

Review of Operating Systems

Operating System: Three Easy Pieces

What happens when a program runs?

- ▣ A running program executes instructions.
 1. The processor **fetches** an instruction from memory.
 2. **Decode**: Figure out which instruction this is
 3. **Execute**: i.e., add two numbers, access memory, check a condition, jump to function, and so forth.
 4. The processor moves on to the **next instruction** and so on.

Operating System (OS)

- Responsible for

- ◆ Making it easy to **run** programs
- ◆ Allowing programs to **share** memory
- ◆ Enabling programs to **interact** with devices

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

Virtualization

- ▣ The OS takes **a physical resource** and transforms it into a **virtual form** of itself.
 - **Physical resource:** Processor, Memory, Disk ...
 - ◆ The virtual form is more general, powerful and easy-to-use.
 - ◆ Sometimes, we refer to the OS as a **virtual machine**.

System call

- ▣ System call allows user **to tell the OS what to do.**
 - ◆ The OS provides some interface (APIs, standard library).
 - ◆ A typical OS exports a few hundred system calls.
 - Run programs
 - Access memory
 - Access devices

The OS is a resource manager.

- ▣ The OS **manage resources** such as *CPU, memory* and *disk*.
- ▣ The OS allows
 - ◆ Many programs to run → Sharing the CPU
 - ◆ Many programs to *concurrently* access their own instructions and data → Sharing memory
 - ◆ Many programs to access devices → Sharing disks

Virtualizing the CPU

- ▣ The system has a very large number of virtual CPUs.
 - ◆ Turning a single CPU into a seemingly infinite number of CPUs.
 - ◆ Allowing many programs to seemingly run at once
→ **Virtualizing the CPU**

Virtualizing the CPU (Cont.)

```
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); // Repeatedly checks the time and
17         // returns once it has run for a second
18         printf("%s\n", str);
19     }
20 }
```

Simple Example(cpu.c): Code That Loops and Prints

All Code in--> <https://github.com/remzi-arpacidusseau/ostep-code>

Virtualizing the CPU (Cont.)

- Execution result 1.

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
^C
prompt>
```

Run forever; Only by pressing “Control-c” can we halt the program

Virtualizing the CPU (Cont.)

❑ Execution result 2.

```
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
C
B
D
...
```

Even though if we have only **one processor**, all four of programs seem to be running at the same time!

Virtualizing Memory

- ▣ The physical memory is *an array of bytes*.
- ▣ A program keeps all of its data structures in memory.
 - ◆ **Read memory** (load):
 - Specify an address to be able to access the data
 - ◆ **Write memory** (store):
 - Specify the data to be written to the given address

Virtualizing Memory (Cont.)

▣ A program that Accesses Memory (mem.c)

```
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: allocate some
10         memory
11         assert(p != NULL && "Error on malloc");
12         printf("(%d) address of p: %08x\n",
13             getpid(), (unsigned) p); // a2: print out the
14             address of the memory
15         *p = 0; // a3: put zero into the first slot of the memory
16         while (1) {
17             Spin(1);
18             *p = *p + 1;
19             printf("(%d) p: %d\n", getpid(), *p); // a4
20         }
21     return 0;
22 }
```

Virtualizing Memory (Cont.)

❑ The output of the program mem.c

```
prompt> ./mem
(2134) memory address of p: 00200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

- ◆ The newly allocated memory is at address 00200000.
- ◆ It updates the value and prints out the result.

Virtualizing Memory (Cont.)

Running `mem.c` multiple times

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 00200000
(24114) memory address of p: 00200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
...
...
```

- It is as if each running program has its **own private memory**.
 - Each running program has allocated memory at the same address.
 - Each seems to be updating the value at 00200000 independently.
- CAVEAT: In current systems, ASLR (Address Space Layout Randomization) Makes this to change.

To disable it:

- Linux: `echo 0 > /proc/sys/kernel/randomize_va_space`
- OSX: `gcc -o mem mem.c -Wall -Wl,-no_pie`

Virtualizing Memory (Cont.)

- ▣ Each process accesses its own private **virtual address space**.
 - ◆ The OS maps **address space** onto the **physical memory**.
 - ◆ A memory reference within one running program does not affect the address space of other processes.
 - ◆ Physical memory is a shared resource, managed by the OS.

Persistence

- ▣ Devices such as DRAM store values in a volatile.
- ▣ *Hardware* and *software* are needed to store data **persistently**.
 - ◆ **Hardware:** I/O device such as a hard drive, solid-state drives(SSDs)
 - ◆ **Software:**
 - File system manages the disk.
 - File system is responsible for storing any files the user creates.

Persistence (Cont.)

- >Create a file (/tmp/file) that contains the string "hello world"

