 고려 요소
  - **성능** : 어떻게 지나친 오버헤드 없이 가상화를 구현할 것인가?
  - **제어** : 어떻게 여러가지 상황에서 CPU에 대한 통제를 유지하면서 프로세스를 실행할 것인가?

직접 실행

- Just run the program directly on the CPU.

OS	Program
1. 프로세스 목록의 항목을 생성 2. 프로그램 메모리 할당 3. 메모리에 프로그램 로딩 4. 스택에 `argc` / `argv` 넣기 5. 레지스터 내용 초기화 6. `main()`으로 점프	
	7. `main()` 실행 8. `main()`에서 `return`
9. 프로세스 메모리 반환 10. 프로세스 목록에서 제거	

Program은 *제한* 없이 뭐든지 할 수 있다.
OS는 프로그램 실행을 그저 바라보기만 해야 하고, "일반 라이브러리" 처럼 호출 당해야 한다.

문제점 1: 제한된 연산

- 만일 프로세스가 다음과 같은, 하면 안되는 동작을 한다면…
  - DISK에 임의의 위치에 I/O 명령을 내리기
  - CPU나 메모리를 직접 조작해서 더 많은 자원을 얻어내기.
- **해결책**: 사용자 모드의 도입
  - 사용자 모드(User mode): 하드웨어 자원에 대한 접근을 제한한다. 모든 프로세스에 적용
  - 커널 모드(Kernel mode): 기계의 모든 자원을 조작할 수 있음. 운영체제가 사용.

시스템 콜

- 사용자 모드의 문제점 해결
  - I/O는 어떻게 할 것인가? (어떻게 안전하게 Kernel mode로 바꿀 것인가?)
  - 커널을 함부로 조작하는 것을 어떻게 막을 것인가?
- **주의 깊게 열어 놓은** 특정 기능만 수행할 수 있는 핵심 통로.
  - 예) 프로세스 생성/제거, 타른 프로세스와의 통신, 메모리 추가 할당

시스템 콜(계속)

- **Trap(x86의 SYSCALL)** 명령어
  - 커널로 이동
    - 트랩 테이블(trap table)에 정해진 주소로만 이동할 수 있음.
  - 권한 레벨을 커널 모드로 상승시킴
- **Return-from-trap (x86 IRET)** 명령어
  - 호출한 프로그램으로 되돌아 가기
  - 권한 레벨을 다시 유저 모드로 낮춤

제한된 직접 수행 프로토콜

OS @ boot (kernel mode)	Hardware
trap table에 적절한 값을 기록한다.	메모리에 trap table이 저장됨

OS @ run (kernel mode)	Hardware	Program (user mode)
프로세스 목록에 항목 추가 프로그램을 위한 메모리 할당 프로그램을 메모리에 로딩 argv를 스텍에 기록 레지스터와 PC의 값을 커널 스택에 기록 **return-from -trap**	커널 스택에서 레지스터값 로드 (사용자 모드 전환, main으로 이동)	main() 실행 … **trap** 으로 OS 시스템 호출

제한된 직접 수행 프로토콜 (계속.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(계속)	
	레지스터 값을 커널 스택에 저장 커널 모드로 전환 trap handeler로 jump	
트랩 처리 시스템 호출 요청 처리 **return-from-trap**		
	커널 스택에서 레지스터값 로드 (사용자 모드 전환, 호출한 명령 다음으로 이동)	
		... main()에서 리턴 trap 운영체제 호출 (`exit`)
프로세스 메모리 반환 프로세스 목록에서 제거		

문제점 2: 프로세스간 전환

- 운영체제는 어떻게 CPU를 **다시 획득**하여 프로세스를 전환할 수 있는가?
  - 협조(cooperative) 방식: **시스템 호출을 하면 리턴할 때 프로세스 전환**
  - 비협조(Non-Cooperative) 방식: **OS가 제어권 확보**

협조 방식 : 시스템 콜 대기

- 프로세드들이 자주 시스템 콜을 해서 CPU를 양보한다. (`yield`나 `Sleep`).
  - 운영체제가 이어서 실행할 프로세스 선택
  - 시스템 콜이 아니더라도 오류가 발생하면 OS로 이동.
    - Divide by zero
    - 접근 불가 메모리 억세스
  - Ex) 초기 매킨토시 OS, 구버전의 Xerox Alto system

프로세스가 무한루프에 빠지면.
→ 컴퓨터를 껏다 켠다.

비협조 방식: 운영체제 전권행사

- 어떻게 강제로 운영체제를 호출할까?
- 타이머 인터럽트로 구현
  - 부팅 시 OS가 타이머를 켬.
  - 타이머는 몇 백분의 1초마다 인터럽트를 발생시킴
  - 인터럽트가 발생하면 :
    - 시스템호출 할 때와 똑같은 일이 발생함.
    - HW가 자동적으로
      - 레지스터 값을 저장하고, 커널모드로 바꾼 후
      - OS가 등록해 놓은 주소로 이동

타이머 인터럽트는 일정한 시간 간격으로 OS가 다시 CPU를 차지하도록 해준다.

Context 저장과 복원

- 스케줄러 가 결정한다:
  - **지금 프로세스**를 계속 실행할지, **다른 프로스세스**를 실행할 지.
  - 다른 프로세스로 옮긴 다면 운영체제는 문맥 교환(context switch)을 한다.

문맥 교환(Context Switch)

- 내부 구현
  - 현재 프로세스에서 사용 중인 **레지스터**를 PCB에 저장
  - 실행할 프로세스의 PCB에서 **레지스터**를 로드
  - **실행할 프로세스의 커널 stack**으로 변경
  - return-from-trap

Limited Direction Execution Protocol (Timer interrupt)

OS @ boot (kernel mode)	Hardware
trap table에 적절한 값을 기록한다	아래의 주소값들이 저장된다 syscall handler timer handler
interrupt timer 시작	타이머 HW가 동작 시작 Xms 마다 CPU에 인터럽트

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A …
	timer interrupt 레지스터들(A)을 k-stack(A)에 저장 커널모드로 전환 timer handler로 이동	

Limited Direction Execution Protocol (Timer interrupt)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	

timer handler 실행
switch() 기능 호출
  레지스터들(A)을 to PCB(A)에 저장
  PCB(B)에서 레지스터들(B)을 로딩
  k-stack(B)로 스택을 전환
return-from-trap (B 에게)		

	k-stack(B)에서 레지스터들(B) 복구 유저 모드로 전환 B의 PC위치로 점프	
		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
```

숙제 #3

- 운영체제 호출에 걸리는 시간 측정
- 강제 문맥 교환 코드

```
#include <thread>

std::this_thread::yield();
```

```
#include <stdio.h>

printf("Hello World.\n");
```

- 시간 측정
  - 위의 두개의 프로그램을 따로 측정
    - 루프를 돌면서 100번 측정을 한 후 평균값을 구하라.
  - std::chrono::high_resolution_clock 을 사용해서 nano second단위로 측정한다
- 제출
  - e-class
  - 측정한 컴퓨터의 CPU 모델을 같이 제출 (예: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz 3.40 GHz)