Lecture Note 2: Processes

March 7, 2021

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 States 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.*

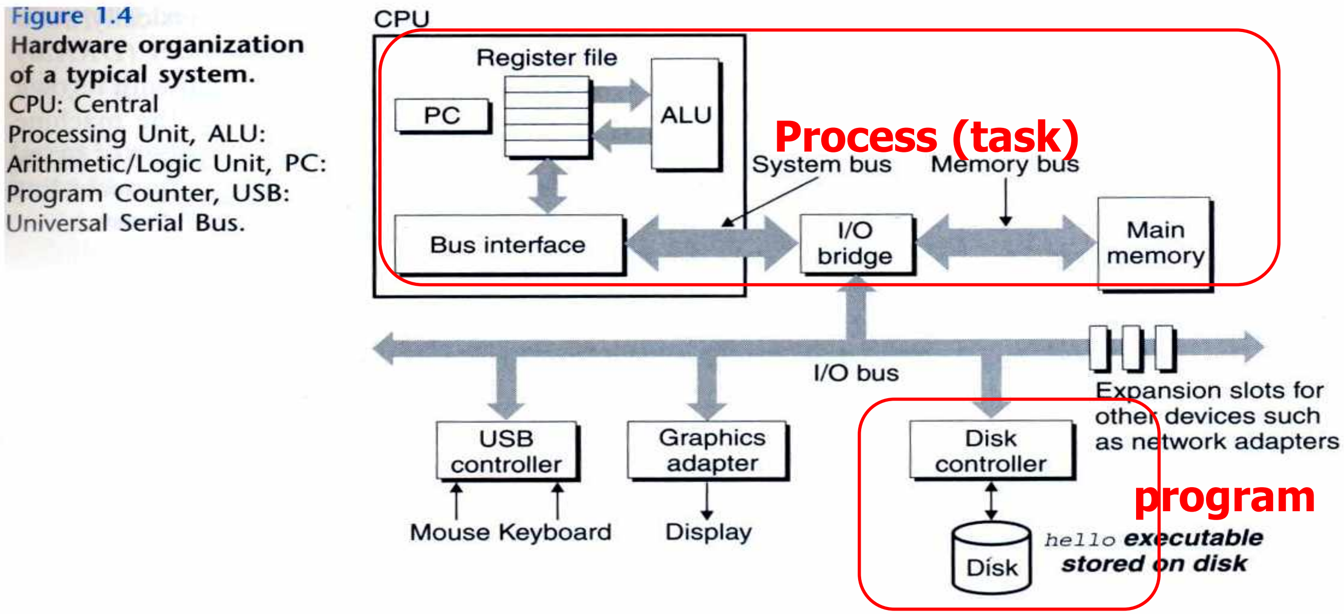
Professor: *Ah yes! Good point. But that is the thing with many eaters; most of the time they are napping or doing something else, and thus, you can snatch that peach away and give it to someone else for a while. And thus we create the illusion of many virtual peaches, one peach for each person!*

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 running program, scheduling entity
    - c.f.) program: a lifeless thing, sit on the disk and waiting to spring into action
    - Run on memory and CPU
  - ✓ There exist multiple processes (e.g. browser, editor, player, and so on)
    - Each process has its own memory (address space), virtual CPU, state, …



(Source: computer systems: a programmer perspective)

Chap 4. The Abstraction: The Process

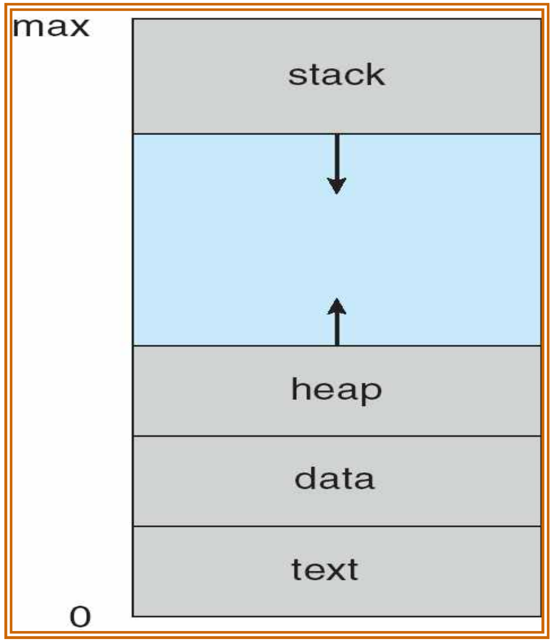
- How to virtualize CPU? Time sharing on multiple processes

- ✓ 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")

  - ✓ cf.) program
    - Passive entity
    - A file containing instructions stored on disk (executable file or binary)
    - 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.

