Processes

Operating System (OS)

안인규

강의자료는 KAIST 원유집 교수님 OS 강의 자료를 참고했습니다.

Recap

OS is in charge of making sure the system operates correctly and efficiently

가상화 동시성 영속성

Virtualization		Concurrency	Persistence
3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*
4 Processes	13 Address Spaces code	26 Concurrency and Threads code	36 I/O Devices
5 Process API code	14 Memory API	27 Thread API code	37 Hard Disk Drives
6 Direct Execution	15 Address Translation	28 Locks code	38 Redundant Disk Arrays (RAID)
7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories
8 Multi-level Feedback	17 Free Space Management	30 Condition Variables code	40 File System Implementation
9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores code	41 Fast File System (FFS)
10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling
11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)
	21 Swapping: Mechanisms	34 *Summary*	44 Flash-based SSDs
	22 Swapping: Policies		45 Data Integrity and Protection
	23 Complete VM Systems		46 *Summary*
	24 *Summary*		47 *Dialogue*
			48 Distributed Systems
			49 Network File System (NFS)
			50 Andrew File System (AFS)
			51 *Summary*

Recap | Virtualization (CPU)

```
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <sys/time.h>
4       #include <assert.h>
5       #include "common.h"
6
7       int
8       main(int argc, char *argv[])
9       {
10              if (argc != 2) {
11                      fprintf(stderr,
                        "usage: cpu <string>\n");
12                      exit(1);
13              }
14              char *str = argv[1];
15              while (1) {
16                      Spin(1);
17                      printf("%s\n", str);
18              }
19              return 0;
20      }
```