```
1      #include <stdio.h>
2      #include <unistd.h>
3      #include <assert.h>
4      #include <fcntl.h>
5      #include <sys/types.h>
6
7      int
8      main(int argc, char *argv[])
9      {
10          int fd = open("/tmp/file", O_WRONLY | O_CREAT
11                      | O_TRUNC, S_IRWXU);
12          assert(fd > -1 && "Error creating file");
13          int rc = write(fd, "hello world\n", 13);
14          assert(rc == 13 && "Disk full?");
15          close(fd);
16          return 0;
17      }
```

open(), write(), and close() system calls are routed to the part of OS called the file system, which handles the requests

Persistence (Cont.)

- ▣ What OS does in order to write to disk?
 - ◆ Figure out **where** on disk this new data will reside
 - ◆ **Issue I/O** requests to the underlying storage device
- ▣ File system handles system crashes during write.
 - ◆ **Journaling** or **copy-on-write**
 - ◆ Carefully ordering writes to disk

Design Goals

- ▣ **Build up abstraction**
 - ◆ Make the system convenient and easy to use.
- ▣ **Provide high performance**
 - ◆ Minimize the overhead of the OS.
 - ◆ OS must strive to provide virtualization without excessive overhead.
- ▣ **Protection** between applications
 - ◆ Isolation: Bad behavior of one does not harm other and the OS itself.

Design Goals (Cont.)

- High degree of **reliability**
 - ◆ The OS must also run non-stop.
- Other issues
 - ◆ Energy-efficiency
 - ◆ Security
 - ◆ Mobility

The Abstraction: The Process

Operating System: Three Easy Pieces

How to provide the illusion of many CPUs?

- ▣ CPU virtualizing

- ◆ The OS can promote the illusion that many virtual CPUs exist.
- ◆ **Time sharing:** Running one process, then stopping it and running another
 - The potential cost is **performance**.

A Process

A process is a **running program**.

- Comprising of a process:
 - ◆ Memory (address space)
 - Instructions
 - Data section
 - ◆ Registers (processor architectural state (?))
 - Program counter
 - Stack pointer
 - ...

Process API

- These APIs are available on any modern OS.

- ◆ **Create**

- Create a new process to run a program

- ◆ **Destroy**

- Halt a runaway process

- ◆ **Wait**

- Wait for a process to stop running

- ◆ **Miscellaneous Control**

- Some kind of method to suspend a process and then resume it

- ◆ **Status**

- Get some status info about a process

- ◆ ...

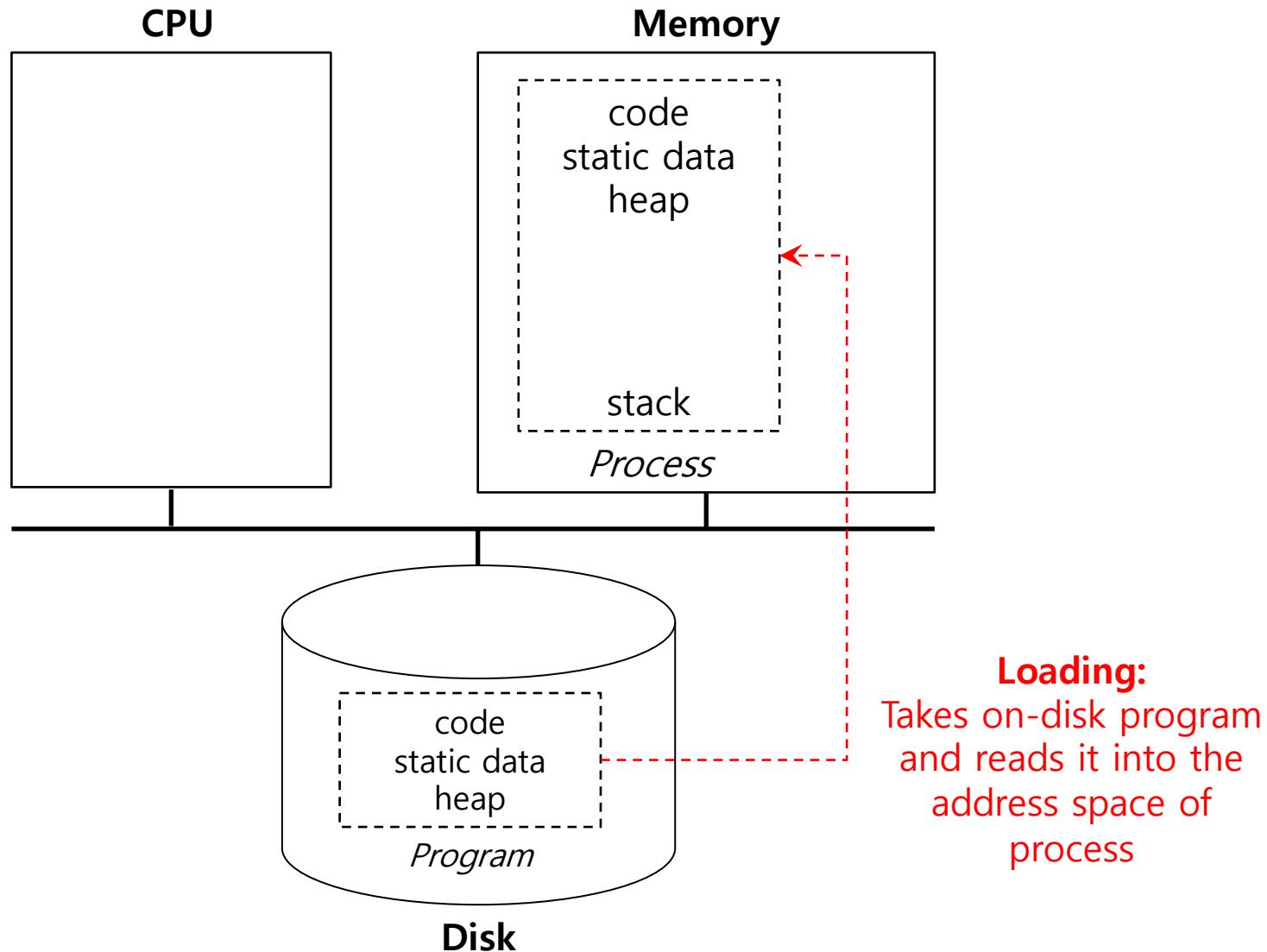
Process Creation

1. **Load** a program code into memory, into the address space of the process.
 - ◆ Programs initially reside on disk in *executable format (code + static data)*.
 - ◆ OS perform the loading process **lazily**.
 - Loading pieces of code or data only as they are needed during program execution.
2. The program's run-time **stack** is allocated.
 - ◆ Use the stack for *local variables, function parameters, and return address*.
 - ◆ Initialize the stack with arguments → argc and the argv array of main() function

Process Creation (Cont.)

3. The program's **heap** is created.
 - ◆ Used for explicitly requested dynamically allocated data.
 - ◆ Program request such space by calling `malloc()` and free it by calling `free()`.
4. The OS do some other initialization tasks.
 - ◆ input/output (I/O) setup
 - Each process by default has three open file descriptors.
 - Standard input, output and error
5. **Start the program** running at the entry point, namely `main()`.
 - ◆ The OS *transfers control* of the CPU to the newly-created process.

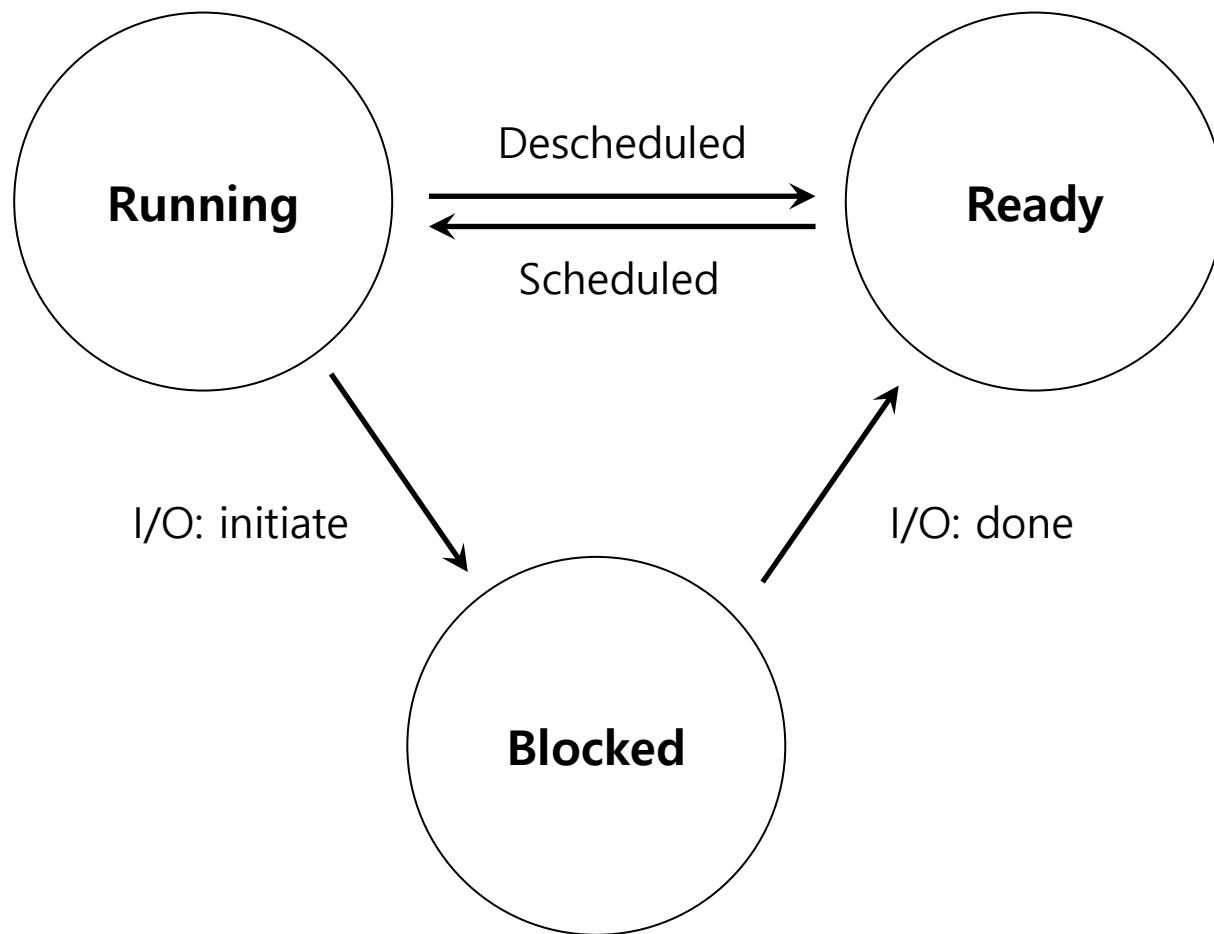
Loading: From Program To Process



Process States

- ▣ 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



Data structures

- ▣ The OS has **some key data structures** that track various relevant pieces of information.
 - ◆ **Process list**
 - Ready processes
 - Blocked processes
 - Current running process
 - ◆ **Register context**
- ▣ PCB(Process Control Block)
 - ◆ A C-structure that contains information **about each process**.

Example) The xv6 kernel Proc Structure

```
// 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 };
```

XV6 available in <https://github.com/mit-pdos/xv6-public>

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[NFILE];      // Open files
    struct inode *cwd;              // Current directory
    struct context context;          // Switch here to run process
    struct trapframe *tf;           // Trap frame for the
                                    // current interrupt
};
```

Mechanism: Limited Direct Execution

Operating System: Three Easy Pieces

How to efficiently virtualize the CPU with control?

- ▣ The OS needs to share the physical CPU by **time sharing**.
- ▣ Issue
 - ◆ **Performance:** How can we implement virtualization without adding excessive overhead to the system?
 - ◆ **Control:** How can we run processes efficiently while retaining control over the CPU?

Direct Execution (without limits!)

- Just run the program directly on the CPU.

OS	Program
<ol style="list-style-type: none">1. Create entry for process list2. Allocate memory for program3. Load program into memory4. Set up stack with <code>argc / argv</code>5. Clear registers6. Execute call <code>main()</code> 9. Free memory of process10. Remove from process list	<ol style="list-style-type: none">7. Run <code>main()</code>8. Execute <code>return</code> from <code>main()</code>

Without *limits* on running programs,
the OS wouldn't be in control of anything and
thus, would be "**just a library**"

Problem 1: Restricted Operation

- ▣ What if a process wishes to perform some kind of restricted operation such as ...
 - ◆ Issuing an I/O request to a disk
 - ◆ Gaining access to more system resources such as CPU or memory
- ▣ **Solution:** Using protected control transfer (processor must support it)
 - ◆ **User mode:** Applications do not have full access to hardware resources.
 - ◆ **Kernel mode:** The OS has access to the full resources of the machine

System Call

- ▣ Allow the kernel to **carefully expose** certain key pieces of functionality to user program, such as ...
 - ◆ Accessing the file system
 - ◆ Creating and destroying processes
 - ◆ Communicating with other processes
 - ◆ Allocating more memory

**But why they look like a regular procedure call “sometimes”
(e.g., libc calls)?**

System Call (Cont.)

