Lecture Note 2: Processes

March 11, 2024

Jongmoo Choi

Dept. of Software
Dankook University
http://embedded.dankook.ac.kr/~choijm

Contents

- From Chap 3~6 of the OSTEP
- Chap 3. A Dialogue on Virtualization
- Chap 4. The abstraction: The Process
  - ✓ Process, Process API, Process State and Data Structure
- Chap 5. Interlude: Process API
  - ✓ System calls: fork(), wait(), exec(), kill() , …
- Chap 6. Mechanism: Limited Direct Execution
  - ✓ Basic Technique: Limited Direct Execution
  - ✓ Switch between Modes
  - ✓ Switch between Processes

Chap 3. A Dialogue on Virtualization

- Virtualization

Student: *But what is virtualization, oh noble professor?*

Professor: *Imagine we have a peach.*

Student: *A peach? (incredulous)*

Professor: *Yes, a peach. Let us call that the **physical** peach. But we have many eaters who would like to eat this peach. What we would like to present to each eater is their own peach, so that they can be happy. We call the peach we give eaters **virtual** peaches; we somehow create many of these virtual peaches out of the one physical peach. And the important thing: in this illusion, it looks to each eater like they have a physical peach, but in reality they don't.*

Student: *So you are sharing the peach, but you don't even know it?*

Professor: *Right! Exactly.*

Student: *But there's only one peach.*

Professor: *Yes. And...?*

Student: *Well, if I was sharing a peach with somebody else, I think I would notice.*

Professor: *Ah yes! Good point. But that is the thing with many eaters; most of the time they are napping or doing something else, and thus, you can snatch that peach away and give it to someone else for a while. And thus we create the illusion of many virtual peaches, one peach for each person!*

Student: *Sounds like a bad campaign slogan. You are talking about computers, right Professor?*

Professor: *Ah, young grasshopper, you wish to have a more concrete example. Good idea! Let us take the most basic of resources, the CPU. Assume there is one physical CPU in a system (though now there are often two or four or more). What virtualization does is take that single CPU and make it look like many virtual CPUs to the applications running on the system. Thus, while each application*

Chap 4. The Abstraction: The Process

- Process definition
  - ✓ A program in execution
  - ✓ Scheduling entity (CPU), has its memory (DRAM)
    - c.f.) program: a lifeless thing, sit on the disk and waiting to spring into action
  - ✓ There exist multiple processes (e.g. ppt, browser, word, player, …)
    - Each process has its own memory (address space), virtual CPU, state, …

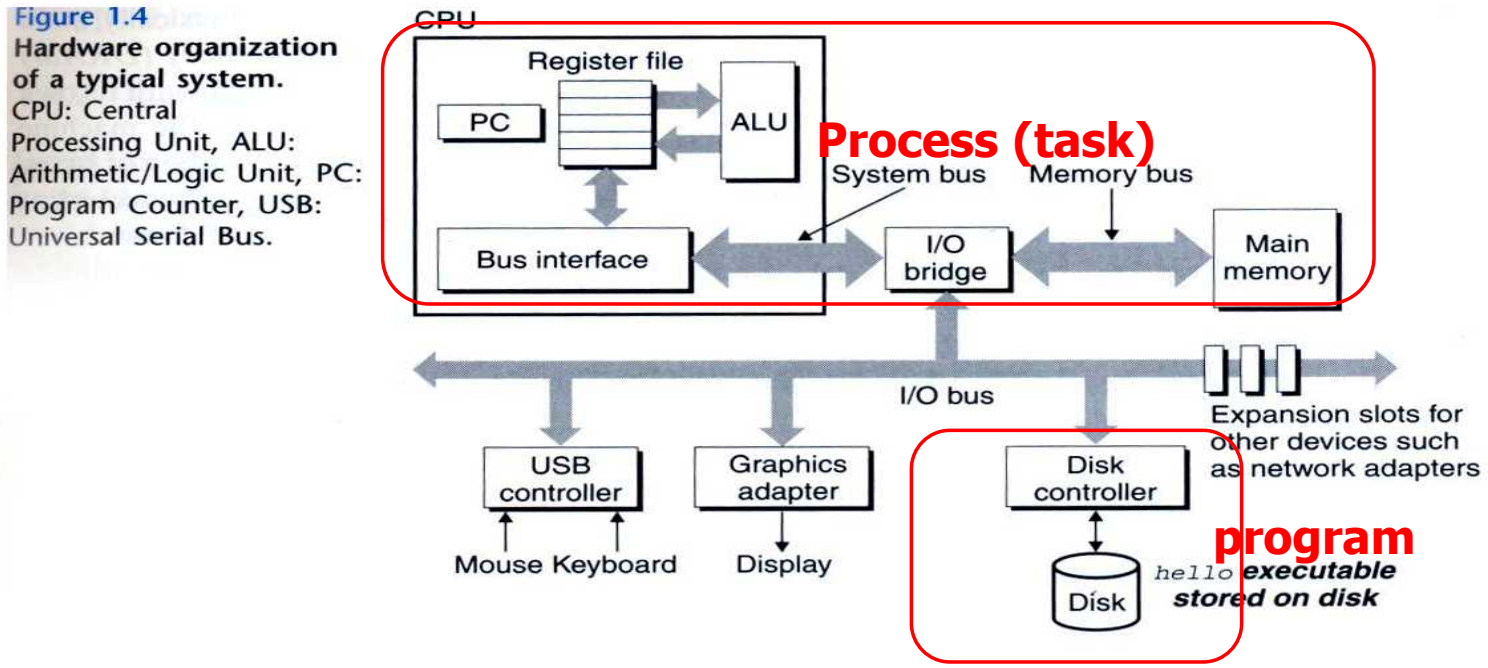
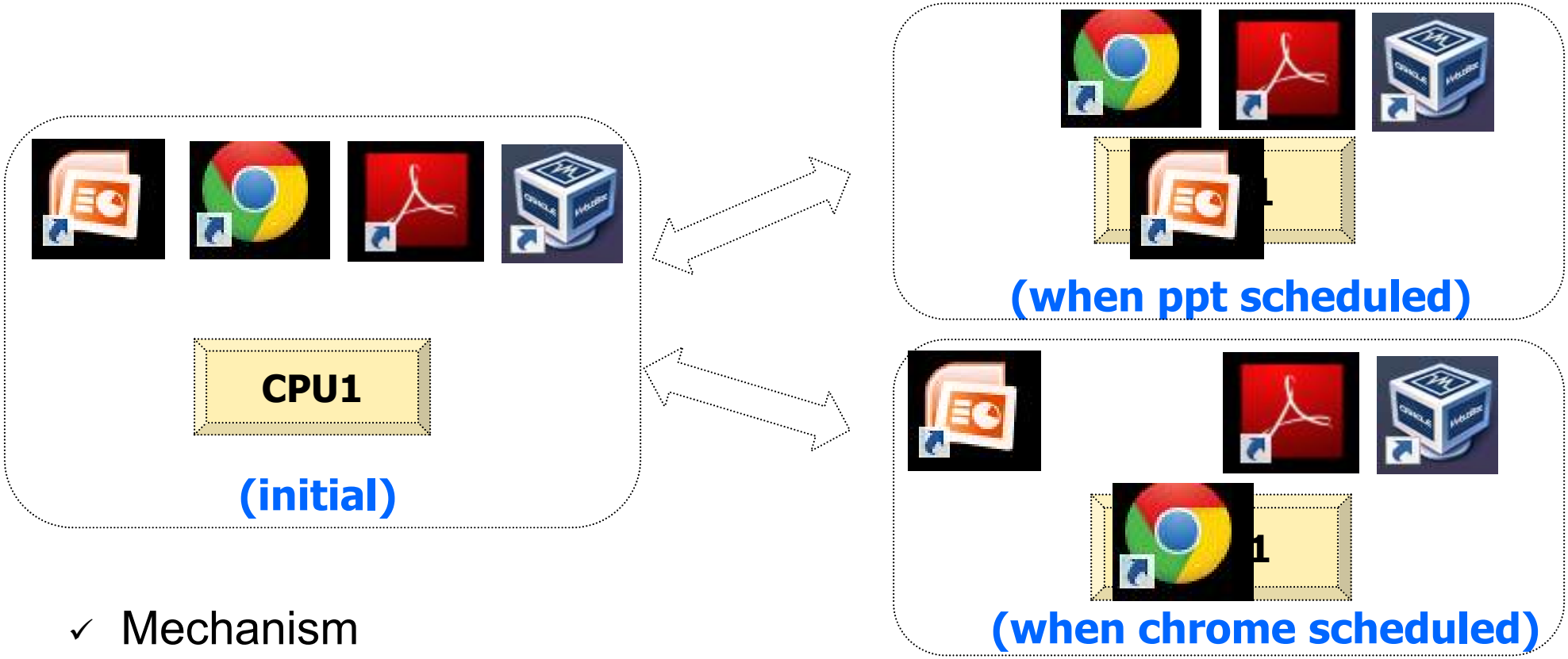


Figure 1.4 Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program Counter, USB: Universal Serial Bus.