☛ Refer to chapter 5 in OSTEP

4.3 Process Creation: 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()

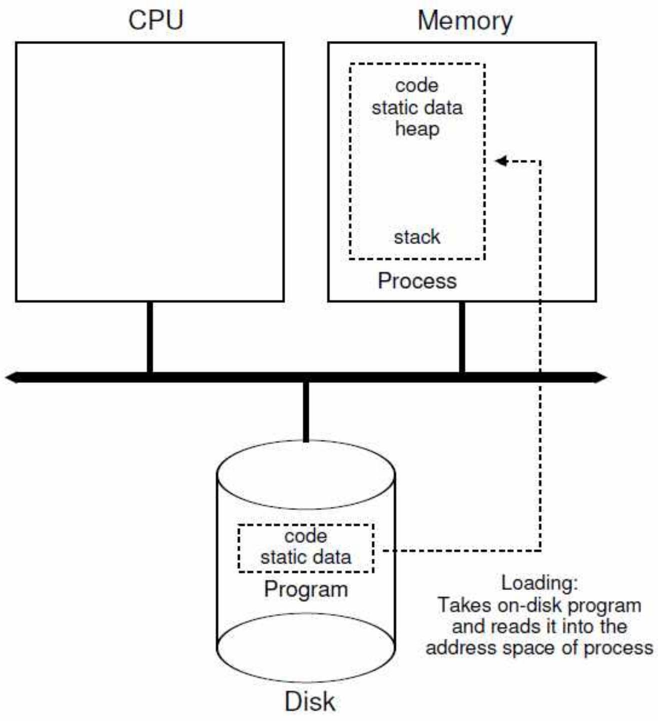
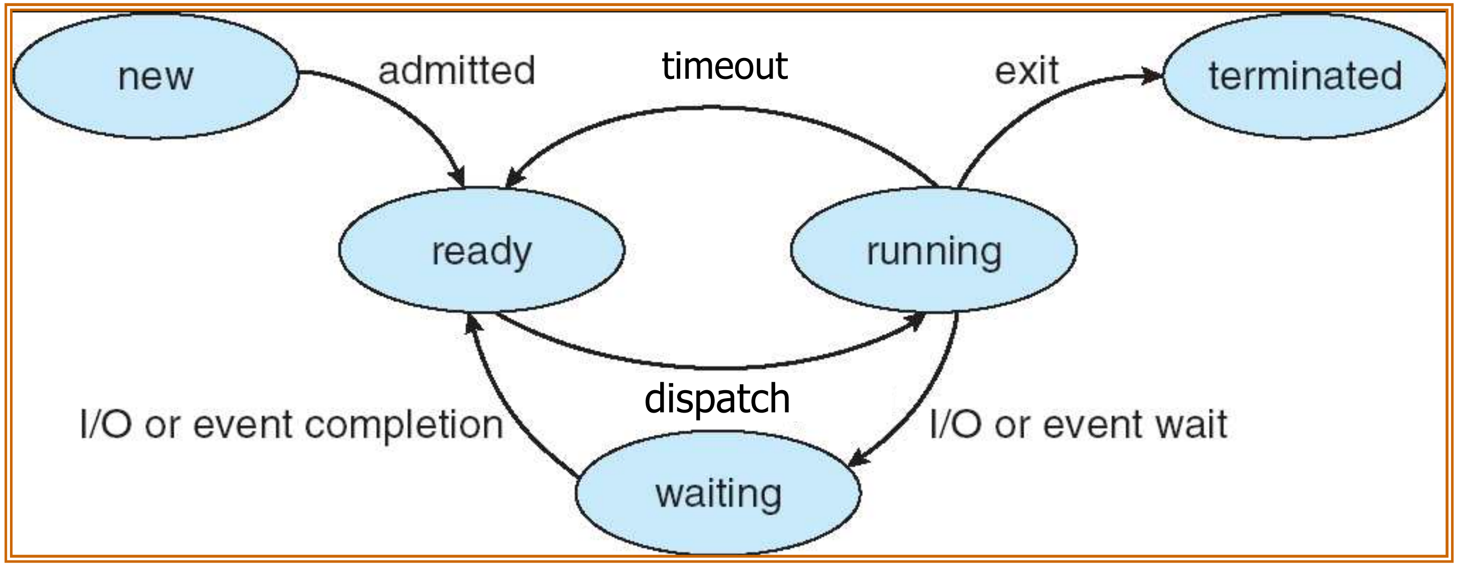


Figure 4.1: **Loading: From Program To Process**

4.4 Process States

- State and transition

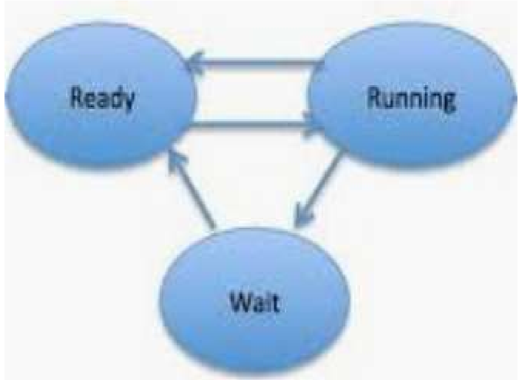


(Source: A. Silberschatz, “Operating system Concept”)

- ✓ State
  - new(created, embryo), ready, running, waiting(blocked), terminated (zombie)
- ✓ Transition
  - admitted, dispatch (schedule), timeout (preemptive, descheduled), wait (sleep, I/O initiate), wakeup (I/O done), 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**