▫ Trap instruction

- ◆ Jump into the kernel (how to tell where?)
- ◆ Raise (the processor) privilege level to kernel mode

▫ Return-from-trap instruction

- ◆ Return into the calling user program
- ◆ Reduce (the processor) privilege level back to user mode

▫ Trap Table

Limited Direction Execution Protocol

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember address of ... syscall handler	
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 proc kernel stack move to user mode jump to main	Run main() ... Call system trap into OS

Limited Direction Execution Protocol (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
<p>Handle trap Do work of syscall return-from-trap</p> <p>Free memory of process Remove from process list</p>	<p>save regs to proc kernel stack move to kernel mode jump to trap handler</p> <p>restore regs from proc kernel stack move to user mode jump to PC after trap</p>	<p>...</p> <p>return from main trap (via <code>exit()</code>)</p>

Problem 2: Switching Between Processes

- ▣ How can the OS **regain control** of the CPU so that it can switch between *processes*?
 - ◆ A cooperative Approach: **Wait for system calls**
 - ◆ A Non-Cooperative Approach: **The OS takes control**

A cooperative Approach: Wait for system calls

- ▣ Processes **periodically give up the CPU** by making **system calls** such as `yield`.
 - ◆ The OS decides to run some other task.
 - ◆ Application also transfer control to the OS when they do something illegal.
 - Divide by zero
 - Try to access memory that it shouldn't be able to access
 - ◆ Ex) Early versions of the Macintosh OS, The old Xerox Alto system

A process gets stuck in an infinite loop.
→ **Reboot the machine**

A Non-Cooperative Approach: OS Takes Control

▫ A timer interrupt

- ◆ During the boot sequence, the OS starts the timer (hardware).
- ◆ The timer raise an interrupt every so many milliseconds. (hardware)
- ◆ When the interrupt is raised :
 - The currently running process is halted.
 - Save enough of the state of the program
 - A pre-configured interrupt handler in the OS runs.

A **timer interrupt** gives OS the ability to run again on a CPU.

Saving and Restoring Context

- **Scheduler** makes a decision:
 - ◆ Whether to continue running the **current process or** switch to a **different one.**
 - ◆ If the decision is made to switch, the OS executes context switch.

Context Switch

- ▣ A low-level piece of assembly code
 - ◆ **Save a few register values** for the current process onto its kernel stack
 - General purpose registers
 - PC
 - kernel stack pointer
 - ◆ **Restore a few** for the soon-to-be-executing process from its kernel stack
 - ◆ **Switch to the kernel stack** for the soon-to-be-executing process

Limited Direction Execution Protocol (Timer interrupt)

OS @ boot
(kernel mode)

Hardware

initialize trap table

remember address 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

...

timer interrupt

save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler

Limited Direction Execution Protocol (Timer interrupt)

OS @ run (kernel mode)	Hardware	Program (user mode)
---------------------------	----------	------------------------

(Cont.)

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

...

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 (pop from stack to mem)  
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 8(%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
```

Current xv6 Code

```
# Context switch
#
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx

    # Save old callee-save registers
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi

    # Switch stacks
    movl %esp, (%eax)
    movl %edx, %esp

    # Load new callee-save registers
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

No actual change to new %eip, because we need to “switch” memory addressing space before (done in the scheduler `switchkvm()`)

In `proc.h`

```
// Don't need to save %eax, %ecx, %edx, because the 46
// x86 convention is that the caller has saved them (I can overwrite them!)
```

Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations:
 - ◆ **Disable interrupts** during interrupt processing
 - ◆ Use several sophisticated **locking** schemes to protect concurrent access to internal data structures.

Scheduling: Introduction

Operating System: Three Easy Pieces

Scheduling: Introduction

- ❑ Workload assumptions:

1. Each job runs for the **same amount of time**.
2. All jobs **arrive** at the same time.
3. All jobs only use the **CPU** (i.e., they perform no I/O).
4. The **run-time** of each job is known.

Single CPU!!

Scheduling Metrics

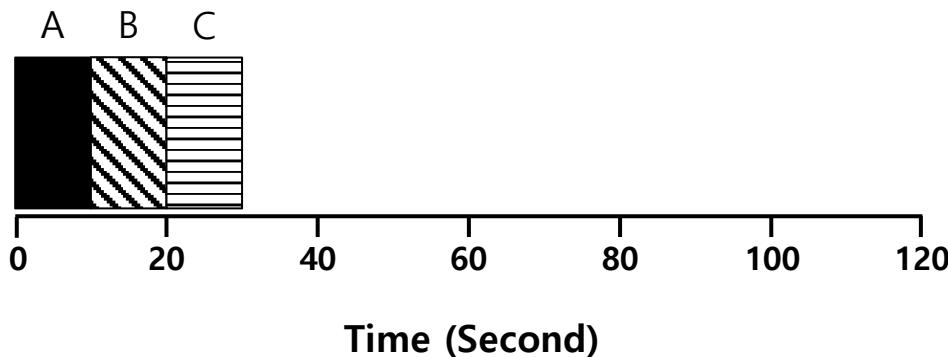
- ❑ Performance metric: Turnaround time
 - ◆ The time at which **the job completes** minus the time at which **the job arrived** in the system.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

- ❑ Another metric is fairness.
 - ◆ Performance and fairness are often at odds in scheduling.

First In, First Out (FIFO)

- ▣ First Come, First Served (FCFS)
 - ◆ Very simple and easy to implement
- ▣ Example:
 - ◆ A arrived just before B which arrived just before C.
 - ◆ Each job runs for 10 seconds.



$$\text{Average turnaround time} = \frac{10 + 20 + 30}{3} = 20 \text{ sec}$$

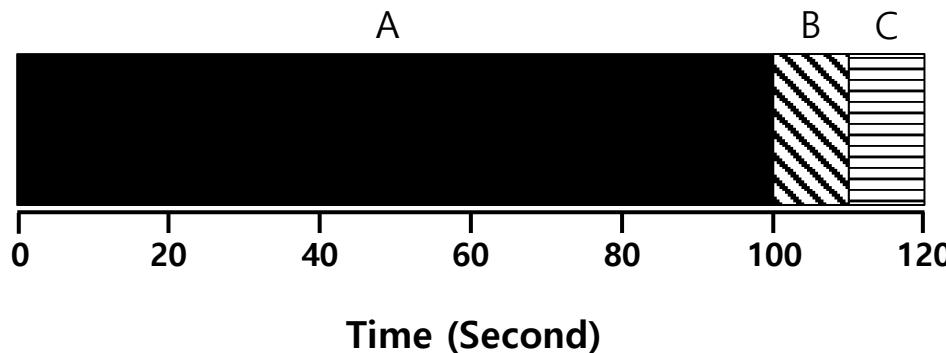
Scheduling: Introduction

- ❑ Workload assumptions:

1. Each job runs for the **same amount of time**.
2. All jobs **arrive** at the same time.
3. All jobs only use the **CPU** (i.e., they perform no I/O).
4. The **run-time** of each job is known.

Why FIFO is not that great? – Convoy effect

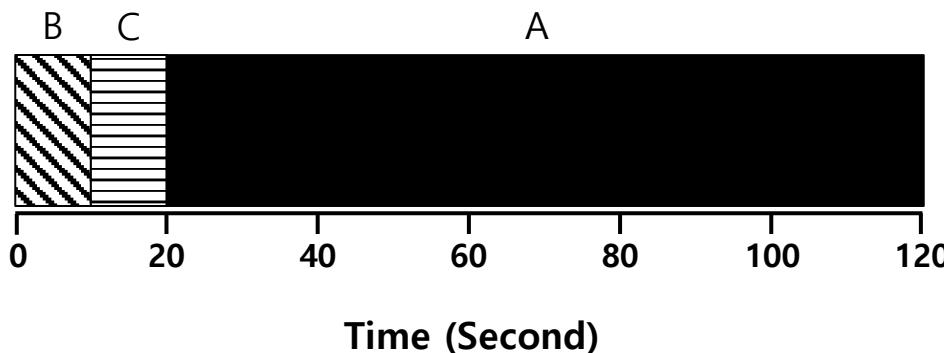
- Let's relax assumption 1: Each job **no longer** runs for the same amount of time.
- Example:
 - A arrived just before B which arrived just before C.
 - A runs for 100 seconds, B and C run for 10 each.



$$\text{Average turnaround time} = \frac{100 + 110 + 120}{3} = 110 \text{ sec}$$

Shortest Job First (SJF)