(Source: computer systems: a programmer perspective)

Chap 4. The Abstraction: The Process

- How to virtualize CPU? Time sharing system

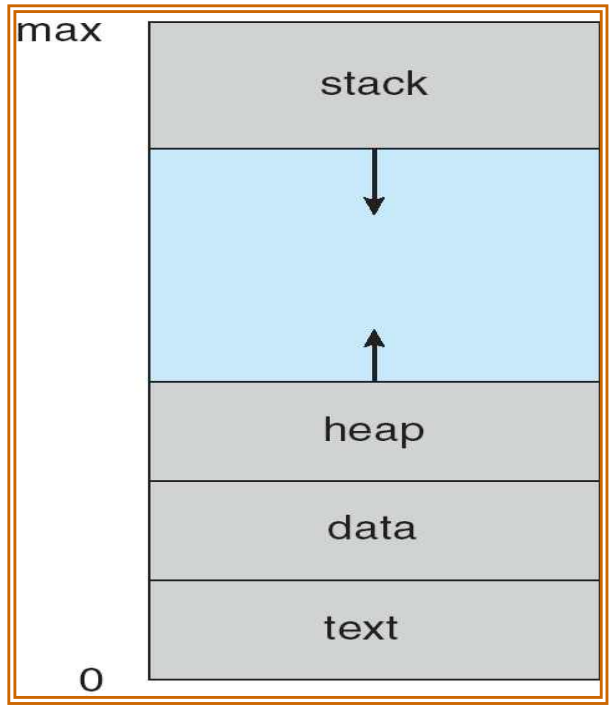


- ✓ Mechanism
  - context switch: an ability to stop running one program and start running another on a given CPU
- ✓ Policy
  - scheduling policy: based on historical information or workload knowledge or performance metric.

☛ Time sharing vs. Space sharing

4.1 Process

- Process structure
  - ✓ Need resources to run:
    - CPU
      - Registers such as PC, SP, ..
    - Memory (address space)
      - Text: program codes
      - Data: global variables
      - Stack: local variables, parameters, …
      - Heap: allocated dynamically
    - I/O information
      - Opened files (including devices)



(Source: A. Silberschatz, "Operating system Concept")

  - ✓ Program vs. Process
    - Program: passive entity, a file containing instructions stored on disk (executable file or binary)
    - Process: active entity, having CPU and memory, doing I/Os
    - Execute a program twice ➔ result in creating two processes (from one program) ➔ text is equivalent while others (data, stack) vary (1-to-n)

4.2 Process API

- Basic APIs for a process

  - **Create:** An operating system must include some method to create new processes. When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program you have indicated.
  - **Destroy:** As there is an interface for process creation, systems also provide an interface to destroy processes forcefully. Of course, many processes will run and just exit by themselves when complete; when they don't, however, the user may wish to kill them, and thus an interface to halt a runaway process is quite useful.
  - **Wait:** Sometimes it is useful to wait for a process to stop running; thus some kind of waiting interface is often provided.
  - **Miscellaneous Control:** Other than killing or waiting for a process, there are sometimes other controls that are possible. For example, most operating systems provide some kind of method to suspend a process (stop it from running for a while) and then resume it (continue it running).
  - **Status:** There are usually interfaces to get some status information about a process as well, such as how long it has run for, or what state it is in.

☛ Related system calls are discussed in the chapter 5 of OSTEP

4.3 Process Execution: A Little More Detail

- How to start a program
  - ✓ Load
    - Bring code and static data into the address space
    - Based on executable format (e.g. ELF, PE, BSD, …)
    - Eagerly vs. Lazily (paging, swapping)
  - ✓ Dynamic allocation
    - Stack
    - Initialize parameters (argc, argv)
    - Heap if necessary
  - ✓ Initialization
    - file descriptors (0, 1, 2)
    - I/O or signal related structure
  - ✓ Jump to the entry point: main()

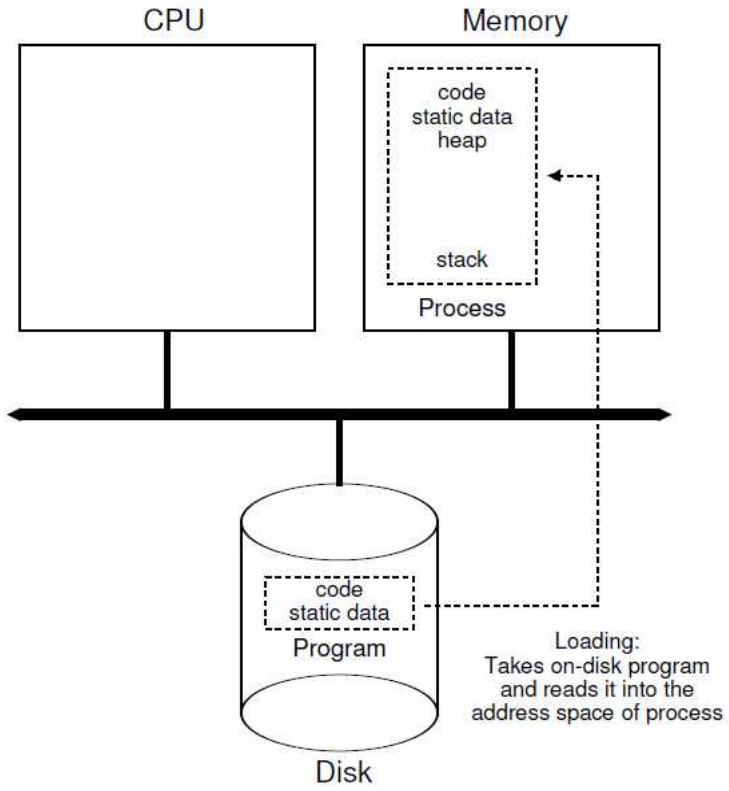
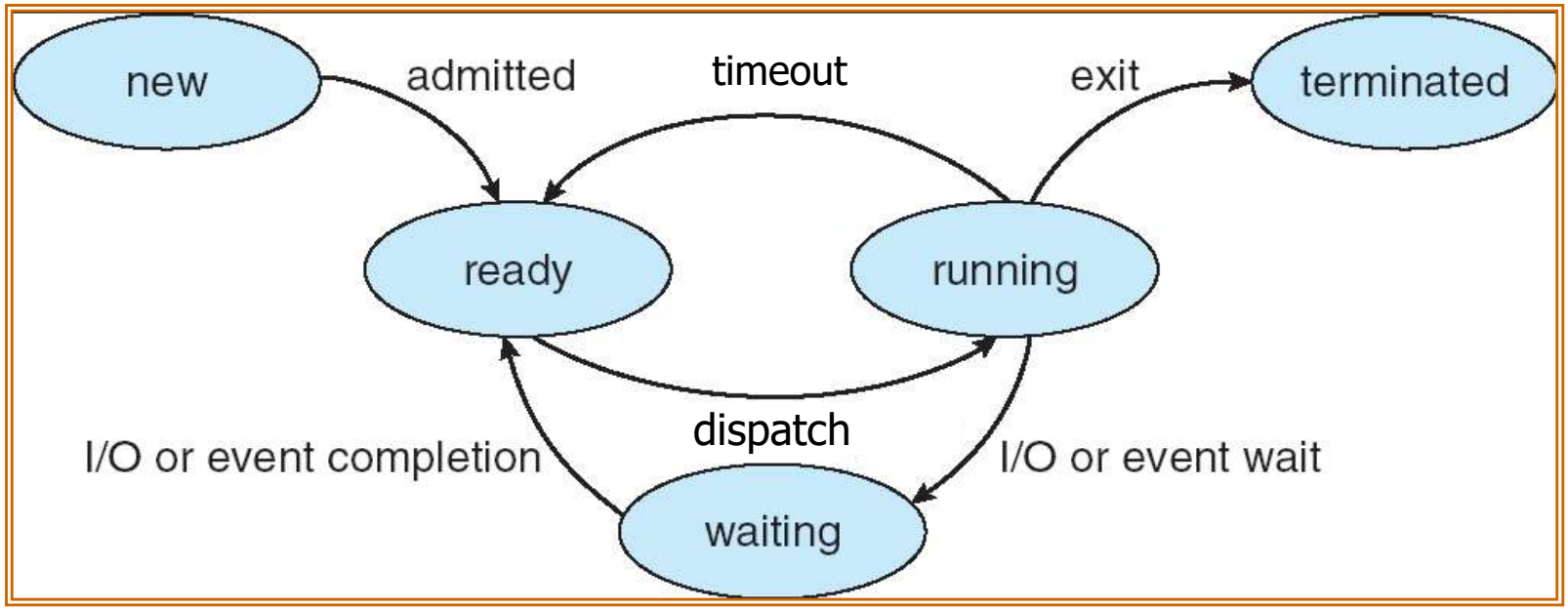


