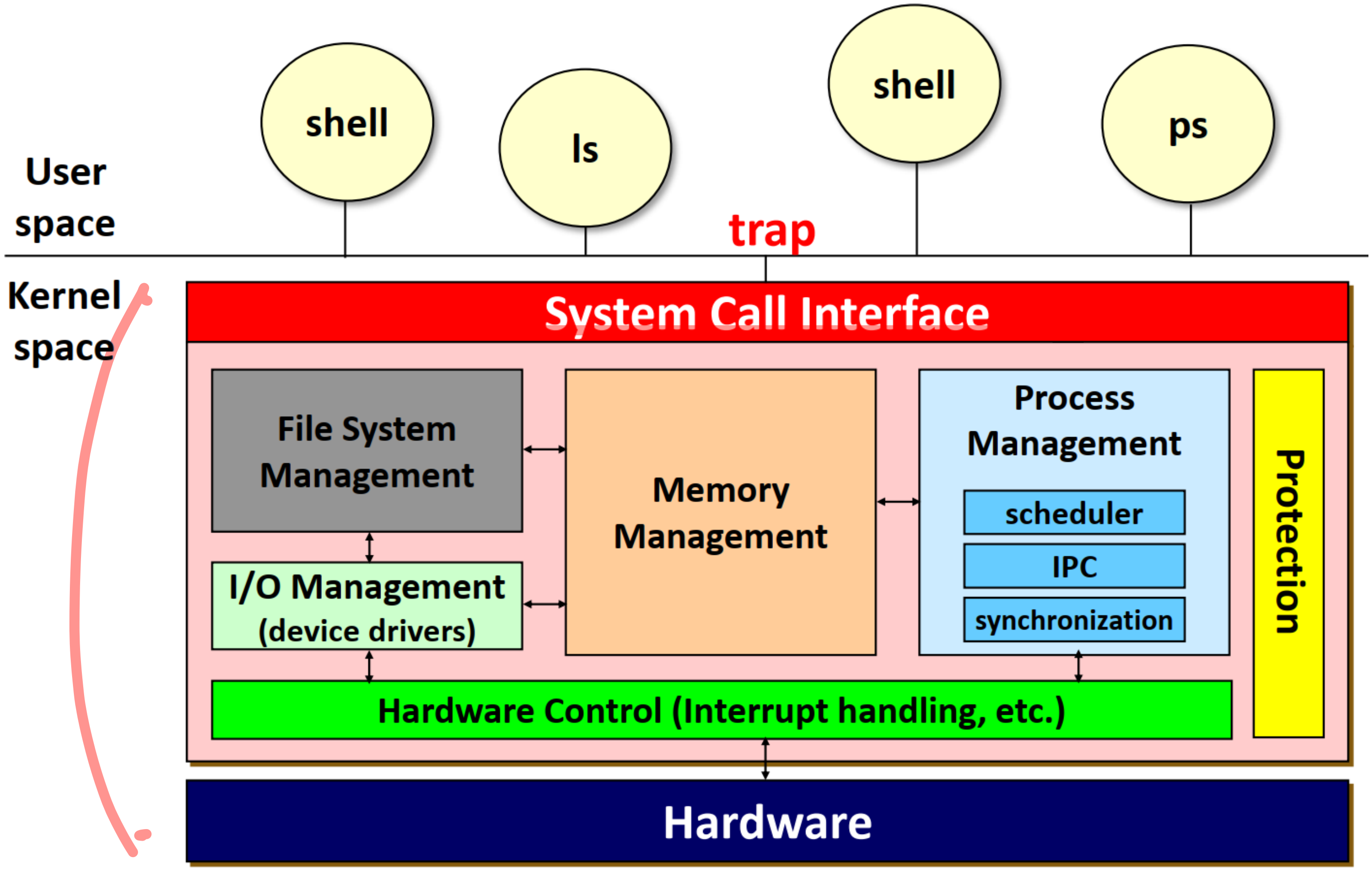
The Abstraction: The Process

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

OS Internals

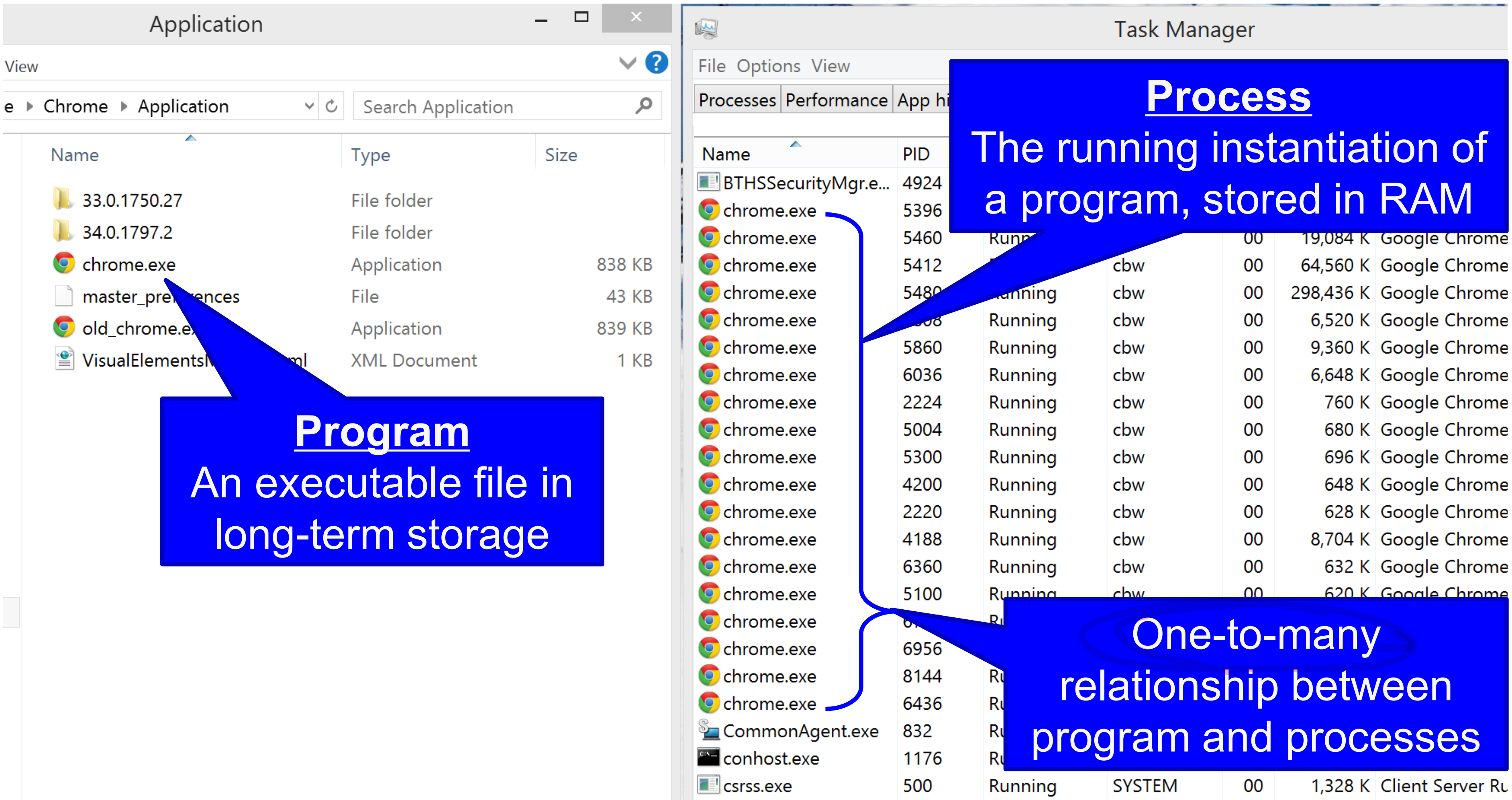


Courtesy of Prof. Jin-Soo Kim @ SNU

How to Provide the Illusion of Many CPUs?

- **The definition of a process, informally, is a running program**
  - The program itself is a lifeless thing; just sitting there on the disk, a bunch of instructions with some data
  - The OS gets them running, transforming it into something useful
  - We often want to run more than one program at once
- **The OS creates the illusion by virtualizing the CPU**
  - The OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU by time sharing of the CPU
  - Time sharing is implemented by running one process, then stopping it and running another, and so forth, and its potential cost is performance
- **To implement the virtualization, the OS will need both**
  - some low-level machinery: mechanisms; how to do something? (context switching)
  - some high-level intelligence: policies; what should be done? (scheduling)