- ▣ Run the shortest job first, then the next shortest, and so on
 - ◆ Non-preemptive scheduler
- ▣ Example:
 - ◆ A arrived just before B which arrived just before C.
 - ◆ A runs for 100 seconds, B and C run for 10 each.



$$\text{Average turnaround time} = \frac{10 + 20 + 120}{3} = 50 \text{ sec}$$

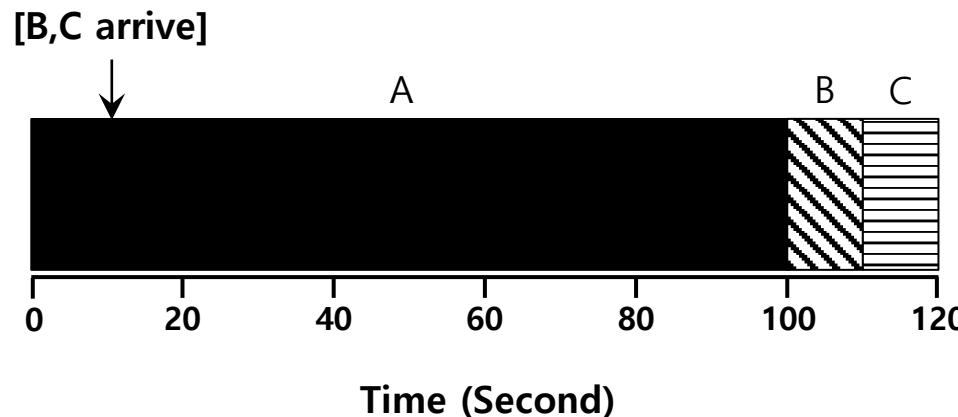
Scheduling: Introduction

- ❑ Workload assumptions:

1. Each job runs for the **same amount of time**.
2. All jobs **arrive** at the same time.
3. All jobs only use the **CPU** (i.e., they perform no I/O).
4. The **run-time** of each job is known.

SJF with Late Arrivals from B and C

- Let's relax assumption 2: Jobs can arrive at any time.
- Example:
 - A arrives at t=0 and needs to run for 100 seconds.
 - B and C arrive at t=10 and each need to run for 10 seconds



$$\text{Average turnaround time} = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33 \text{ sec}$$

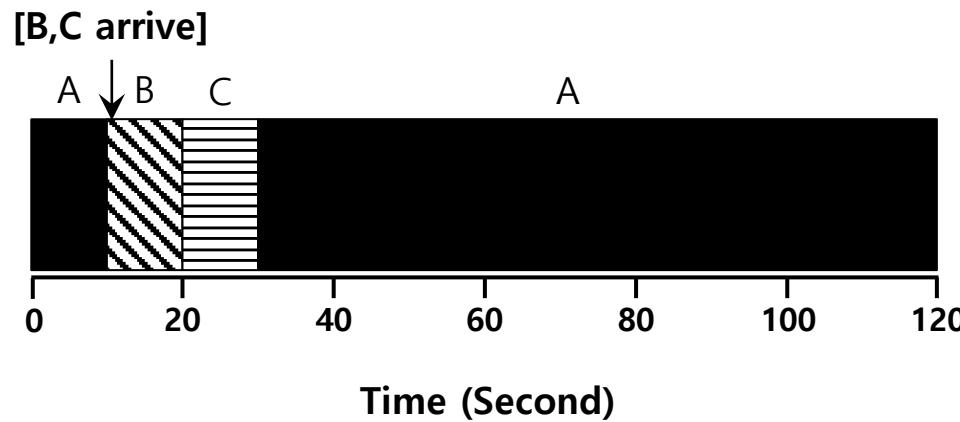
Shortest Time-to-Completion First (STCF)

- ▣ Add **preemption** to SJF
 - ◆ Also known as Preemptive Shortest Job First (PSJF)
- ▣ A new job enters the system:
 - ◆ Determine of the remaining jobs and new job
 - ◆ Schedule the job which has the least time left

Shortest Time-to-Completion First (STCF)

Example:

- ◆ A arrives at t=0 and needs to run for 100 seconds.
- ◆ B and C arrive at t=10 and each need to run for 10 seconds



$$\text{Average turnaround time} = \frac{(120 - 0) + (20 - 10) + (30 - 10)}{3} = 50 \text{ sec}$$

New scheduling metric: Response time

- The time from **when the job arrives** to the **first time it is scheduled**.

$$T_{response} = T_{firstrun} - T_{arrival}$$

- ◆ STCF and related disciplines are not particularly good for response time.

How can we build a scheduler that is
sensitive to response time?

Round Robin (RR) Scheduling

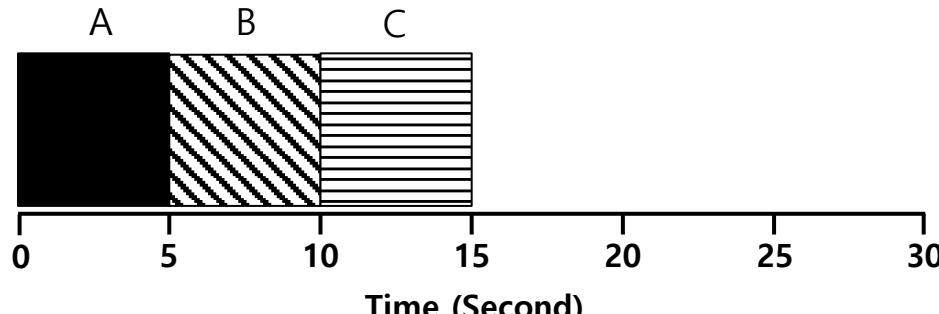
▣ Time slicing Scheduling

- ◆ Run a job for a **time slice** and then switch to the next job in the **run queue** until the jobs are finished.
 - Time slice is sometimes called a scheduling quantum.
- ◆ It repeatedly does so until the jobs are finished.
- ◆ The length of a time slice must be *a multiple of* the timer-interrupt period.

RR is fair, but performs poorly on metrics such as turnaround time

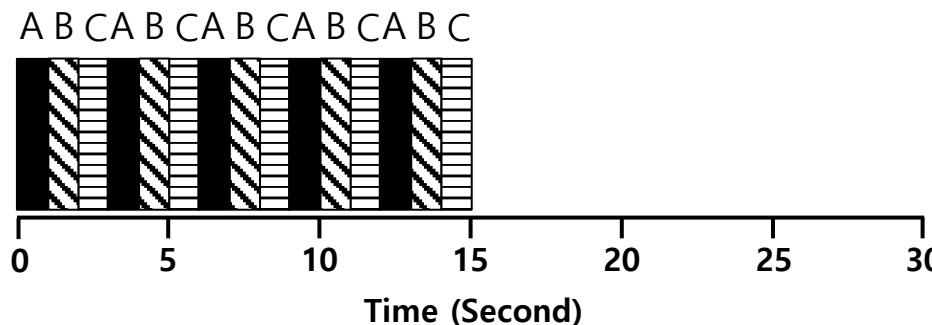
RR Scheduling Example

- ❑ A, B and C arrive at the same time.
- ❑ They each wish to run for 5 seconds.



SJF (Bad for Response Time)

$$T_{average\ response} = \frac{0 + 5 + 10}{3} = 5\ sec$$



RR with a time-slice of 1sec (Good for Response Time)

$$T_{average\ response} = \frac{0 + 1 + 2}{3} = 1\ sec$$

The length of the time slice is critical.

- ▣ The shorter time slice
 - ◆ Better response time
 - ◆ The cost of context switching will dominate overall performance.

- ▣ The longer time slice
 - ◆ Amortize the cost of switching
 - ◆ Worse response time

**Deciding on the length of the time slice presents
a **trade-off** to a system designer**

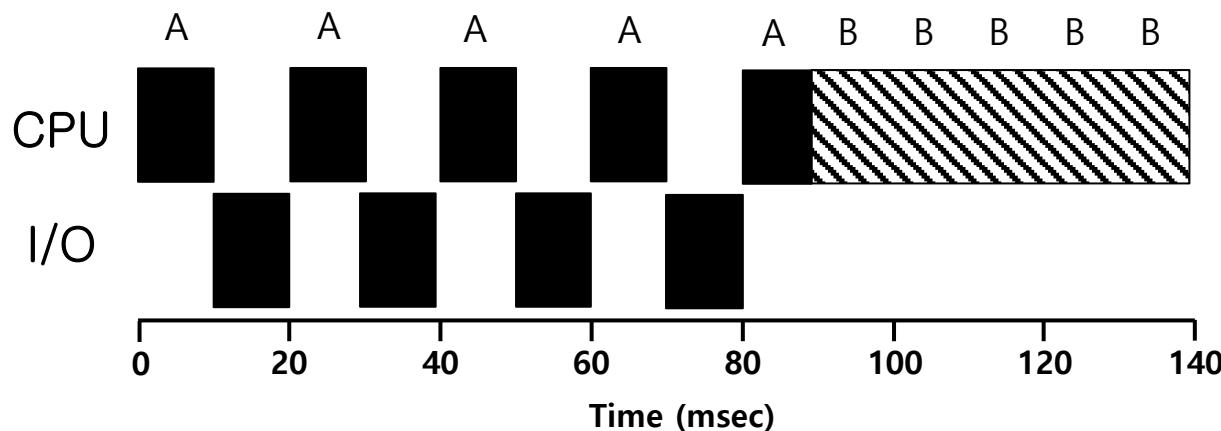
Scheduling: Introduction

- Workload assumptions:
 1. Each job runs for the **same amount of time**.
 2. All jobs **arrive** at the same time.
 3. All jobs only use the **CPU** (i.e., they perform no I/O).
 4. The **run-time** of each job is known.