Figure 4.1: **Loading: From Program To Process**

4.4 Process States

- State and transition

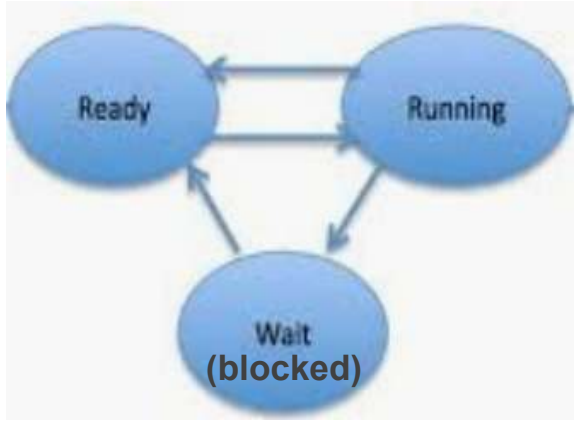


(Source: A. Silberschatz, "Operating system Concept")

- ✓ State
  - ready, running, waiting(blocked) + new(created, embryo), terminated (zombie)
- ✓ Transition
  - dispatch (schedule), timeout (preemptive, descheduled), wait (sleep, I/O initiate), wakeup (I/O done) + admitted, exit
  - suspend and resume: to Disk (swap) or to RAM

4.4 Process States

- Example
  - ✓ Used resources: CPU only ➔ Figure 4.3
  - ✓ Used resources: CPU and I/O ➔ Figure 4.4
    - Note: I/O usually takes quite longer than CPU



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

Figure 4.3: **Tracing Process State: CPU Only**

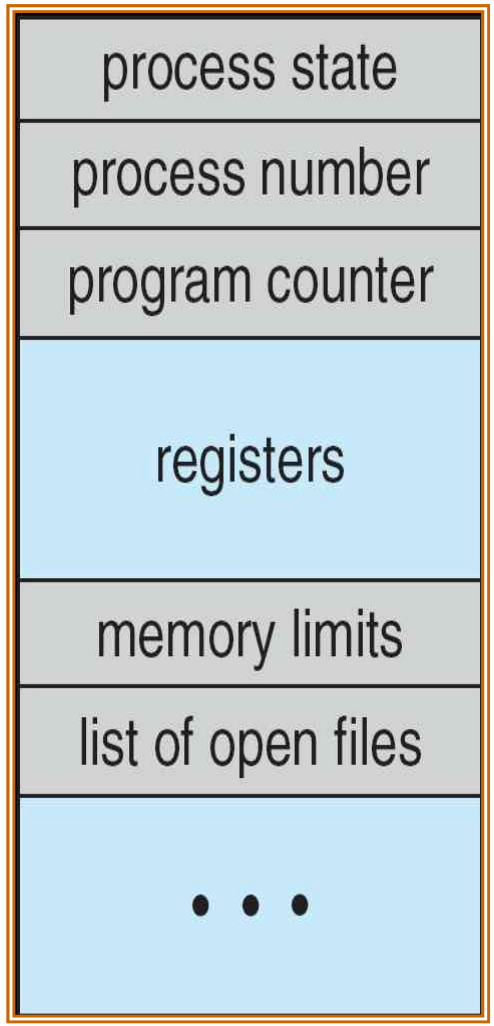
Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$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

Figure 4.4: **Tracing Process State: CPU and I/O**

☛ At the end of time 6 in Figure 4.4, OS can decide to 1) continue running the process1 or 2) switch back to process 0. Which one is better? Discuss tradeoff.

4.5 Data Structure

- PCB (Process Control Block)
  - ✓ Information associated with each process
    - Process state
    - Process ID (pid)
    - Program counter, CPU registers
      - Used during context switch
      - Architecture dependent
    - CPU scheduling information
    - Memory-management information
    - Opened files
    - I/O status information
    - Accounting information
  - ✓ Managed in the kernel's data segment



(Source: A. Silberschatz, "Operating system Concept")

4.5 Data Structure

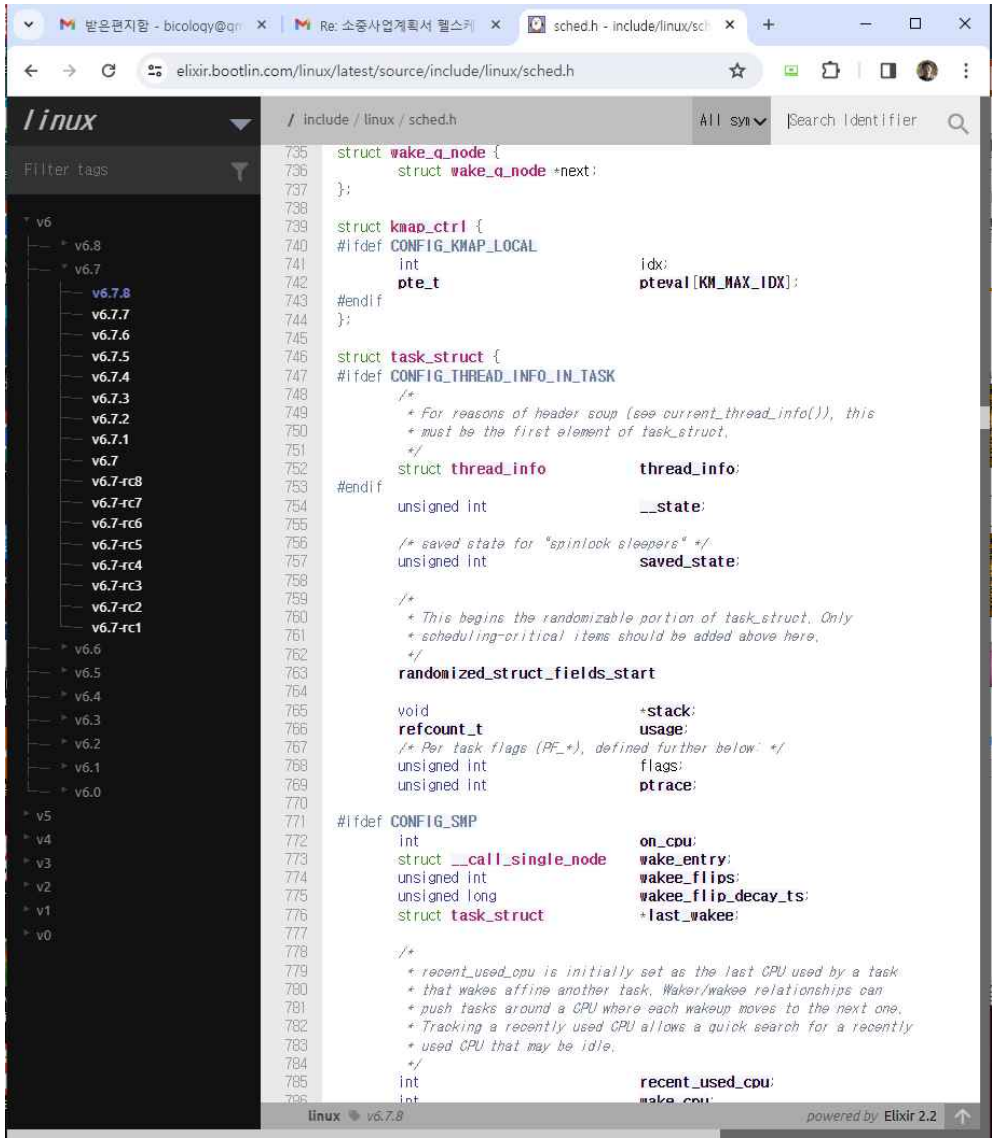
- PCB Implementation example in OSTEP
  - ✓ OS is a program, implementing a process using data structure (e.g. struct proc and struct context)
  - ✓ All "proc" structures are manipulated using a list