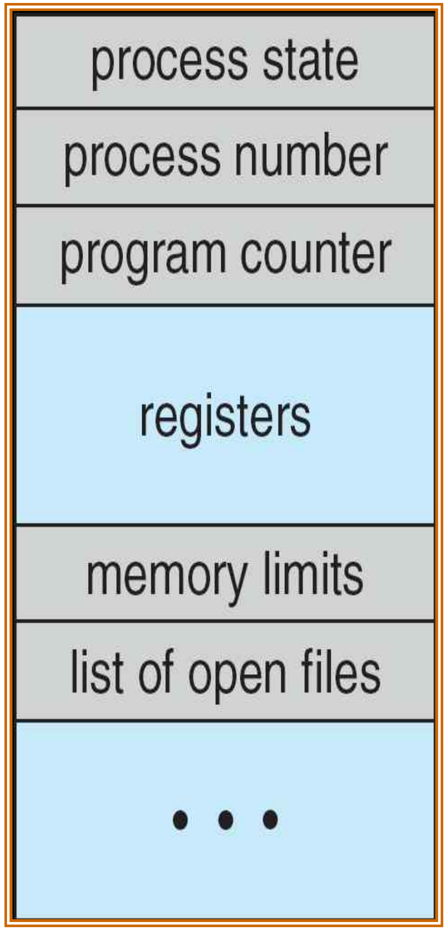
Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$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

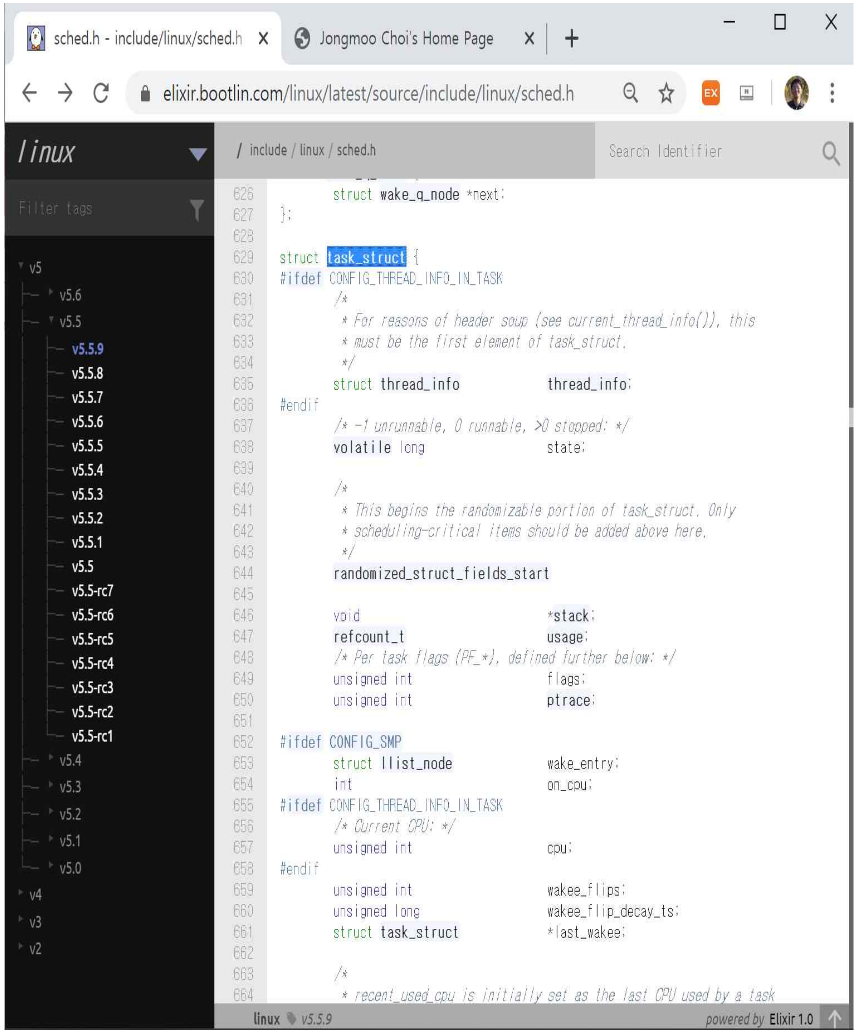
- Implementation example
  - ✓ OS is a program, implementing a process using data structure (e.g. struct proc and struct context)
  - ✓ All "proc" structures are manipulated using a list