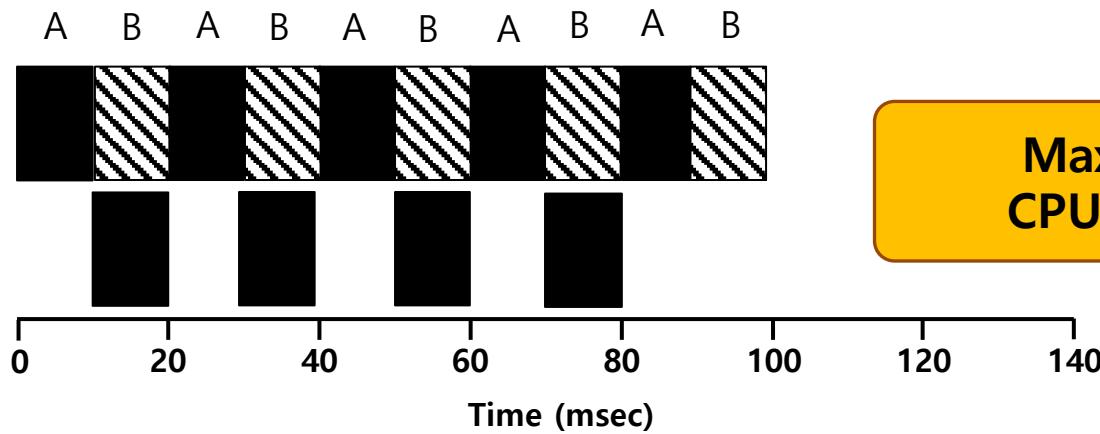
Incorporating I/O

- ▣ Let's relax assumption 3: All programs perform I/O
- ▣ Example:
 - ◆ A and B need 50ms of CPU time each.
 - ◆ A runs for 10ms and then issues an I/O request
 - I/Os each take 10ms
 - ◆ B simply uses the CPU for 50ms and performs no I/O
 - ◆ The scheduler runs A first, then B after

Incorporating I/O (Cont.)



Poor Use of Resources



**Maximize the
CPU utilization**

Overlap Allows Better Use of Resources

Incorporating I/O (Cont.)

- ▣ When a job initiates an I/O request.
 - ◆ The job is blocked waiting for I/O completion.
 - ◆ The scheduler should schedule another job on the CPU.
- ▣ When the I/O completes
 - ◆ An interrupt is raised.
 - ◆ The OS moves the process from blocked back to the ready state.
- ▣ No more Oracle (assumption 4)
 - ◆ In general purpose, the OS knows very little about the length of each job.
How can build SJF/STCF without a priori knowledge?

Scheduling: Proportional Share

Operating System: Three Easy Pieces

Proportional Share Scheduler

- ▣ **Fair-share** scheduler

- ◆ Guarantee that each job obtain *a certain percentage* of CPU time.
- ◆ Not optimized for turnaround or response time

Basic Concept

❑ Tickets

- ◆ Represent the share of a resource that a process should receive
- ◆ The percent of tickets represents its share of the system resource in question.

❑ Example

- ◆ There are two processes, A and B.
 - Process A has 75 tickets → receive 75% of the CPU
 - Process B has 25 tickets → receive 25% of the CPU

Lottery scheduling

- ▣ The scheduler picks a winning ticket.
 - ◆ Load the state of that *winning process* and runs it.
- ▣ Example
 - ◆ There are 100 tickets
 - Process A has 75 tickets: 0 ~ 74
 - Process B has 25 tickets: 75 ~ 99

Scheduler's winning tickets: 63 85 70 39 76 17 29 41 36 39 10 99 68 83 63

Resulting scheduler: A B A A B A A A A A B A B A

The longer these two jobs compete,
The more likely they are to achieve the desired percentages.

The beauty of randomness (in scheduling)

- ▣ Deals easily with corner-case situations
 - ◆ Others should have a lot of "ifs" to prevent them
- ▣ Little state required
 - ◆ No need to track the details of each process in the past
- ▣ Really fast
 - ◆ More speed → more pseudo-randomness (or HW assistance)

Ticket Mechanisms

- ▣ Ticket currency
 - ◆ A user allocates tickets among their own jobs in whatever currency they would like.
 - ◆ The system converts the currency into the correct global value.
 - ◆ Example
 - There are 200 tickets (Global currency)
 - Process A has 100 tickets
 - Process B has 100 tickets
- User A** → 500 (A's currency) to A1 → 50 (global currency)
→ 500 (A's currency) to A2 → 50 (global currency)
- User B** → 10 (B's currency) to B1 → 100 (global currency)

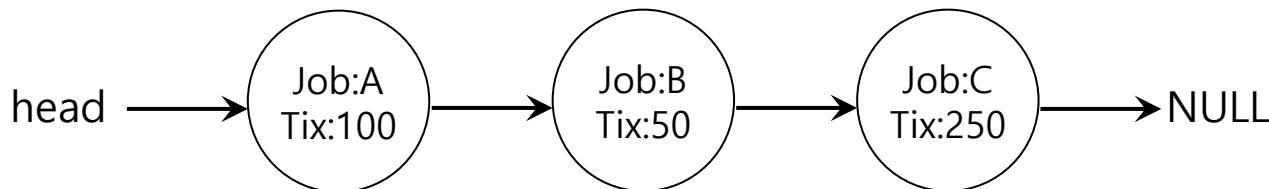
Ticket Mechanisms (Cont.)

- ▣ Ticket transfer
 - ◆ A process can temporarily hand off its tickets to another process.
- ▣ Ticket inflation
 - ◆ A process can temporarily raise or lower the number of tickets it owns.
 - ◆ If any one process needs *more CPU time*, it can boost its tickets.

Implementation

- Example: There are three processes, A, B, and C.

- Keep the processes in a list:



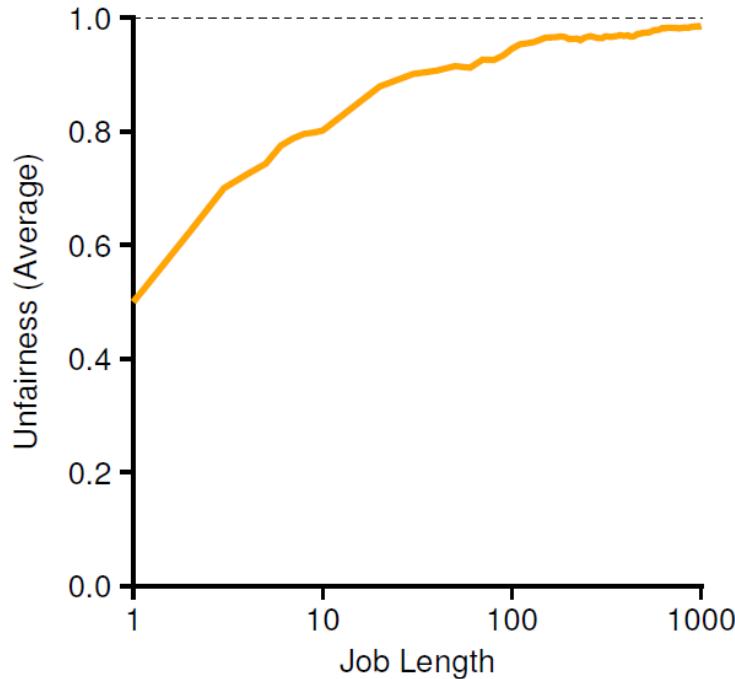
```
1 // counter: used to track if we've found the winner yet
2 int counter = 0;
3
4 // winner: use some call to a random number generator to
5 // get a value, between 0 and the total # of tickets
6 int winner = getrandom(0, totaltickets);
7
8 // current: use this to walk through the list of jobs
9 node_t *current = head;
10
11 // loop until the sum of ticket values is > the winner
12 while (current) {
13     counter = counter + current->tickets;
14     if (counter > winner)
15         break; // found the winner
16     current = current->next;
17 }
18 // 'current' is the winner: schedule it...
```

Implementation (Cont.)