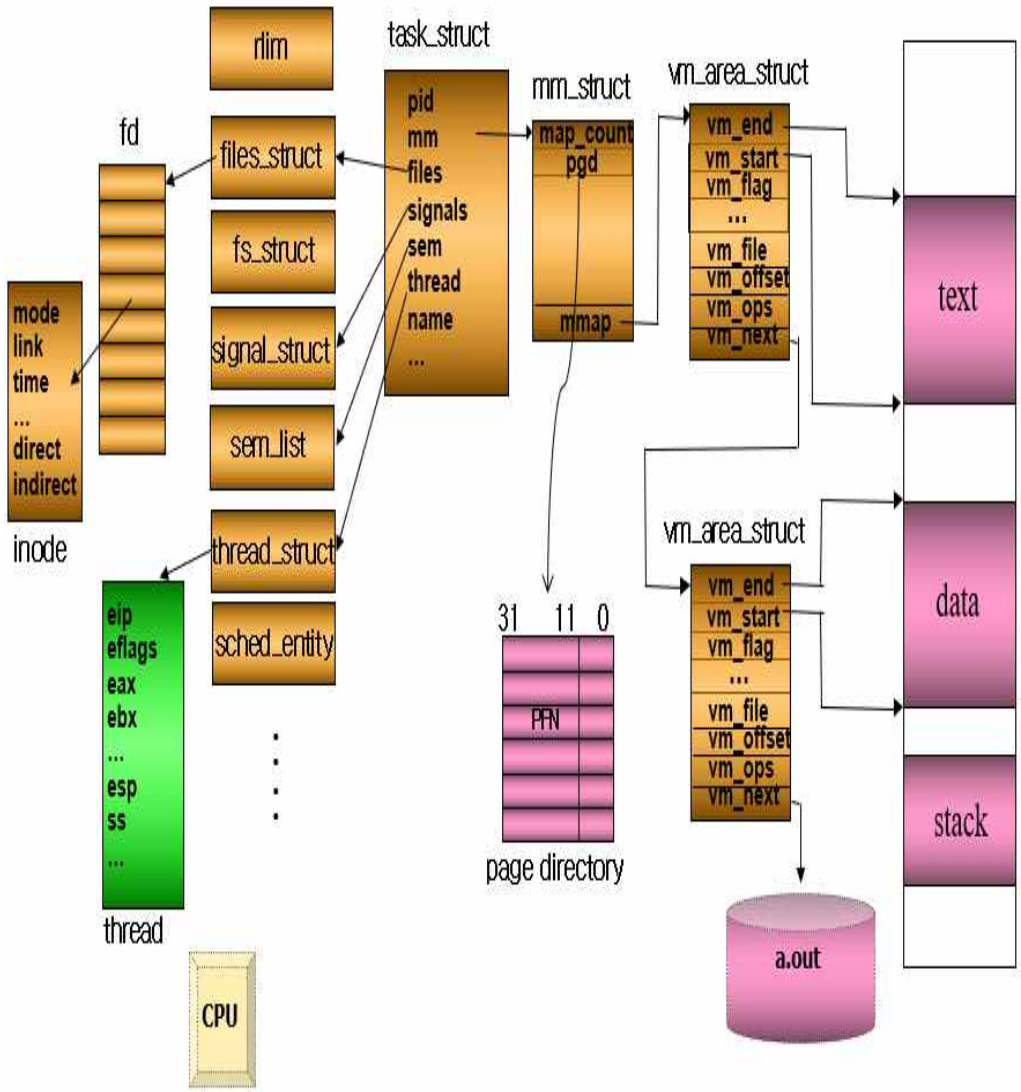
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

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

// 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
};
```

Figure 4.5: **The xv6 Proc Structure**

4.5 Data Structure (Optional)

- PCB in real OS (task structure in Linux)





<https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h >

(Source: 리눅스 커널 내부구조)

Chap 5. Interlude: Process API

- Comments for Interlude by Remzi

> ASIDE: INTERLUDES
> Interludes will cover more practical aspects of systems, including a particular focus on operating system APIs and how to use them. If you don't like practical things, you could skip these interludes. But you should like practical things, because, well, they are generally useful in real life; companies, for example, don't usually hire you for your non-practical skills.

5.1 fork() system call

- fork()
  - ✓ Create a new process: parent, child
  - ✓ Return two values: one for parent (>0) and the other for child (0)
  - ✓ Non-determinism: not decide which one run first.


```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[]) {
6    printf("hello (pid:%d)\n", (int) getpid());
7    int rc = fork();
8    if (rc < 0) {
9      // fork failed
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) {
13     // child (new process)
14     printf("child (pid:%d)\n", (int) getpid());
15   } else {
16     // parent goes down this path (main)
17     printf("parent of %d (pid:%d)\n",
18            rc, (int) getpid());
19   }
20   return 0;
21 }
22
```

Figure 5.1: Calling fork() (p1.c)

5.2 wait() system call

- wait()
  - ✓ Block a calling process until one of its children finishes
  - ✓ Now, deterministic ➔ synchronization


```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(int argc, char *argv[]) {
7    printf("hello (pid:%d)\n", (int) getpid());
8    int rc = fork();
9    if (rc < 0) {           // fork failed; exit
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) { // child (new process)
13     printf("child (pid:%d)\n", (int) getpid());
14   } else {                // parent goes down this path
15     int rc_wait = wait(NULL);
16     printf("parent of %d (rc_wait:%d) (pid:%d)\n",
17            rc, rc_wait, (int) getpid());
18   }
19   return 0;
20 }
21
```

Figure 5.2: **Calling fork() And wait() (p2.c)**

5.3 exec() system call

- exec()
  - ✓ Load and overwrite code and static data, re-initialize stack and heap, and execute it (never return) ➔ refer to 8 page
  - ✓ 6 variations: execl, execlp, execle, execv, execvp, execve

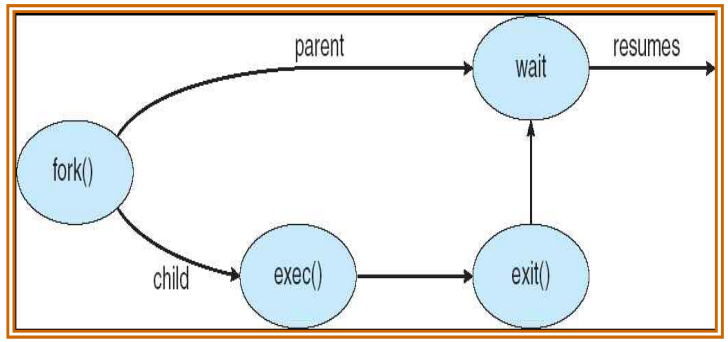

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char *argv[]) {
8     printf("hello (pid:%d)\n", (int) getpid());
9     int rc = fork();
10    if (rc < 0) {           // fork failed; exit
11       fprintf(stderr, "fork failed\n");
12       exit(1);
13    } else if (rc == 0) { // child (new process)
14       printf("child (pid:%d)\n", (int) getpid());
15       char *myargs[3];
16       myargs[0] = strdup("wc");     // program: "wc"
17       myargs[1] = strdup("p3.c"); // arg: input file
18       myargs[2] = NULL;             // mark end of array
19       execvp(myargs[0], myargs);   // runs word count
20       printf("this shouldn't print out");
21    } else {                  // parent goes down this path
22       int rc_wait = wait(NULL);
23       printf("parent of %d (rc_wait:%d) (pid:%d)\n",
24              rc, rc_wait, (int) getpid());
25    }
26    return 0;
27 }
28
```