Program vs Process



The Abstraction: A Process and API

- **The abstraction provided by OS of a running program is process**

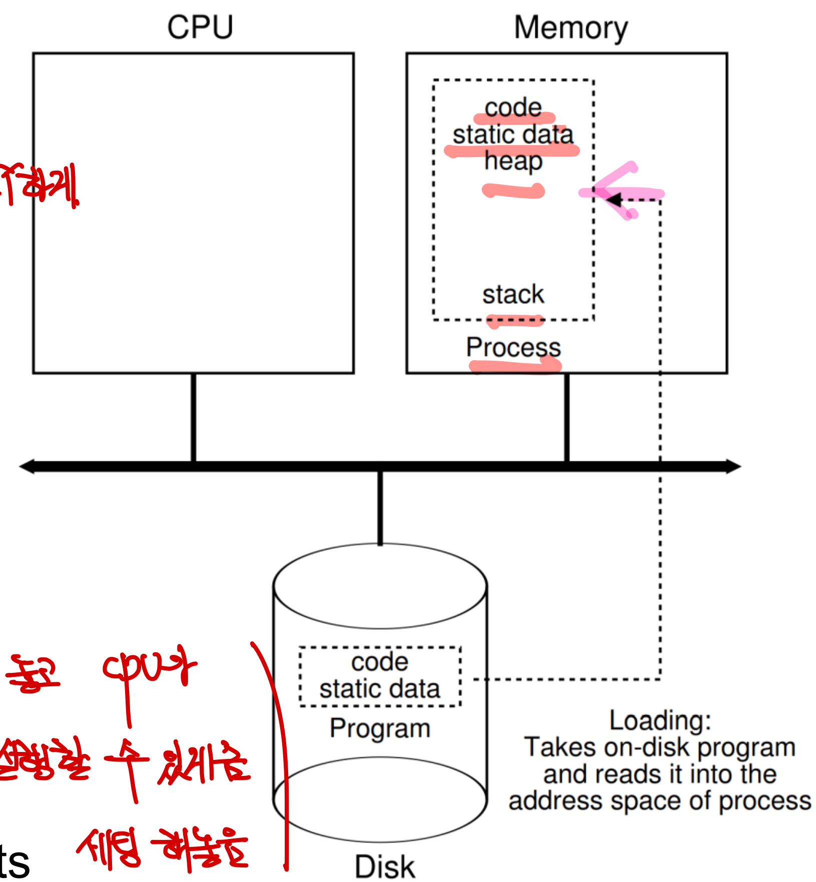
- **To understand what constitutes a process, we have to know its machine state**
  - The machine state that comprises a process are:
    1) Memory: instructions and data lie in memory (address space)
    2) Registers: many instructions explicitly read or update registers, such as program counter, stack pointer, and frame pointer
    3) Storage: I/O information that the process is currently accessing (e.g. file)

- **Modern OS has an interface by providing APIs:**
  - Create: create a new process to run a program
  - Destroy: halt a runaway process (interface to destroy processes forcefully)
  - Wait: wait for a process to stop running
  - Miscellaneous control: other controls (e.g. suspend a process and resume it)
  - Status: get some status information about a process

Process Creation: A Little More Detail (1)

1) The first step is to **load** its code and any static data into memory, into the address space of the process

- Programs initially reside on disk in an executable format → 한 번에 다올림. 프로그램은
- Early OSes load eagerly all at once before running the program → 현재는: LAZY하게
- Modern OSes perform the load process lazily, by loading pieces of code or data only as they are needed during program execution (swapping, paging)



2) Some memory is allocated for the program's **run-time stack**

할당 해 놓고 cpu가 실행할 수 있게끔 세팅 해놓음

- The stack for local variable, function parameters, and return addresses
- The OS initializes the stack with arguments (i.e. `argc` and `argv` of `main()`)

Process Creation: A Little More Detail (2)