- ▣ U: unfairness metric
 - ◆ The time the first job completes divided by the time that the second job completes.
- ▣ Example:
 - ◆ There are two jobs, each job has runtime 10.
 - First job finishes at time 10
 - Second job finishes at time 20
 - ◆ $U = \frac{10}{20} = 0.5$
 - ◆ U will be close to 1 when both jobs finish at nearly the same time.

Lottery Fairness Study

- There are two jobs.
 - ◆ Each job has the same number of tickets (100).



When the job length is not very long,
average unfairness can be quite severe.

Stride Scheduling (deterministic Fair-share scheduler)

- **Stride** of each process

- ◆ Defined as (one large number) / (the number of tickets of the process)
- ◆ Example: one large number = 10,000
 - Process A has 100 tickets → stride of A is 100
 - Process B has 50 tickets → stride of B is 200
 - Process C has 250 tickets → stride of C is 40

- A process runs, increment a counter(=pass value) for it by its stride.
- ◆ Pick the process to run that has **the lowest pass value**

```
current = remove_min(queue);           // pick client with minimum pass
schedule(current);                   // use resource for quantum
current->pass += current->stride;    // compute next pass using stride
insert(queue, current);              // put back into the queue
```

A pseudo code implementation

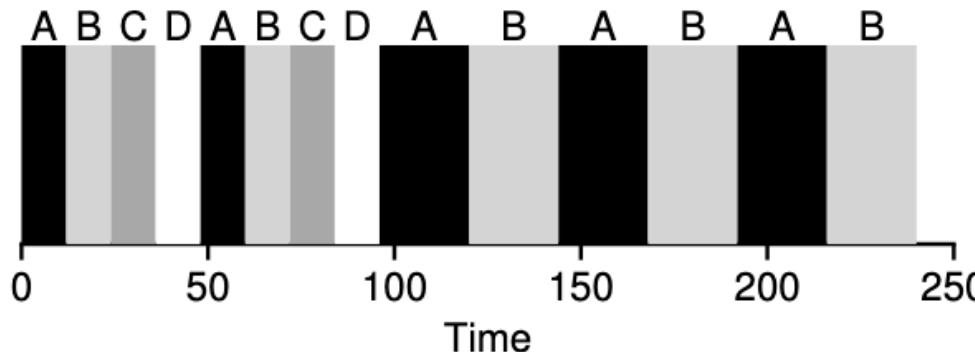
Stride Scheduling Example

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

If new job enters with pass value 0,
It will **monopolize** the CPU!:
Stride scheduler requires global state (in
contrast with lottery)

Linux Completely Fair Scheduler (CFS)

- ▣ 5% overall datacenter CPU time wasted in scheduler
- ▣ Focus on fairness and minimizing scheduling overhead
 - ◆ Keep track of the (virtual) runtime of each process
 - ◆ At scheduling time, **chooses** the process with the **lowest virtual run time**
 - ◆ Time slice is variable: according number of ready to run processes (from **sched_latency** (48 ms) for 1 process to **min_granularity** (6 ms) for a ny number of processes



Niceness and Weights

- Time slice and virtual runtime of the process can be affected by weights (niceness)

$$\text{time_slice}_k = \frac{\text{weight}_k}{\sum_{n=0}^{n-1} \text{weight}_i} \cdot \text{sched_latency}$$

- Typically, from -20 to 19
- By default, 0

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */ 9548, 7620, 6100, 4904, 3906,
    /* -5 */ 3121, 2501, 1991, 1586, 1277,
    /* 0 */ 1024, 820, 655, 526, 423,
    /* 5 */ 335, 272, 215, 172, 137,
    /* 10 */ 110, 87, 70, 56, 45,
    /* 15 */ 36, 29, 23, 18, 15,
};
```

Example

- Two process: A (niceness -5) B (niceness 0)
- $\text{weight}_A = 3121$, $\text{weight}_B = 1024$
- $\text{time_slice}_A = 3/4$ (36 ms), $\text{time_slice}_B = 1/4$ (12 ms) of **sched_latency** (48 ms)
- Also, virtual runtime changes with weights

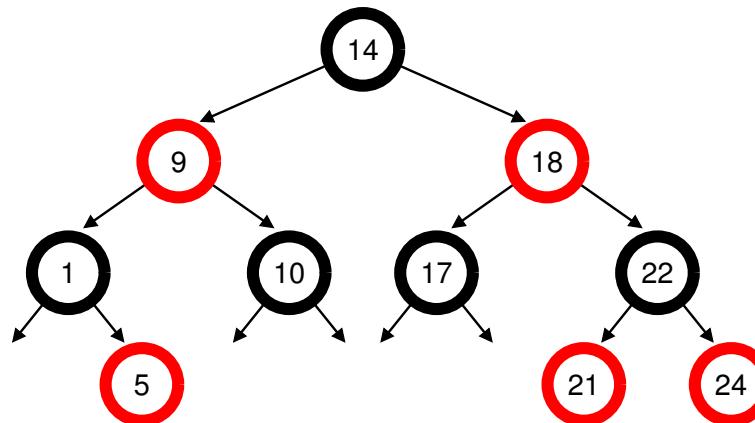
$$\text{vruntime}_i = \text{vruntime}_i + \frac{\text{weight}_0}{\text{weight}_i} \cdot \text{runtime}_i$$

- A accumulates virtual runtime at 1/3 of B

Linux Completely Fair Scheduler (CFS)

- Minimal scheduler overhead

- Data structures should be scalable: No lists
- CFS uses a balanced tree (red-black tree) of the ready-to-run processes
- $O(\log n)$ insertions and searches



- Many more details

- I/O is handled faking the `vruntime` of the awoken process to the minimum value in the tree
- Heuristics for multi-CPU scheduling (cache affinity, frequency, core complexity...)
- Cooperative multi-process schedule
- ... next

The Abstraction: Address Space

Operating System: Three Easy Pieces

Memory Virtualization

❑ What is **memory virtualization**?

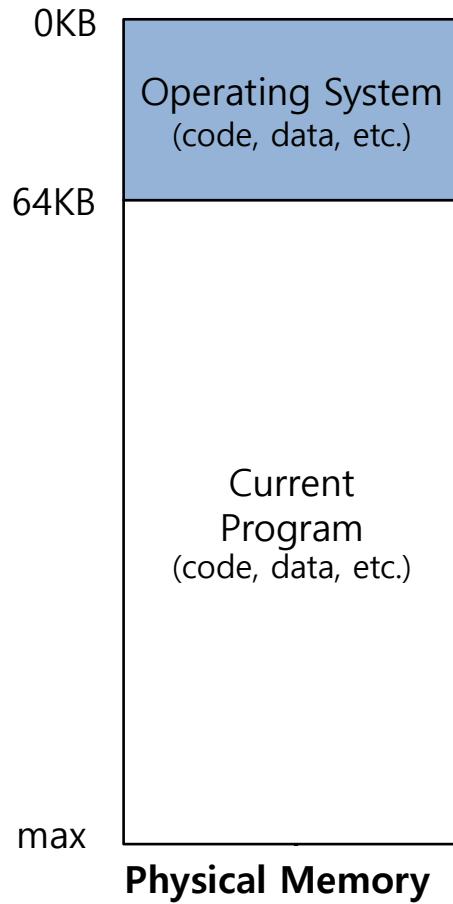
- ◆ OS virtualizes its physical memory.
- ◆ OS provides an **illusion memory space** per each process.
- ◆ It seems to be seen like **each process uses the whole memory** .

Benefit of Memory Virtualization

- ▣ Ease of use in programming
- ▣ Memory efficiency in terms of time and space
- ▣ The guarantee of isolation for processes as well as OS
 - ◆ Protection from **errant accesses** of other processes

OS in The Early System

- Load only one process in memory.
 - ◆ Poor utilization and efficiency



Perhaps in the future
too? -- Unikernels

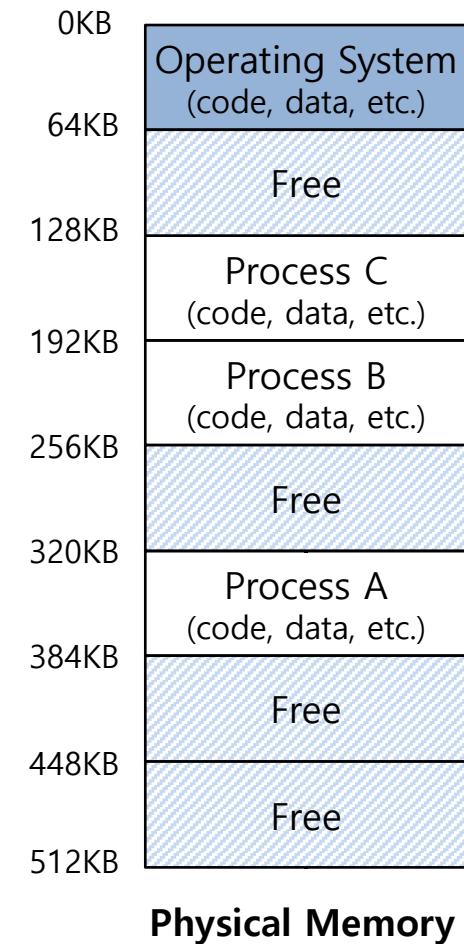
Multiprogramming and Time Sharing

- **Load multiple processes** in memory.

- ◆ Execute one for a short while.
- ◆ Switch processes between them in memory.
- ◆ Increase utilization and efficiency.

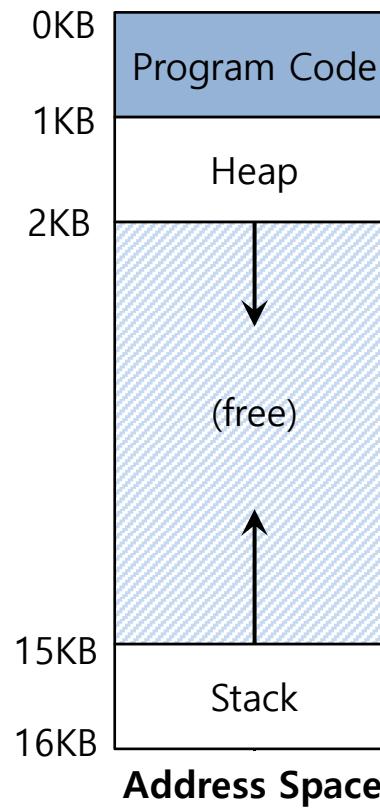
- Cause an important **protection issue**.

- ◆ Errant memory accesses from other processes



Address Space

- OS creates an **abstraction** of physical memory.
 - The address space contains all about a running process.
 - That is consist of program code, heap, stack and etc.



▽

Address Space(Cont.)

□ Program

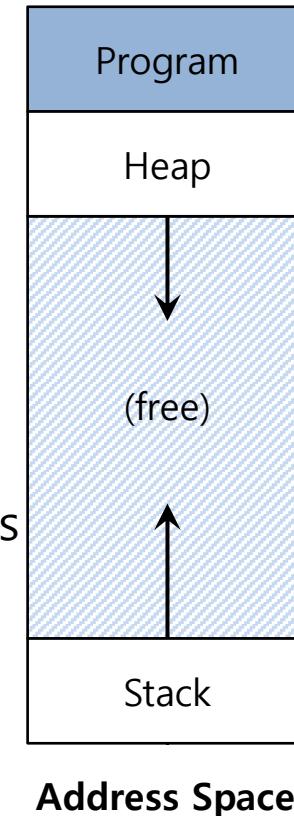
- ◆ Where instructions and static data live

□ Heap

- ◆ Dynamically allocate memory.
 - malloc/free in C language
 - new/delete in object-oriented language
- ◆ Implicitly handled in (memory) managed languages

□ Stack

- ◆ Store return addresses or values.
- ◆ Contain local variables arguments to routines.
- ◆ Implicitly handled in HLL



Address Space

Virtual Address

- Every address in a running program is virtual.

- OS provides the “mechanism” to translate the virtual address to physical address
- HW assists to make such translation “painless”