Figure 5.3: **Calling fork(), wait(), And exec() (p3.c)**

☛ Comments from Remzi: Do it on a Linux system. "Type in the code and run it is better for understanding"

5.4 Why? Motivating the API (optional)

- Why separate fork() from exec()?
  - ✓ Modular approach of UNIX, support extensibility




```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <string.h>
5   #include <fcntl.h>
6   #include <sys/wait.h>
7
8   int main(int argc, char *argv[]) {
9     int rc = fork();
10    if (rc < 0) {
11      // fork failed
12      fprintf(stderr, "fork failed\n");
13      exit(1);
14    } else if (rc == 0) {
15      // child: redirect standard output to a file
16      close(STDOUT_FILENO);
17      open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC,
18           S_IRWXU);
19      // now exec "wc"...
20      char *myargs[3];
21      myargs[0] = strdup("wc");    // program: wc
22      myargs[1] = strdup("p4.c"); // arg: file to count
23      myargs[2] = NULL;            // mark end of array
24      execvp(myargs[0], myargs);   // runs word count
25    } else {
26      // parent goes down this path (main)
27      int rc_wait = wait(NULL);
28    }
29    return 0;
30  }
```

Figure 5.4: **All Of The Above With Redirection (p4 . c)**

5.5 Other parts of the API

- Other APIs
  - ✓ getpid(): get process id
  - ✓ kill(): send a signal to a process
  - ✓ signal(): register a signal catch function
  - ✓ scheduling related
  - ✓ …
- Command and tool
  - ✓ ps, top, perf, …
  - ✓ read the man pages for commands and tools

ASIDE: **RTFM** — READ THE MAN PAGES

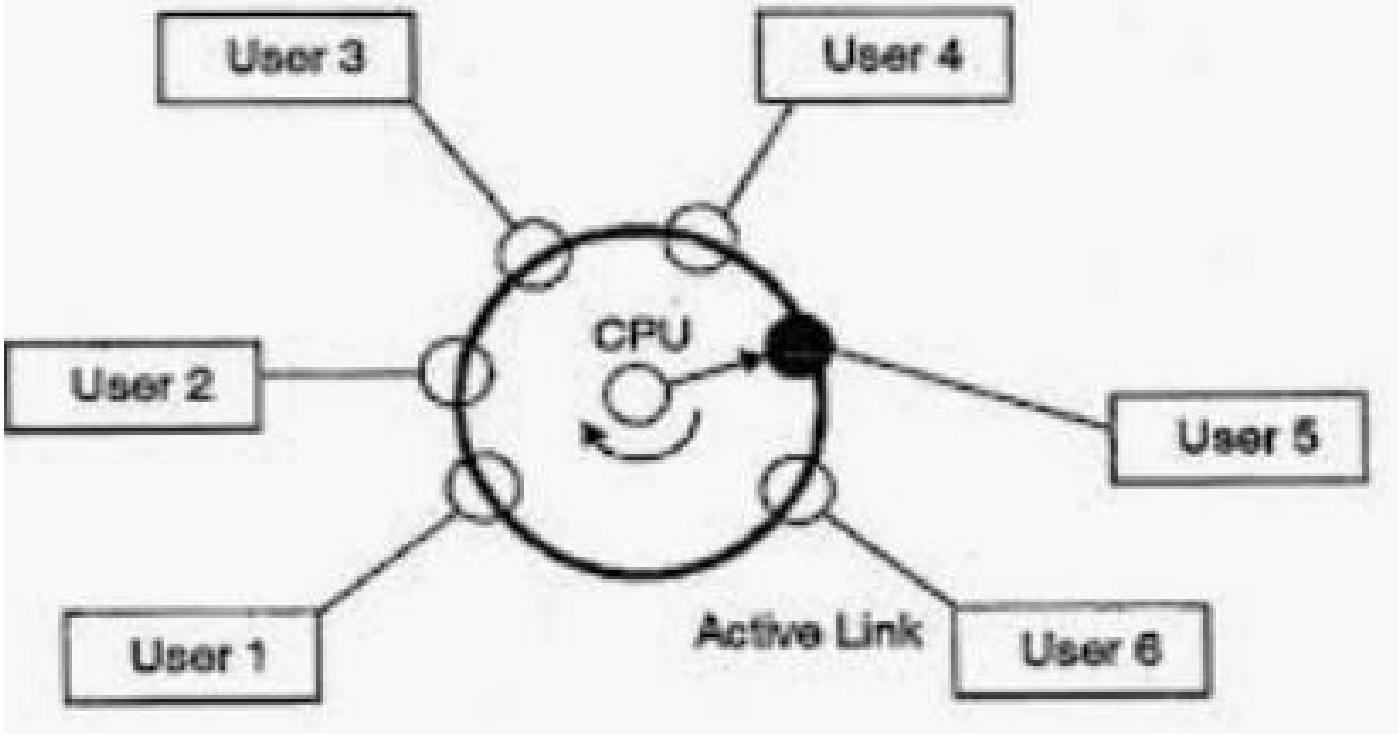
Many times in this book, when referring to a particular system call or library call, we'll tell you to read the **manual pages**, or **man pages** for short. Man pages are the original form of documentation that exist on UNIX systems; realize that they were created before the thing called **the web** existed.

Spending some time reading man pages is a key step in the growth of a systems programmer; there are tons of useful tidbits hidden in those pages. Some particularly useful pages to read are the man pages for whichever shell you are using (e.g., **tcsh**, or **bash**), and certainly for any system calls your program makes (in order to see what return values and error conditions exist).

Finally, reading the man pages can save you some embarrassment. When you ask colleagues about some intricacy of `fork()`, they may simply reply: "RTFM." This is your colleagues' way of gently urging you to Read The Man pages. The F in RTFM just adds a little color to the phrase...

Chap 6. Mechanism: Limited Direct Execution

- Time sharing
  - ✓ Key technique for virtualizing CPU
  - ✓ Issues
    - Performance: how to minimize the virtualization overhead?
    - Control: how to run processes while retaining control over the CPU?