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A ...
```

Virtualizing the CPU!

→ The illusion that the system has a very large number of virtual CPUS

How to provide the illusion of many CPUs?

- CPU virtualizing
  - OS는 여러 개의 가상 CPU가 존재하는 것처럼 보이도록 할 수 있다.
  - **Time sharing**: 하나의 Process를 실행한 후, 중지하고, 다른 프로세스를 실행
    - 잠재적인 비용은 성능 저하
    - 즉, 각 Process는 느려짐

A Process

Process? → Process는 실행 중인 프로그램이다

- Comprising of a process:
  - Memory (address space)
    - Instructions (Code Segment, 프로그램 명령어)
    - Data section (전역/정적 변수)
    - Stack (지역 변수, 함수 호출 정보)
    - Heap (동적 메모리 할당 영역)
  - Registers
    - Program counter (PC)
    - Stack pointer (SP)
  - File Descriptors (Process가 사용하는 파일, 소켓, 입출력 장치 등)

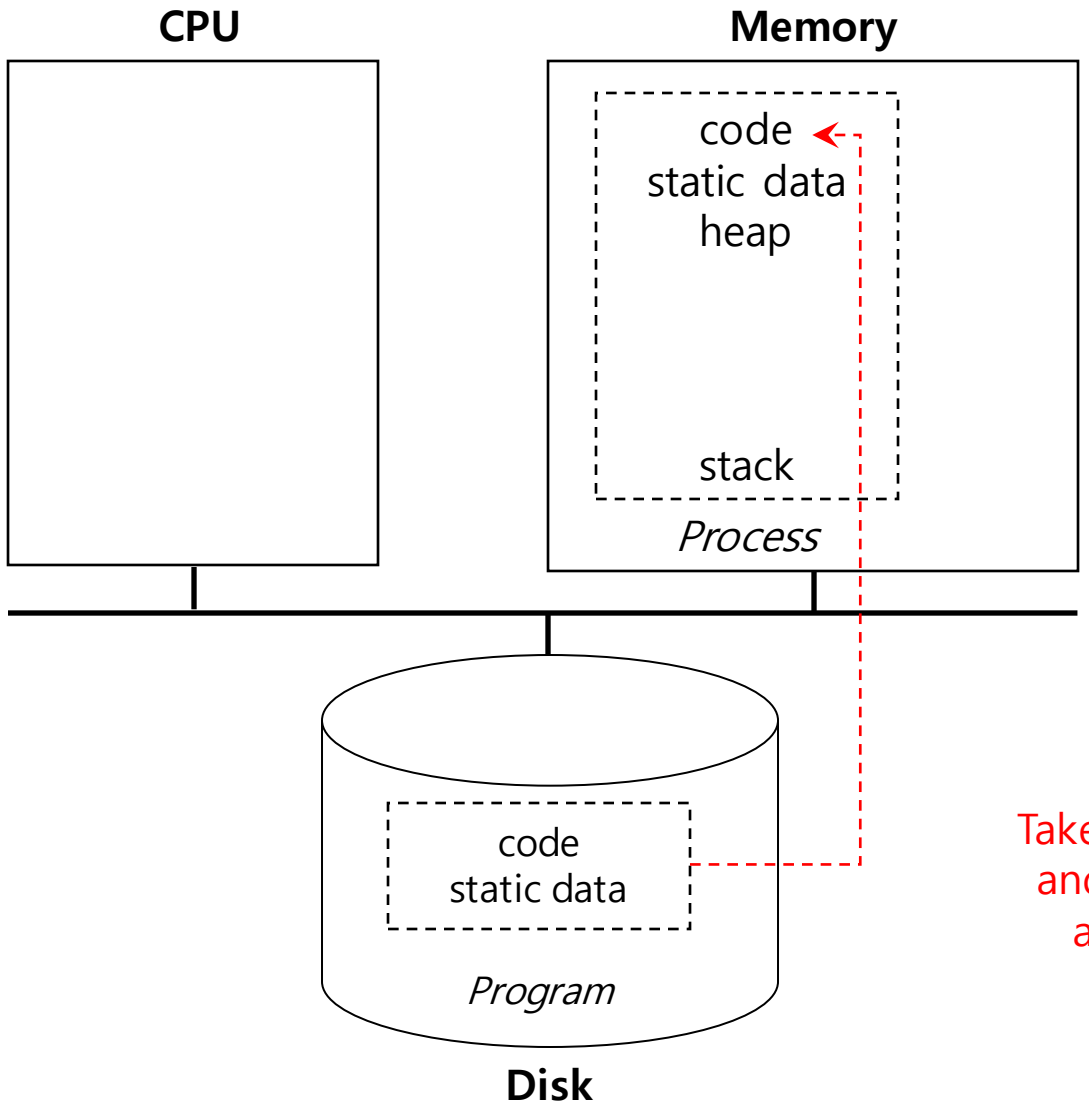
Process API

- 아래와 같은 API는 모든 최신 OS에서 사용할 수 있다.
  - **Create (생성)**
    - 새로운 프로세스를 생성하여 프로그램을 실행
  - **Destroy (제거)**
    - 제어할 수 없는 프로세스를 중지
  - **Wait (대기)**
    - 프로세스가 실행을 멈출 때까지 대기
  - **Miscellaneous Control (기타 제어)**
    - 프로세스를 일시 중단한 후 다시 재개하는 방법 제공
  - **Status (상태)**
    - 프로세스에 대한 상태 정보를 가져옴

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

Loading: From Program To Process



Loading:
Takes on-disk program and reads it into the address space of process

Process Creation

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()`.