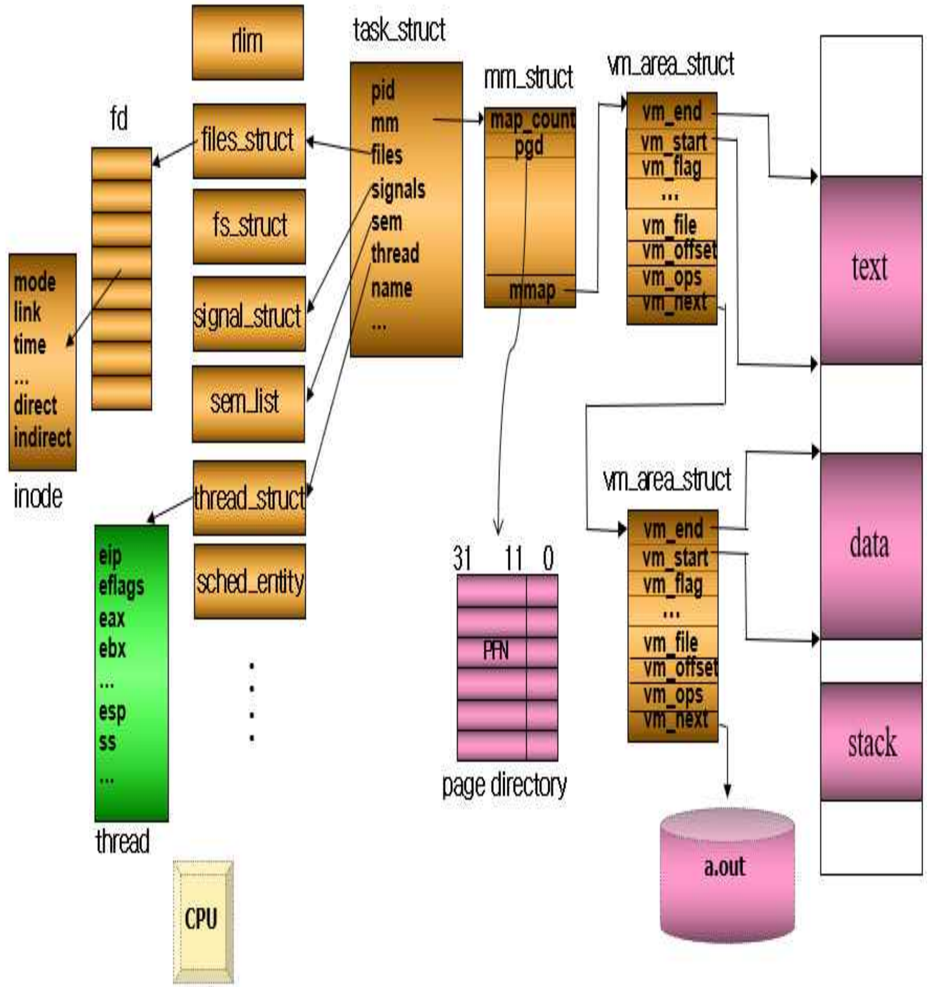
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// 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 and the other for child
  - ✓ Non-determinism: not decide which one run first.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {           // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {                // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
                rc, (int) getpid());
    }
    return 0;
}
```

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {              // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

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

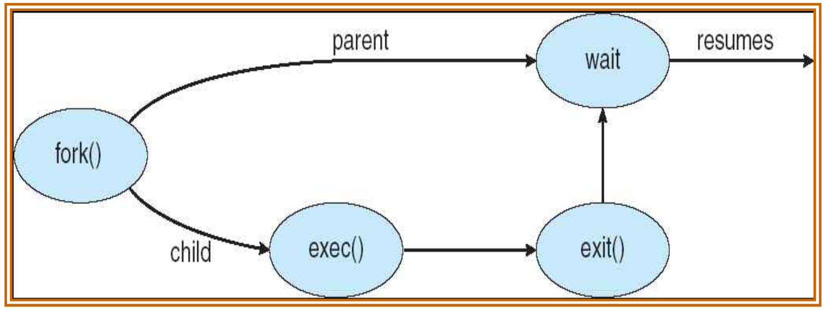
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {               // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

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 (especially for shell)




```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {           // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {                // parent goes down this path (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

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

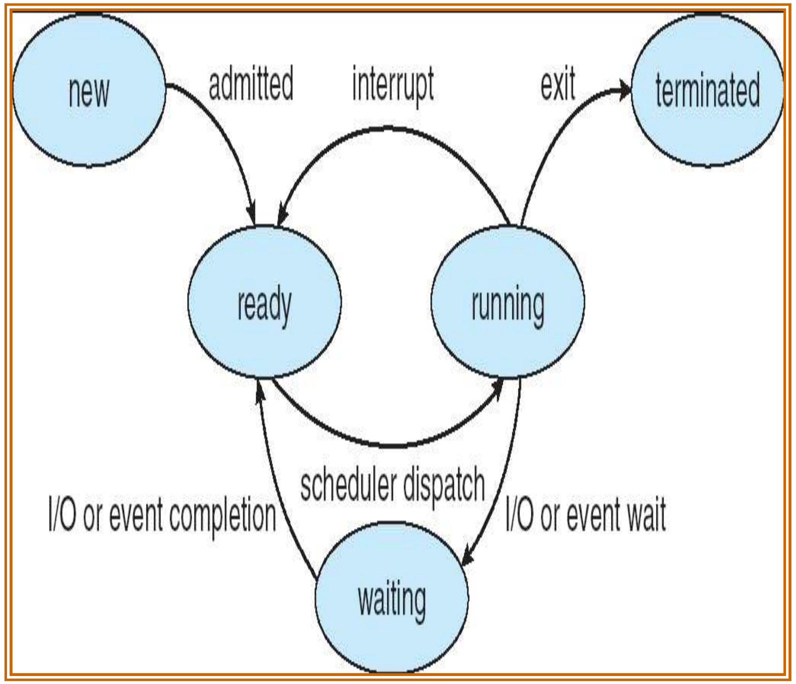
Many times in this book, when referring to a particular system call or library call, we'll tell you to read the **manual pages**, or **man pages** for short. Man pages are the original form of documentation that exist on UNIX systems; realize that they were created before the thing called **the web**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