(Source: Google image. Users can be replaced with programs or processes)

6.1 Basic Technique: Limited Direct Execution

- Performance-oriented ➔ Direct execution
  - ✓ Run the program directly on the CPU
  - ✓ Efficient but not controllable

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute call main()	
	Run main()
	Execute return from main
Free memory of process	
Remove from process list	

☛ See 8 page in LN2

Figure 6.1: Direct Execution Protocol (Without Limits)

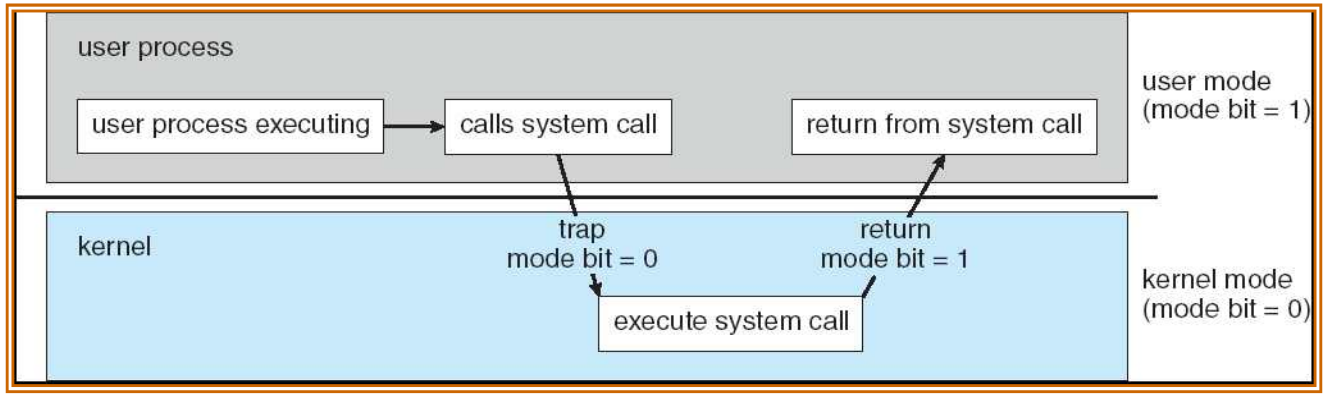
☛ **What is the problem of the above example?**

☛ **Control is particularly important to OS. Without control, a process could run forever, monopolizing resources.**

6.2 Problem #1: Restricted Operation

- Control mechanism 1: Restrict operations
  - ✓ Most operations can run directly (e.g. arithmetic, loop, branch, …)
  - ✓ Some operations that should run indirectly (privileged operations)
    - Gain more system resources such as CPU and memory
    - Issue an I/O request directly to a disk
  - ✓ Through a well defined APIs (system call)
    - E.g.) fork(), nice(), malloc(), open(), read(), write(), …

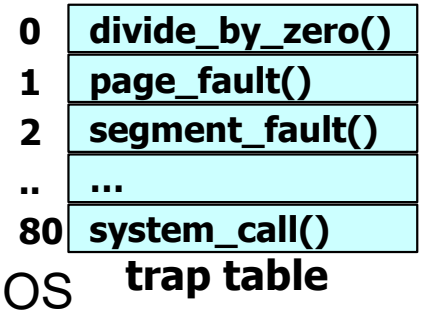
- How to: User mode vs. Kernel mode
  - ✓ User mode: do privileged operation ➔ cause exception and killed
  - ✓ Kernel mode: do privileged operation ➔ allowed
  - ✓ Mode switch: using trap instruction, two stacks (user and kernel stack)



(Source: A. Silberschatz, "Operating system Concept")

6.2 Problem #1: Restricted Operation

- How to handle trap in OS?
  - ✓ Using trap table (a.k.a interrupt vector table)
  - ✓ Trap table consists of a set of trap handlers
    - Trap (interrupt) handler: a routine that deals with a trap in OS
    - system call handler, div_by_zero handler, segment fault handler, page fault handler, and hardware interrupt handler (disk, KBD, timer, …)
    - Initialized at boot time
  - ✓ E.g.: System call processing
    - System call (e.g. fork()) ➔ trap ➔ save context and switch stack ➔ jump to the trap handler ➔ eventually in kernel mode
    - Return from system call ➔ switch stack and restore context ➔ jump to the next instruction of the system call ➔ user mode



OS @ run (kernel mode)	Hardware	Program (user mode)
		...
		Call system call trap into OS
	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	
		...

6.2 Problem #1: Restricted Operation

- Global view

Initialize (Boot)

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

Process create

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	

Direct execution

		Run main()
		...
		Call system call
		trap into OS

Limited (syscall)

	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	

Direct execution

		...
		return from main
		trap (via `exit()`)

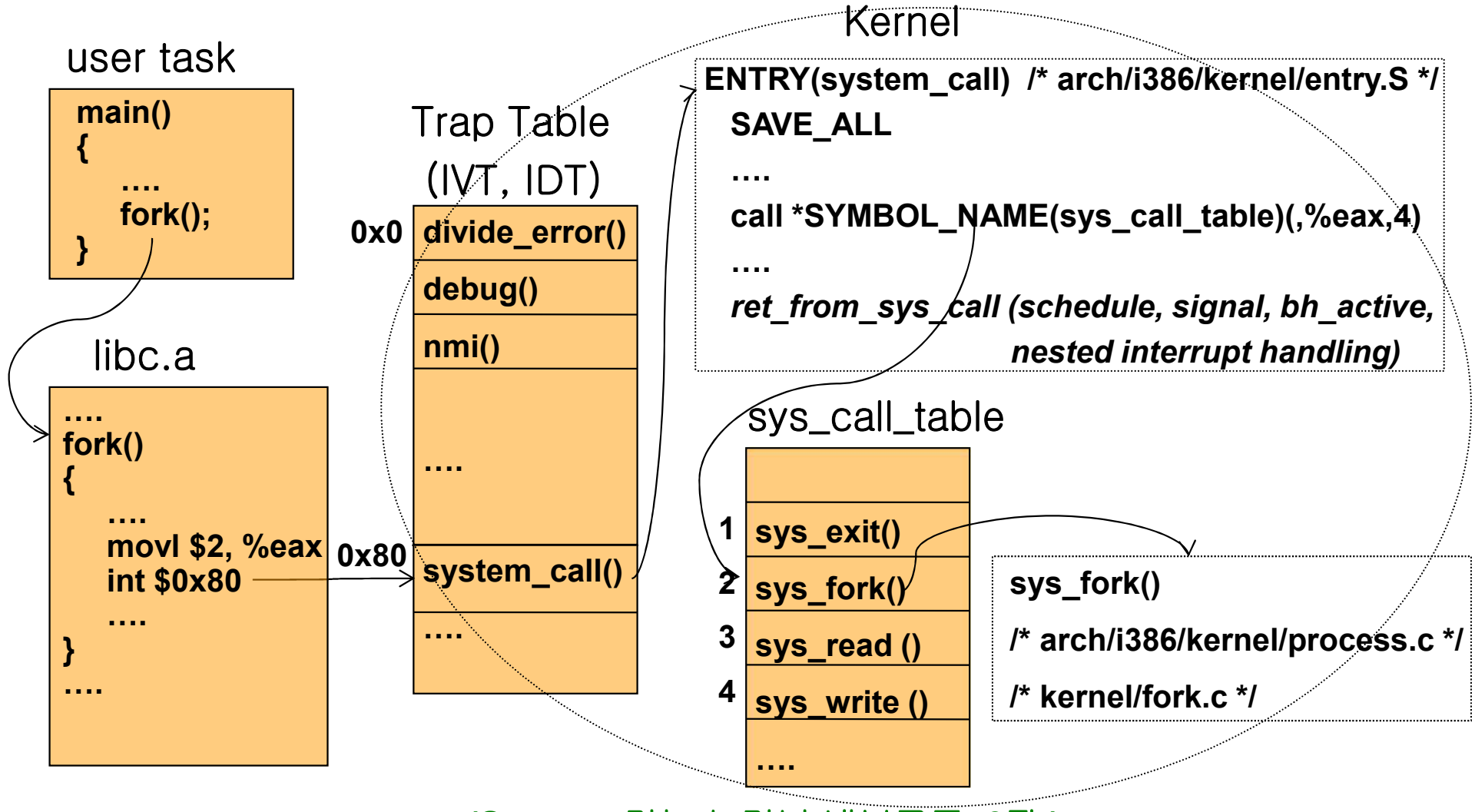
Process destroy

Free memory of process		
Remove from process list		

Figure 6.2: Limited Direct Execution Protocol

6.2 Problem #1: Restricted Operation (optional)

- System call Implementation: Linux case study



(Source: 리눅스 커널 내부구조, **6장)**

☛ Note: This mechanism is a little different in 64bit CPU, but the concept is the same