```
#include <stdio.h>
#include <stdlib.h>
int global=1;
int main(int argc, char *argv[])
{
    int x = 3;
    printf("location of code : %p\n", (void *) main);
    printf("location of data : %p\n", (void *) &global);
    printf("location of heap : %p\n", (void *) malloc(1));
    printf("location of stack : %p\n", (void *) &x);

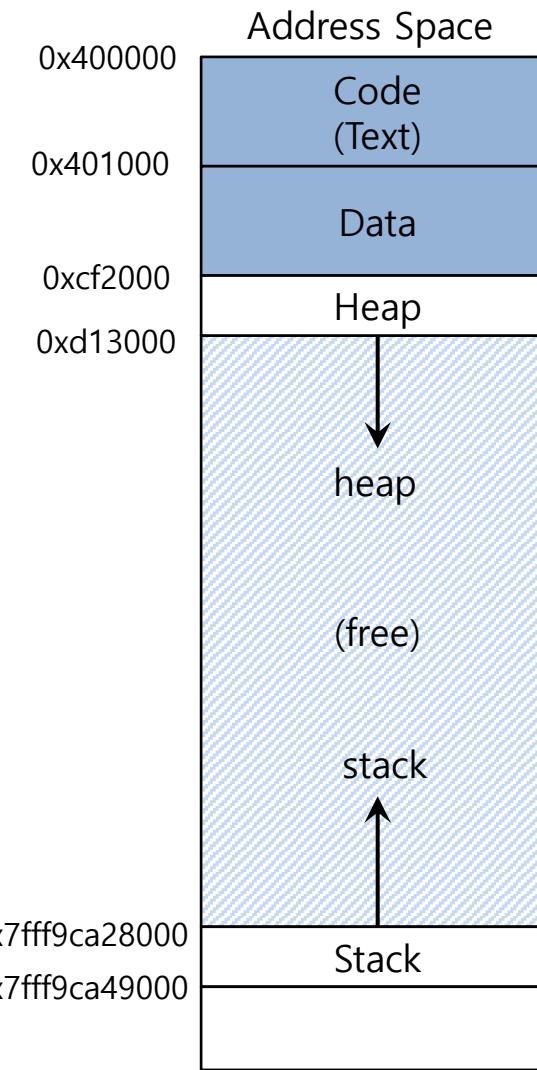
    return x;
}
```

A simple program that prints out addresses

Virtual Address(Cont.)

□ The output in 64-bit Linux machine