3) The OS allocates some memory for the program's heap

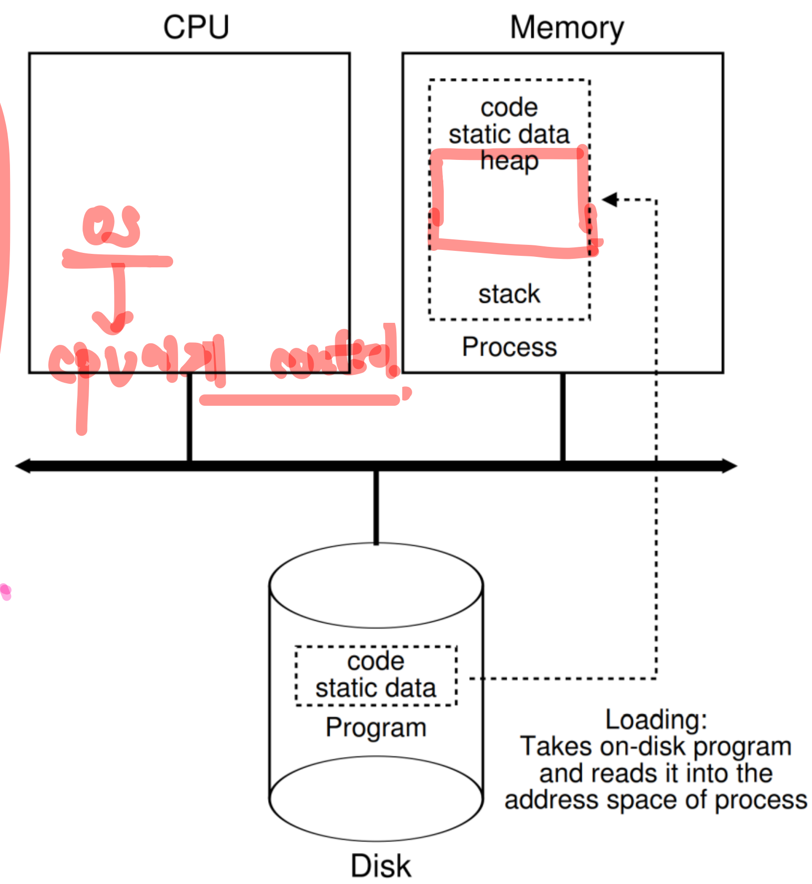
- The heap for explicitly requested dynamically-allocated data (e.g. `malloc()`)

4) The OS does some other initialization tasks, related to I/O

- e.g.) each process in UNIX systems has three open file descriptors (standard input, output, error)

5) The OS starts the program running at the entry point, `main()`

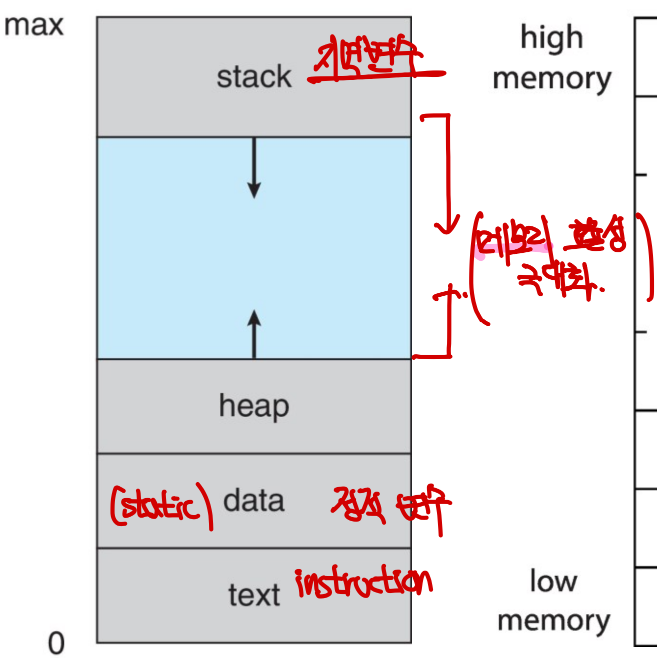
- Then, the OS transfers control of the CPU to the newly-created process
- Thus, the program begins its execution