6.3 Problem #2: Switching between Processes

- Control mechanism 2: Context switch with Timer interrupt
  - ✓ Time sharing: Process A ➔ Process B ➔ Process A ➔ ….
  - ✓ By the way, how can OS regain control of the CPU so that it can switch between processes?
- Two approach
  - ✓ A cooperative approach: exploiting system calls
    - Processes use a system call ➔ control transfer to OS ➔ do scheduling (and switching)
    - A process causes exception (e.g. page fault or divide by zero) ➔ transfer control to OS
    - A process that seldom uses a system call ➔ invoke an yield() system call explicitly
    - No method for a process that does an infinite loop
  - ✓ A Non-cooperative approach: using timer interrupt

6.3 Problem #2: Switching between Processes

- A Non-cooperative approach: using timer interrupt
  - ✓ Interrupt: a mechanism that a device notify an event to OS
    - Interrupt happens ➔ current running process is halted ➔ a related interrupt handler is invoked via interrupt table ➔ transfer control to OS
  - ✓ Timer interrupt (like a heart in human)
    - A timer device raises an interrupt every milliseconds (programmable) ➔ a timer interrupt handler ➔ do scheduling (and switching) if necessary
  - ✓ Context switch
    - Context: information of a process needed when it is re-scheduled later ➔ hardware registers
    - Context save and restore
      - E.g. 1) Process A ➔ Process B: save the context of the process A and restore the context of process B. 2) later Process B ➔ Process A: save the context of the process B and restore the saved context of process A
      - Where to save: proc structure in general

6.3 Problem #2: Switching between Processes

- Context switch: global view

Initialize (Boot)

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

Process create

Direct execution

Interrupt (timer)

Scheduling and Context Switch

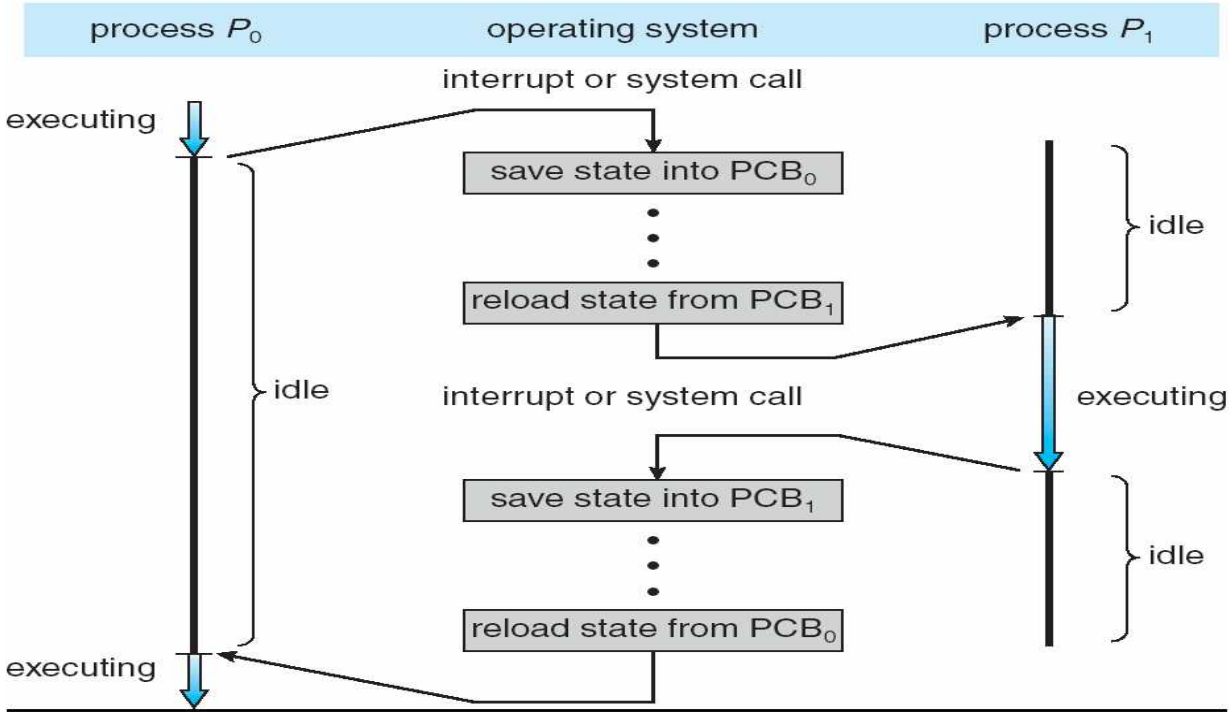
Return from Interrupt

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	
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 ...

Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)

6.3 Problem #2: Switching between Processes

- Context switch
  - ✓ Memorize the last state of a process when it is preempted
    - Context save (state save): storing CPU registers into PCB (in memory)
    - Context restore (state restore): loading PCB into CPU registers
  - ✓ Context-switch time is overhead (the system does no useful work while switching) ➔ utilizing hardware support (hyper-threading)



(Source: A. Silberschatz, "Operating system Concept")

6.3 Problem #2: Switching between Processes

- Context switch: pseudo 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
```

Figure 6.4: The xv6 Context Switch Code

6.4 Worried about concurrency?

- Some issues
  - ✓ What happens when you are handling one interrupt and another one occurs?
  - ✓ What happen when, during a system call, a timer interrupt occurs?

- Some solutions
  - ✓ Disable interrupt (note: disable interrupt too long is dangerous)
  - ✓ Priority
  - ✓ Locking mechanism
  - ✓ ➔ actually Concurrency issue

Summary

- Process (Chapter 4)
  - ✓ Process definition, Process state
  - ✓ Process management (PCB, struct proc, struct task)
- Process manipulation (Chapter 5)
  - ✓ fork(), wait(), exec(), kill() , …
- Mechanism (Chapter 6)
  - ✓ Limited Direct Execution: 1) Mode switch, 2) Context switch
  - ✓ Key terms

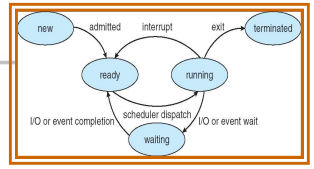
ASIDE: KEY CPU VIRTUALIZATION TERMS (MECHANISMS)

- The CPU should support at least two modes of execution: a restricted **user mode** and a privileged (non-restricted) **kernel mode**.
- Typical user applications run in user mode, and use a **system call** to **trap** into the kernel to request operating system services.
- The trap instruction saves register state carefully, changes the hardware status to kernel mode, and jumps into the OS to a pre-specified destination: the **trap table**.
- When the OS finishes servicing a system call, it returns to the user program via another special **return-from-trap** instruction, which reduces privilege and returns control to the instruction after the trap that jumped into the OS.
- The trap tables must be set up by the OS at boot time, and make sure that they cannot be readily modified by user programs. All of this is part of the **limited direct execution** protocol which runs programs efficiently but without loss of OS control.
- Once a program is running, the OS must use hardware mechanisms to ensure the user program does not run forever, namely the **timer interrupt**. This approach is a **non-cooperative** approach to CPU scheduling.
- Sometimes the OS, during a timer interrupt or system call, might wish to switch from running the current process to a different one, a low-level technique known as a **context switch**.

Quiz for this Lecture

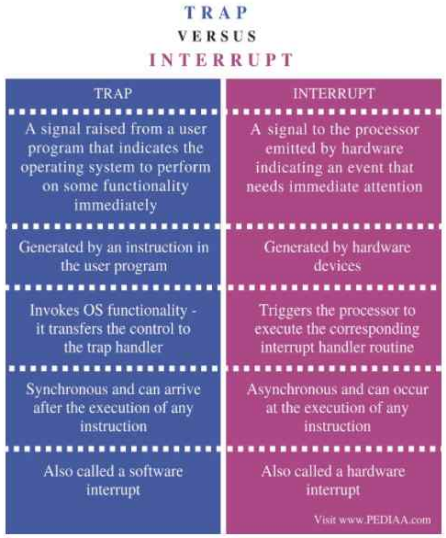
- Quiz
  - ✓ 1. Process is defined as a running program. Discuss what information are managed in PCB (Process Control Block).
  - ✓ 2. Discuss the state of the parent and child process in the below left program just after line 8, 13 and 16, respectively. (assume that the parent is scheduled before the child)
  - ✓ 3. Discuss the differences between trap and interrupt.
  - ✓ 4. Discuss how many mode switch and context switch happen in the below right figure.

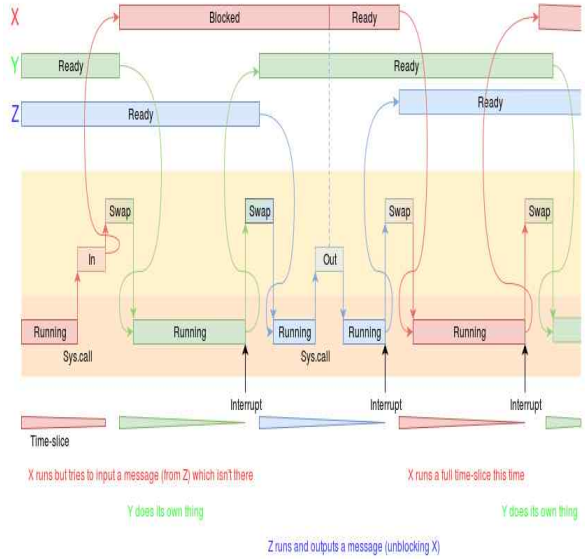
INTERLUDE: PROCESS API