Process Creation

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.

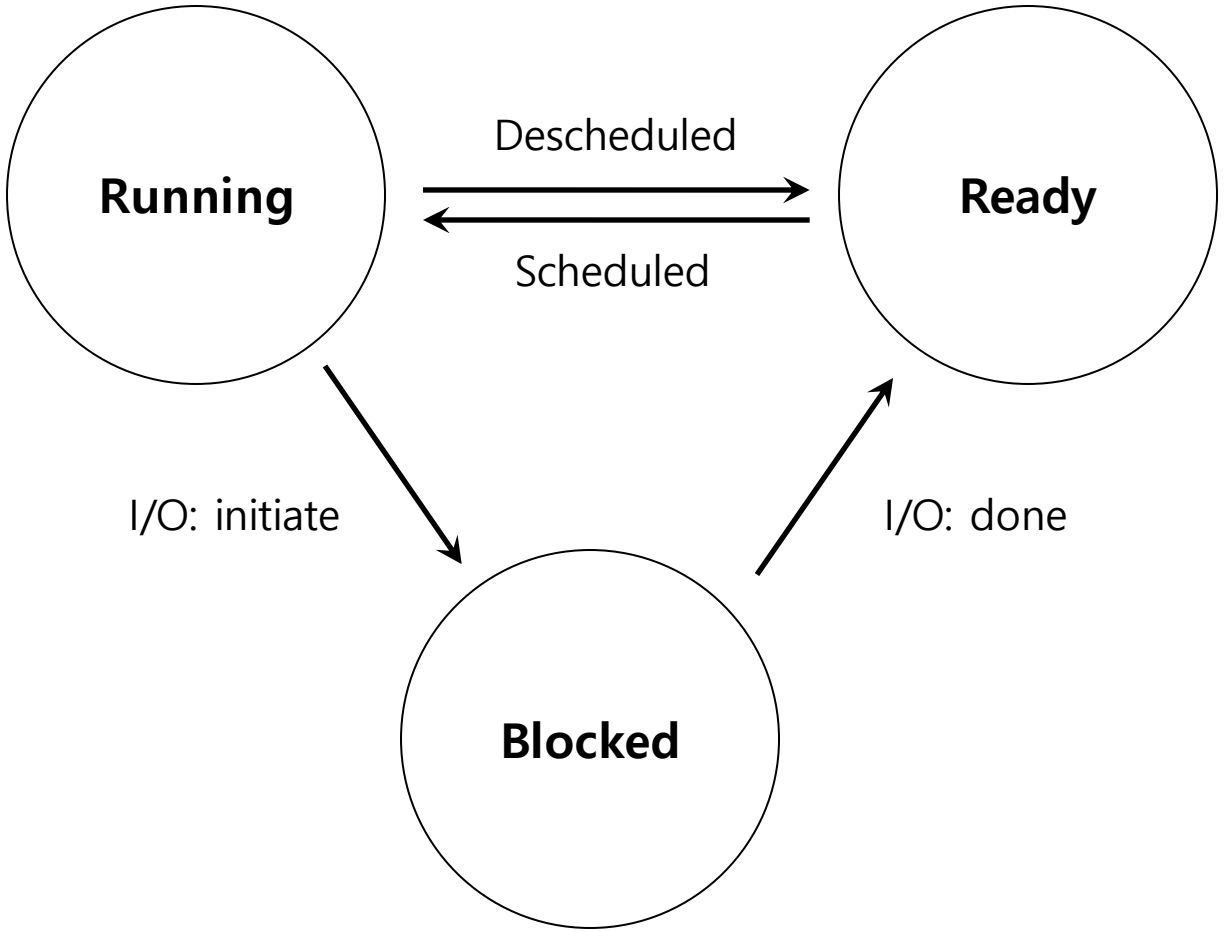
??? → Limited Direct Execution

Process States

- Process가 실행되는 동안 다양한 상태(States)를 가질 수 있음

- A process can be one of three states.
  - **Running**
    - A process is running on a processor.
  - **Ready**
    - A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
  - **Blocked**
    - A process has performed some kind of operation.
    - When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process State Transition



Process State Transition

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

Data structures

- PCB(Process Control Block)
  - A C-structure that contains information about each process.
  - **Register context:** a set of registers that define the state of a process

- **Process list**
  - Ready processes
  - Blocked processes
  - Current running process

Example) The xv6 kernel Proc Structure (Cont.)

```
// the information xv6 tracks about each process
// including its register context and state
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
};
```

Example) Register Context in xv6

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;   // Index 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
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```