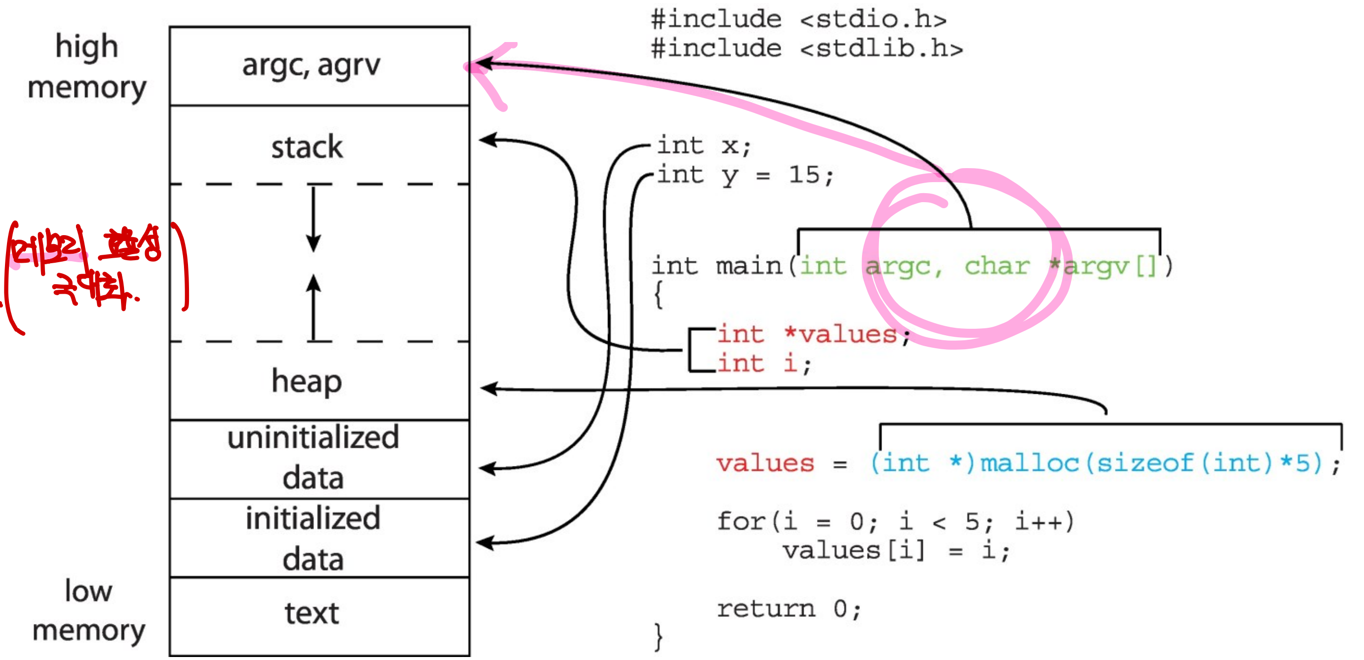


Memory Layout of a C Program

실행 프로그램 → 인스턴스화.

- **The memory layout of a process is typically divided into:**
  - 1) text (executable code), 2) data (global variable),
    3) heap (dynamically allocated memory), 4) stack (temporary data storage)
  - The data section is divided into (a) initialized data and (b) uninitialized data
  - A separate section is provided for the **argc** and **argv** parameters
- **Remember that each process has its own private address space**

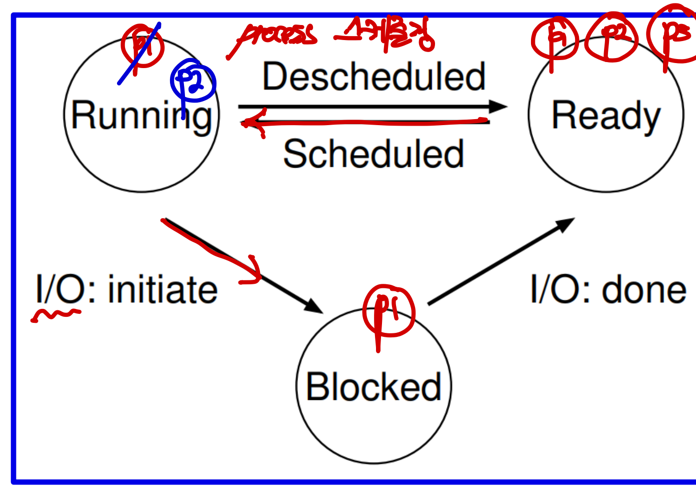




Process States

실행 상태 프로세스 수 = CPU의 개수

- In a simplified view, a process can be in one of three states:
  1) **Running**: a process is running on a CPU (executing instructions)
  2) **Ready**: a process is ready to run, but the OS chosen not to run at this moment
  3) **Blocked**: a process has performed some kind of operation that makes it not ready to run until some other event takes place (e.g. when a process requests an I/O)



- Examples

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

Tracing process states (CPU only)

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

Tracing process states (CPU and I/O)

Data Structures

- **The OS has some key data structures that track various relevant pieces of information:**
  - Process list that includes all of the ready, blocked, and running processes
  - Register context to hold the register contents for a stopped process, which will be used for context switching (save them when stop → restore them when resume)
- **Each process is represented by a process control block (PCB)**
  - C-structure that contains all the information about a process

Machine state를 (최소 3개를 저장) (Memory/register/storage) + α

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

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

다 운영체제 마다 다름

xv6's register context and process states

```
// 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 !zero, sleeping on chan
  int killed;                 // If !zero, has 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
};
```

레지스터의 값

운영체제

xv6's PCB (process descriptor)