Quiz for 2nd-Week 2nd-Lesson

- Quiz
  - ✓ 1. OS makes use of a (   ) mechanism and a (   ) policy to virtualize CPU (for time sharing on multiple processes)
  - ✓ 2. Discuss the state of the parent and child process in the below program just after line 10, 15 and 18, respectively.
  - ✓ Due: until 6 PM Friday of this week (12th, March)

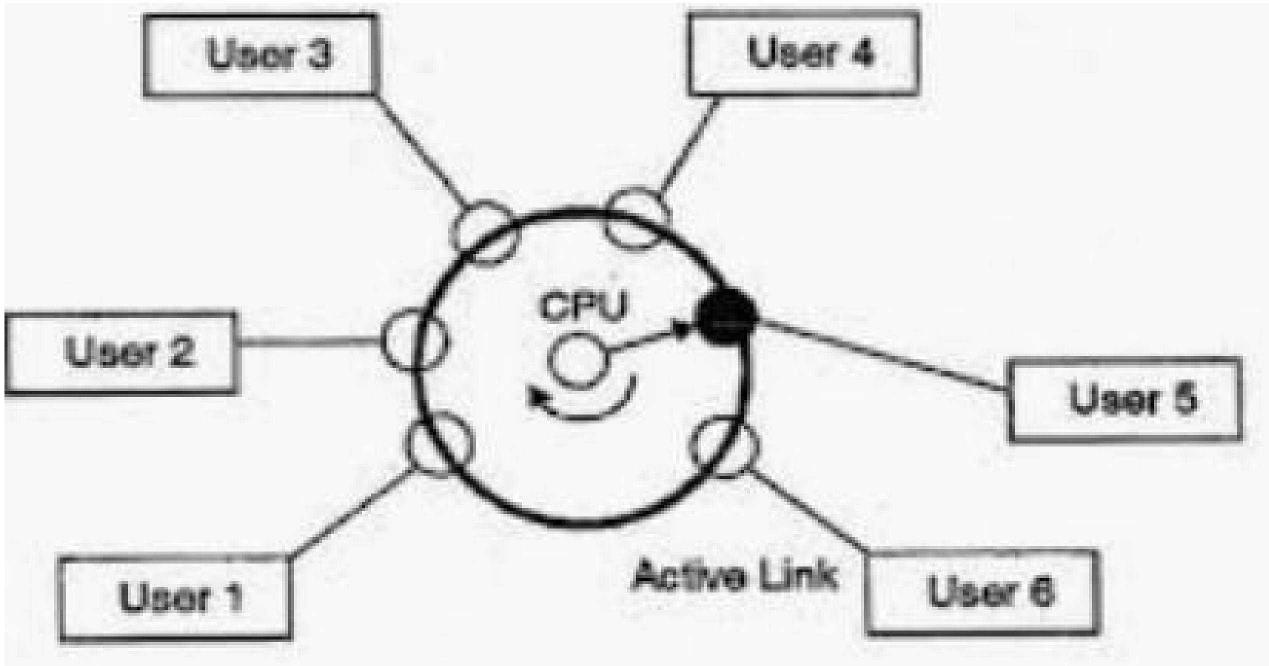



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int
7  main(int argc, char *argv[])
8  {
9      printf("hello world (pid:%d)\n", (int) getpid());
10     int rc = fork();
11     if (rc < 0) {          // fork failed; exit
12         fprintf(stderr,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

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

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

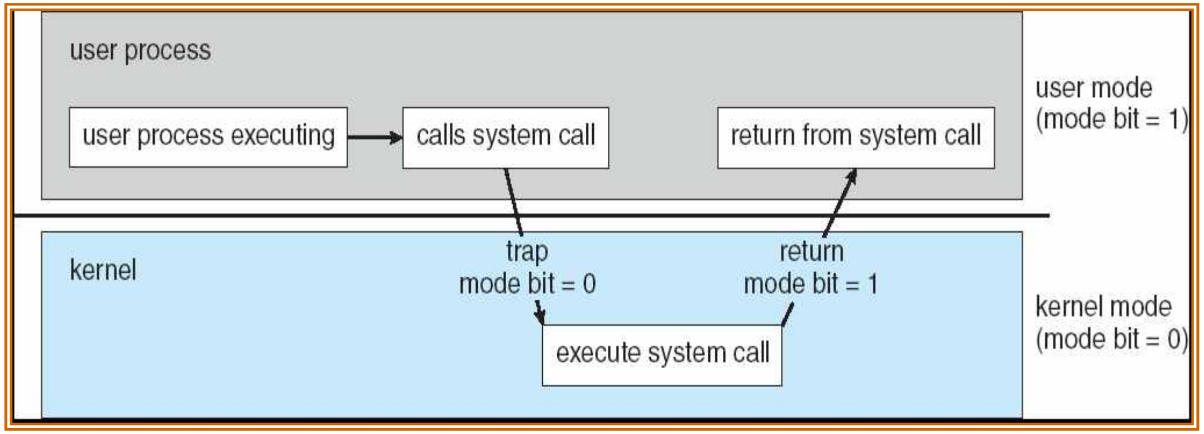
Figure 6.1: **Direct Execution Protocol (Without Limits)**