```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(int argc, char *argv[]) {
7    printf("hello (pid:%d)\n", (int) getpid());
8    int rc = fork();
9    if (rc < 0) {          // fork failed; exit
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) { // child (new process)
13     printf("child (pid:%d)\n", (int) getpid());
14   } else {               // parent goes down this path
15     int rc_wait = wait(NULL);
16     printf("parent of %d (rc_wait:%d) (pid:%d)\n",
17            rc, rc_wait, (int) getpid());
18   }
19   return 0;
20 }
21
```

Figure 5.2: Calling `fork()` And `wait()` (p2.c)





(Source: pediaa.com/difference-between-trap-and-interrupt/)

(Source: xerxes.cs.manchester.ac.uk/comp251/kb/Context_Switching)

Suggestion

- Read the questions in OSTEP Chapter 5 (homework) and Chapter 6 (Measurement homework)
  - ✓ Exercise them in a Linux machine (Ubuntu on Virtual box or server)

ASIDE: CODING HOMEWORKS

Coding homeworks are small exercises where you write code to run on a real machine to get some experience with some basic operating system APIs. After all, you are (probably) a computer scientist, and therefore should like to code, right? If you don't, there is always CS theory, but that's pretty hard. Of course, to truly become an expert, you have to spend more than a little time hacking away at the machine; indeed, find every excuse you can to write some code and see how it works. Spend the time, and become the wise master you know you can be.

Homework (Code)

In this homework, you are to gain some familiarity with the process management APIs about which you just read. Don't worry – it's even more fun than it sounds! You'll in general be much better off if you find as much time as you can to write some code, so why not start now?

Questions

1. Write a program that calls `fork()`. Before calling `fork()`, have the main process access a variable (e.g., `x`) and set its value to something (e.g., `100`). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of `x`?
2. Write a program that opens a file (with the `open()` system call) and then calls `fork()` to create a new process. Can both the child and parent access the file descriptor returned by `open()`? What happens when they are writing to the file concurrently, i.e., at the same time?
3. Write another program using `fork()`. The child process should print "hello"; the parent process should print "goodbye". You should try to ensure that the child process always prints first; can you do this *without* calling **wait()** in the parent?
4. Write a program that calls `fork()` and then calls some form of `exec()` to run the program `/bin/ls`. See if you can try all of the variants of `exec()`, including (on Linux) `execl()`, `execle()`, `execlp()`, `execv()`, `execvp()`, and `execvpe()`. Why do you think there are so many variants of the same basic call?

Homework (Measurement)

ASIDE: MEASUREMENT HOMEWORKS

Measurement homeworks are small exercises where you write code to run on a real machine, in order to measure some aspect of OS or hardware performance. The idea behind such homeworks is to give you a little bit of hands-on experience with a real operating system.

In this homework, you'll measure the costs of a system call and context switch. Measuring the cost of a system call is relatively easy. For example, you could repeatedly call a simple system call (e.g., performing a 0-byte read), and time how long it takes; dividing the time by the number of iterations gives you an estimate of the cost of a system call.

One thing you'll have to take into account is the precision and accuracy of your timer. A typical timer that you can use is `gettimeofday()`; read the man page for details. What you'll see there is that `gettimeofday()` returns the time in microseconds since 1970; however, this does not mean that the timer is precise to the microsecond. Measure back-to-back calls to `gettimeofday()` to learn something about how precise the timer really is; this will tell you how many iterations of your null system-call test you'll have to run in order to get a good measurement result. If `gettimeofday()` is not precise enough for you, you might look into using the `rdtsc` instruction available on x86 machines.

Measuring the cost of a context switch is a little trickier. The lmbench

Appendix

- Answers for questions commonly asked by students

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5
6  int
7  main(int argc, char *argv[])
8  {
9      int *p = malloc(sizeof(int));                    // a1
10     assert(p != NULL);
11     printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                            // a2
13     *p = 0;                                          // a3
14     while (1) {
15         Spin(1);
16         *p = *p + 1;
17         printf("(%d) p: %d\n", getpid(), *p);        // a4
18     }
19     return 0;
20 }
```

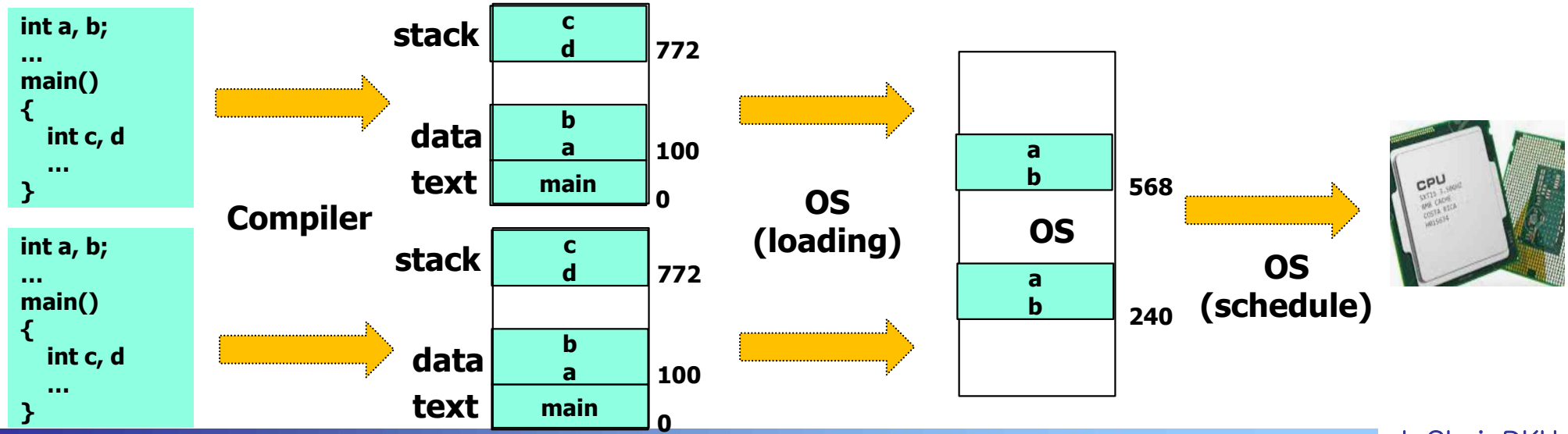
Figure 2.3: A Program That Accesses Memory (mem.c)

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Figure 2.4: Running The Memory Program Multiple Times

(Source: Chapter 2 in OSTEP)

- ✓ Q1: same address in the two processes?
- ✓ Q2: why not 1➔ 2 ➔ 3 ➔ 4 ➔ ….
- ✓ Key concept: Program ➔ CPU using Compiler and OS



Program Binary (virtual address) DRAM (physical address) CPU