☛ **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
  - ✓ Operations that should run indirectly (in a privileged mode)
    - 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(), …

- Mechanism: 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 hander (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

0	**divide_by_zero()**
1	**page_fault()**
2	**segment_fault()**
..	**…**
80	**system_call()**

trap table

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

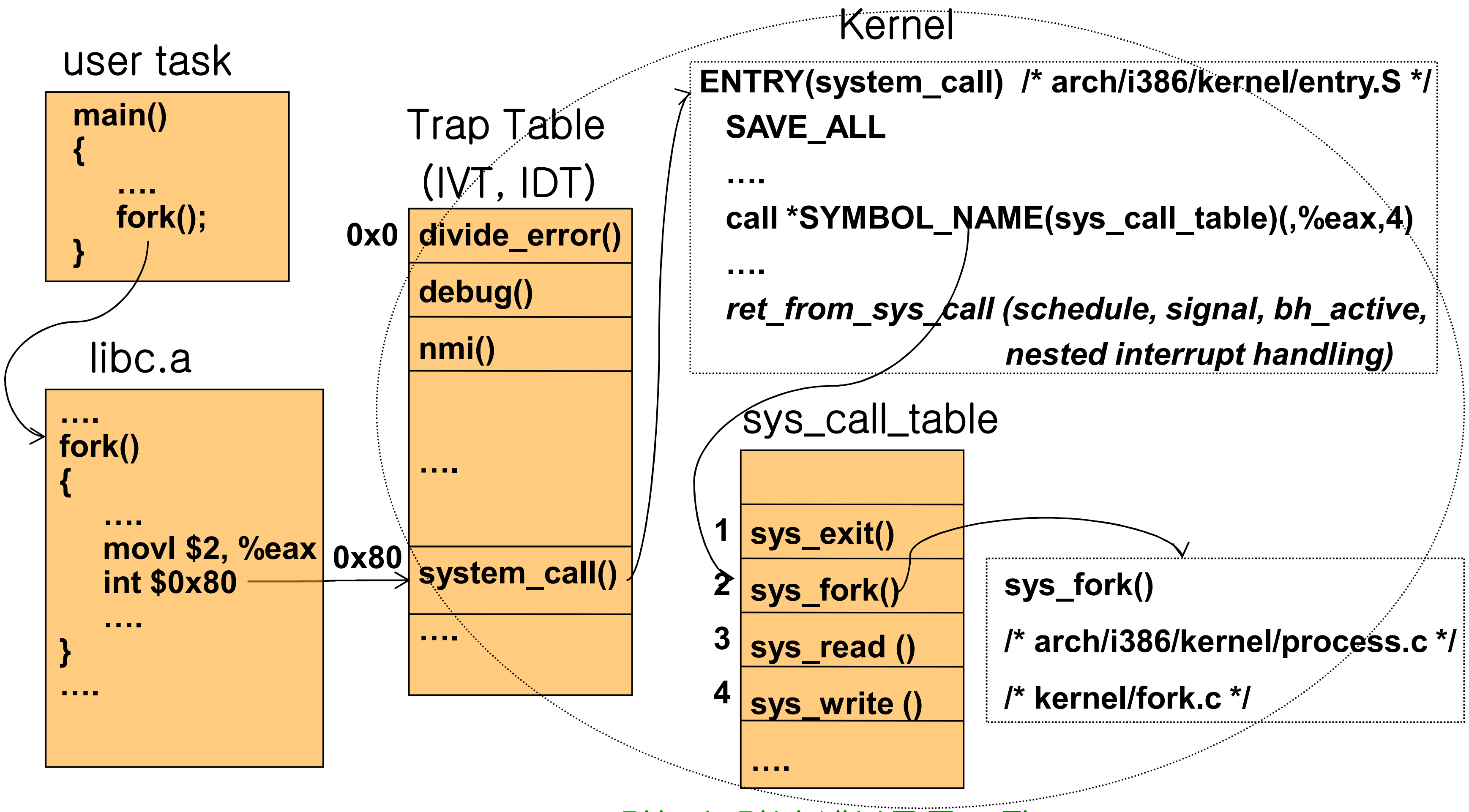
	OS @ boot (kernel mode)	Hardware	
Initialize (Boot)	initialize trap table		
		remember address of... syscall handler	

	OS @ run (kernel mode)	Hardware	Program (user mode)
Process create	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 call trap into OS
Trap (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	
			... 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 hander 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

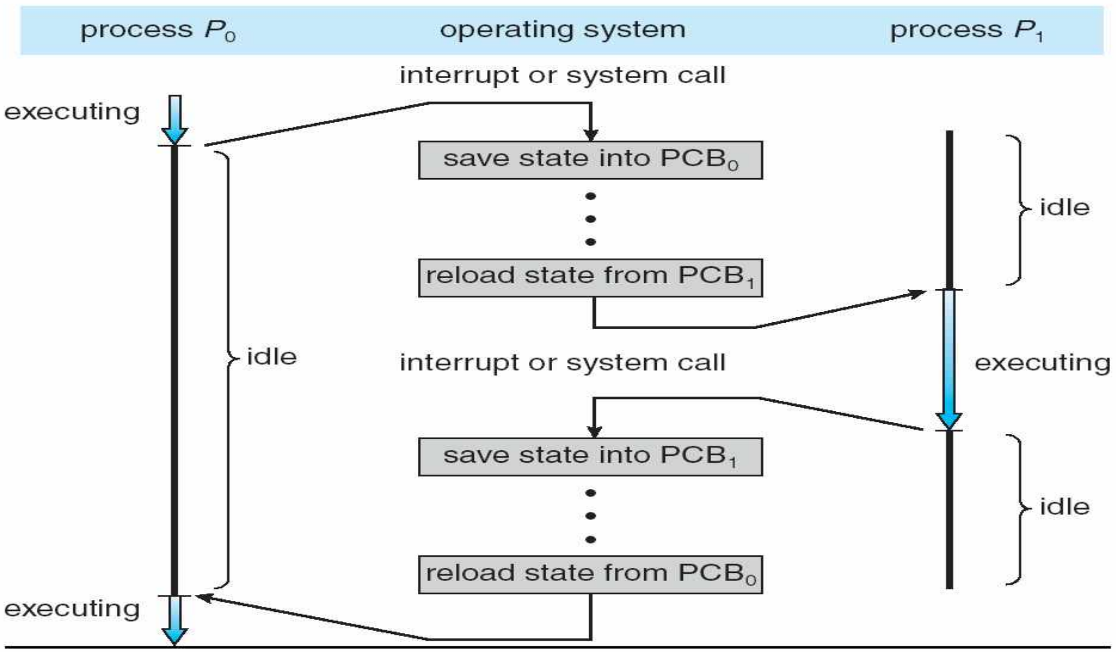
	OS @ boot (kernel mode)	Hardware
Initialize (Boot)	initialize trap table	
		remember addresses of... syscall handler timer handler
	start interrupt timer	
		start timer interrupt CPU in X ms

	OS @ run (kernel mode)	Hardware	Program (user mode)
			Process A ...
Interrupt (timer)		timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Scheduling and Context Switch	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)		
Return from Interrupt		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

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax  # put old ptr into eax
  popl 0(%eax)        # save the old IP
  movl %esp, 4(%eax)  # and stack
  movl %ebx, 8(%eax)  # and other registers
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)

  # Load new registers
  movl 4(%esp), %eax  # put new ptr into eax
  movl 28(%eax), %ebp # restore other registers
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp  # stack is switched here
  pushl 0(%eax)       # return addr put in place
  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
  - ✓ Performance: system call ➔ 4 us, context switch ➔ 6 us on P6 CPU

☛ Suggestion:
- – Check the questions in Chap. 5 (homework) and 6 (measurement homework)
- – Exercise them on a Linux server

百見不如一打

Appendix

- Answers for questions commonly asked by students

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));                    // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",
           getpid(), p);                             // a2
    *p = 0;                                          // a3
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p);        // a4
    }
    return 0;
}
```

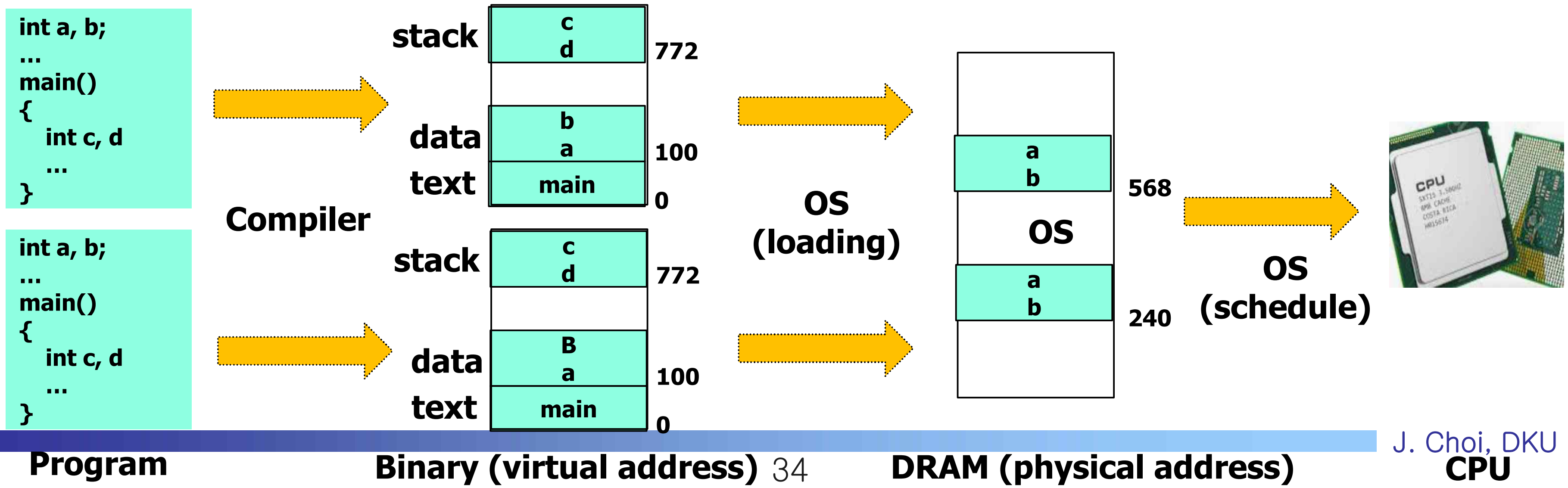
Figure 2.3: A Program That Accesses Memory (mem.c)

```
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

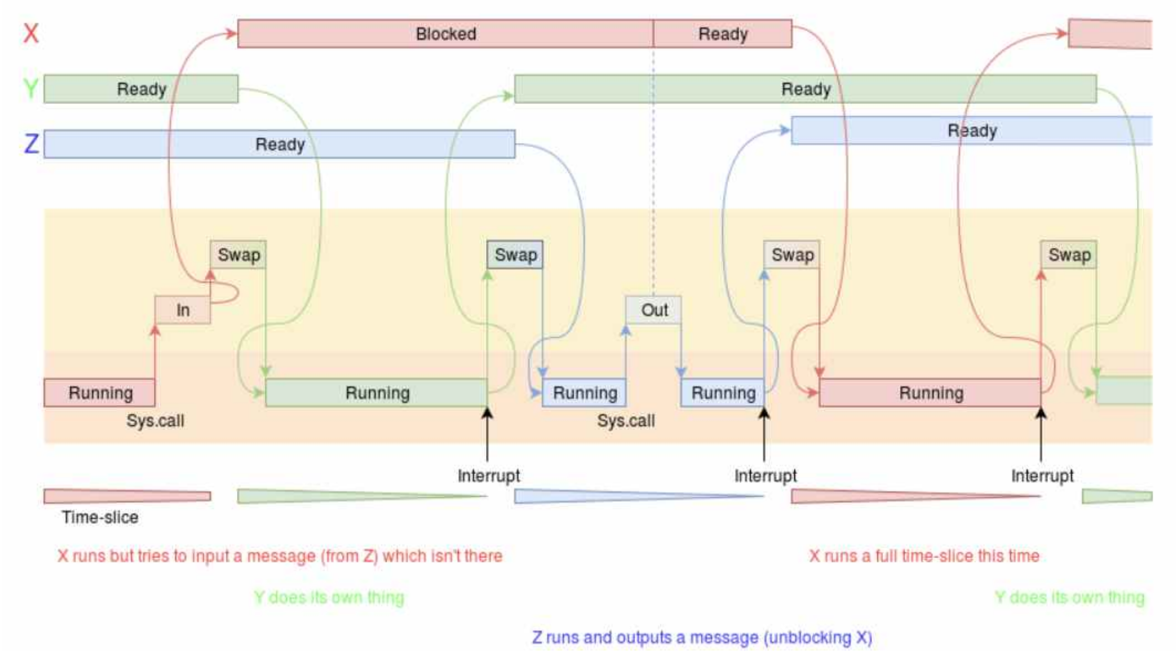
(Source: Chapter 2 in OSTEF

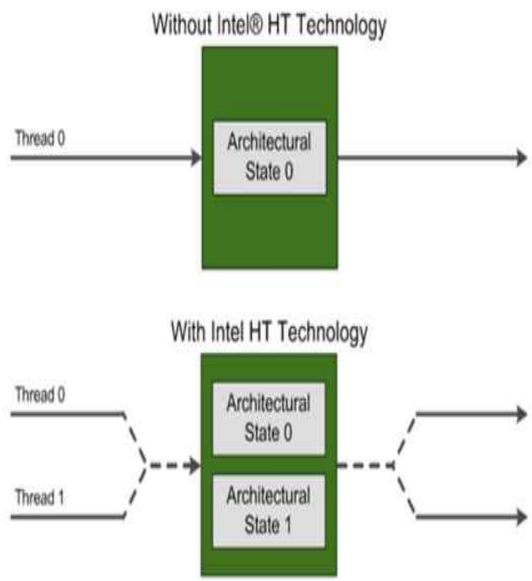
- ✓ Q1: same address in the two processes?
- ✓ Q2: why not 1➔ 2 ➔ 3 ➔ 4 ➔ ….
- ✓ Key concept: Program ➔ CPU using Compiler and OS



Quiz for 3rd-Week 1st-Lesson

- Quiz
  - ✓ 1. Discuss how many mode switch and context switch happen in the below left figure.
  - ✓ 2. What is the AS (Architectural State) in Intel's Hyperthreading Technology?
  - ✓ Due: until 6 PM Friday of this week (19th, March)





(Source: https://xerxes.cs.manchester.ac.uk/comp251/kb/Context_Switching and https://www.dasher.com/will-hyper-threading-improve-processing-performance/)