```
location of code   : 0x40057d  
location of data   : 0x401010  
location of heap    : 0xcf2010  
location of stack   : 0x7fff9ca45fcc
```



Goals of VM

- ▣ Transparency
 - ◆ VM should be invisible to running program
- ▣ Efficiency
 - ◆ Minimize overhead in terms of speed and space
- ▣ Protection
 - ◆ Isolate processes (and OS itself) [but allowing selective "communication"]

Paging: Introduction

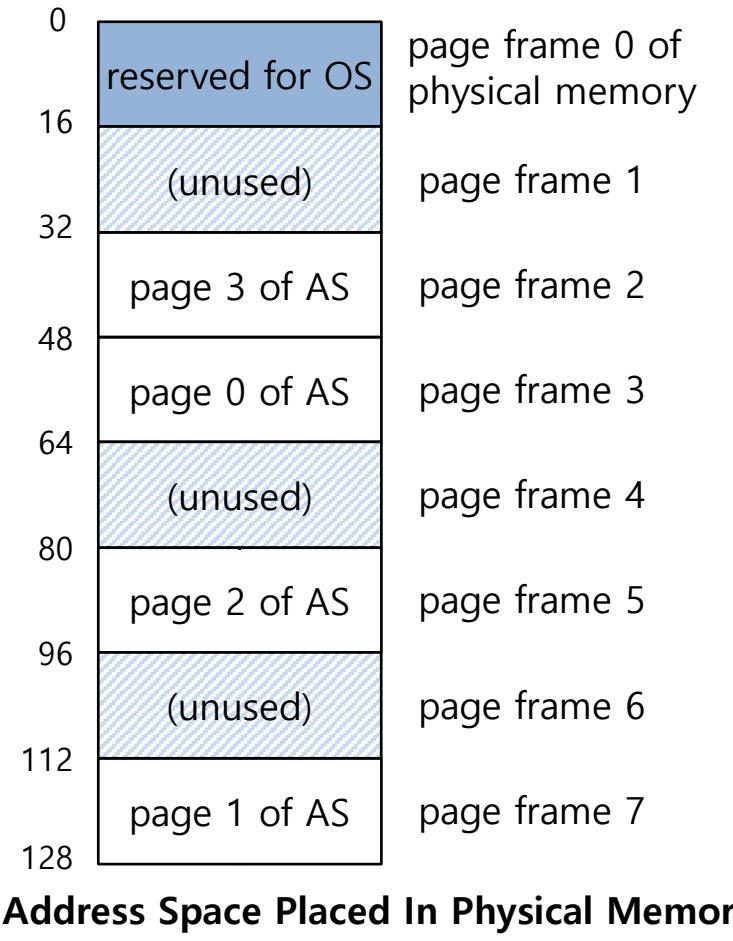
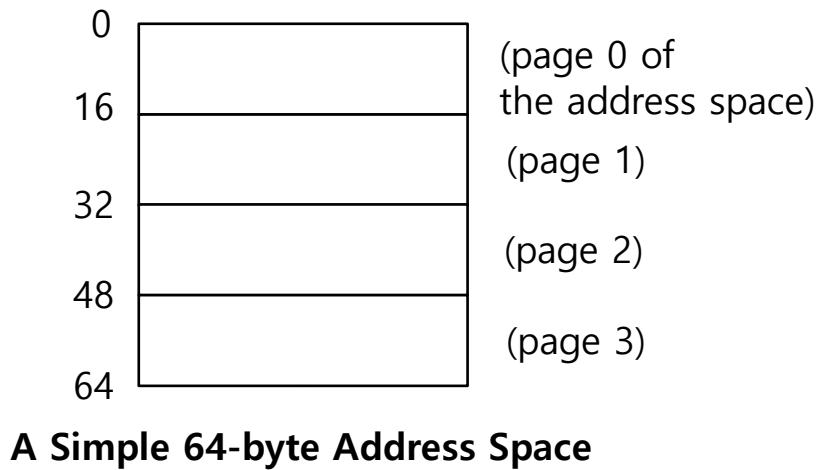
Operating System: Three Easy Pieces

Concept of Paging

- ▣ Paging **splits up** address space into **fixed-sized** unit called a **page**.
 - ◆ Segmentation: variable size of logical segments(code, stack, heap, etc.)
- ▣ With paging, **physical memory** is also **split** into some number of pages called a **page frame**.
- ▣ **Page table** per process is needed **to translate** the virtual address to physical address.

Toy Example: A Simple Paging

- 128-byte physical memory with 16 bytes page frames
- 64-byte address space with 16 bytes pages, 1 byte addressable

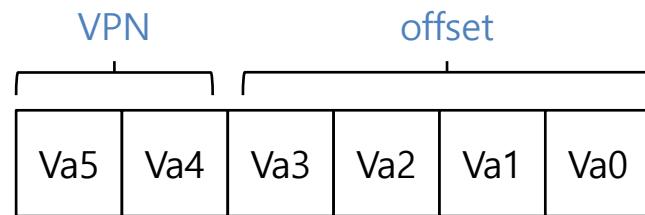


Advantages Of Paging

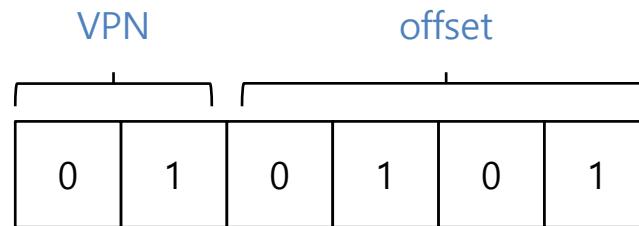
- ▣ **Flexibility:** Supporting the abstraction of address space effectively
 - ◆ Don't need assumption how heap and stack grow and are used.
- ▣ **Simplicity:** ease of free-space management
 - ◆ The page in address space and the page frame are the same size.
 - ◆ Easy to allocate and keep a free list
- ▣ *Con: the semantic meaning of the content is lost...*

Address Translation

- Two components in the virtual address
 - VPN: virtual page number
 - Offset: offset within the page



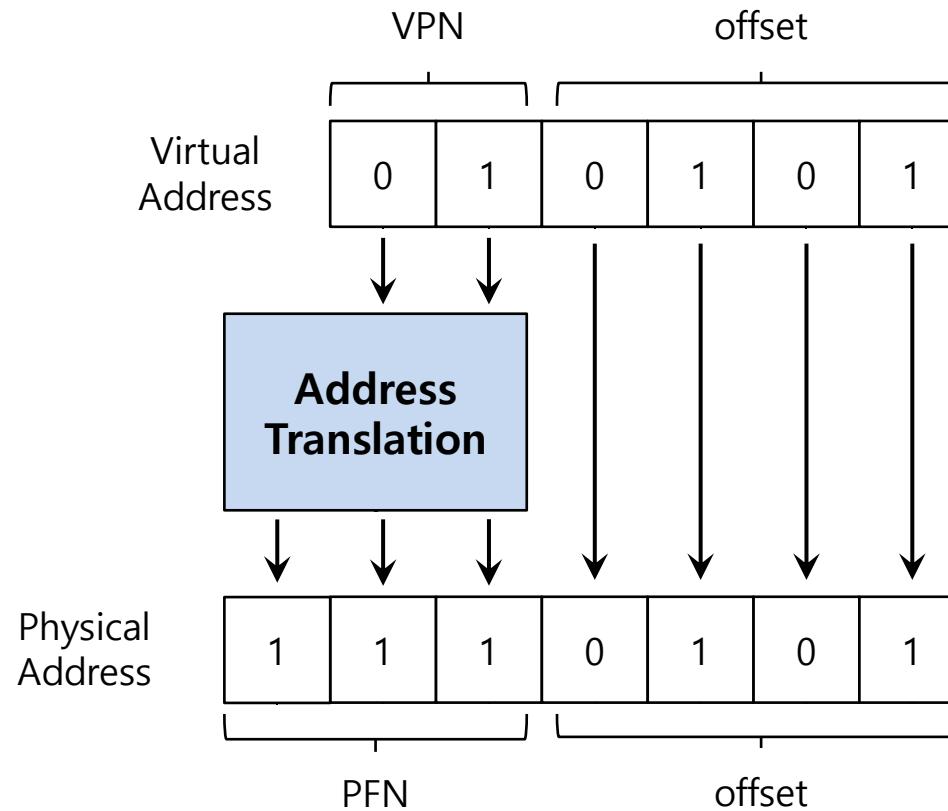
- Example: virtual address 21 in 64-byte address space



`movl 21, %eax`

Example: Address Translation

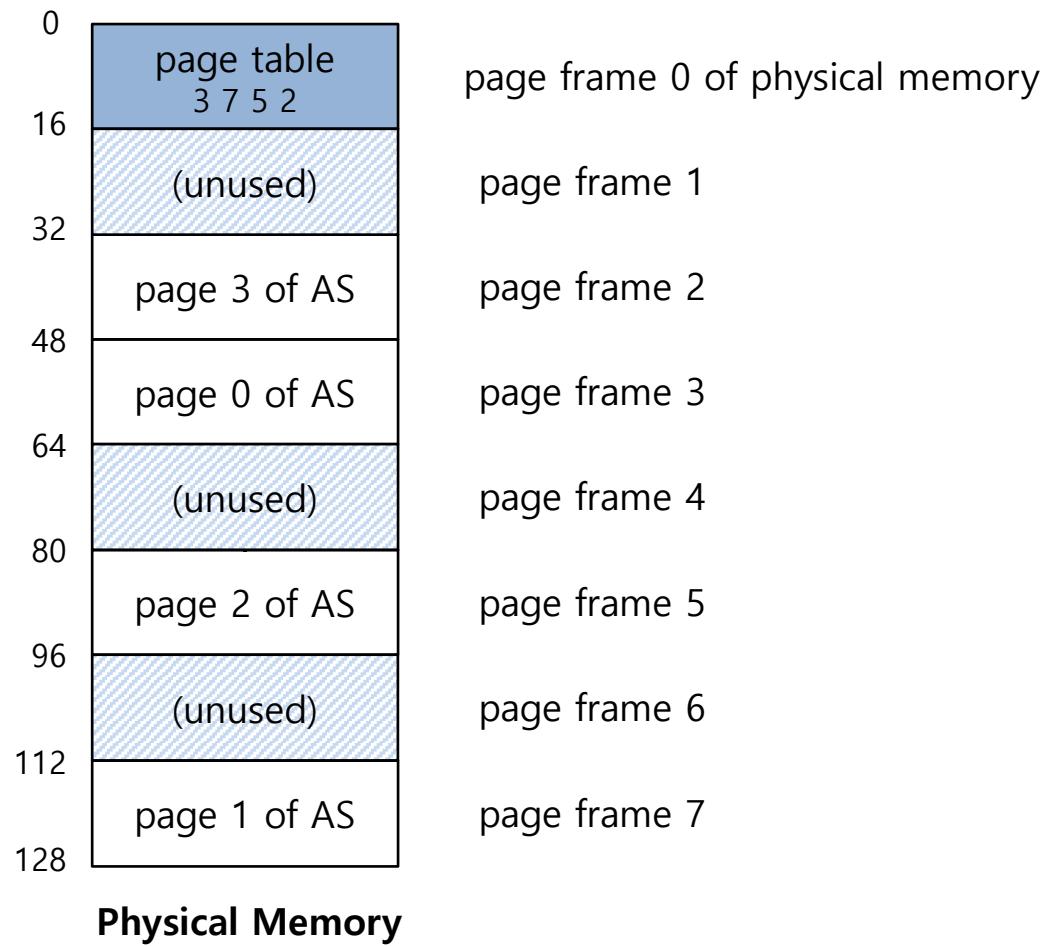
- The virtual address 21 in 64-byte address space



Where Are Page Tables Stored?

- ▣ Page tables can get awfully large
 - ◆ 32-bit address space with 4-KB pages, 20 bits for VPN
 - $4MB = 2^{20} \text{ entries} * 4 \text{ Bytes per page table entry}$
- ▣ Page tables **for each process** are stored in memory.

Example: Page Table in Kernel Physical Memory



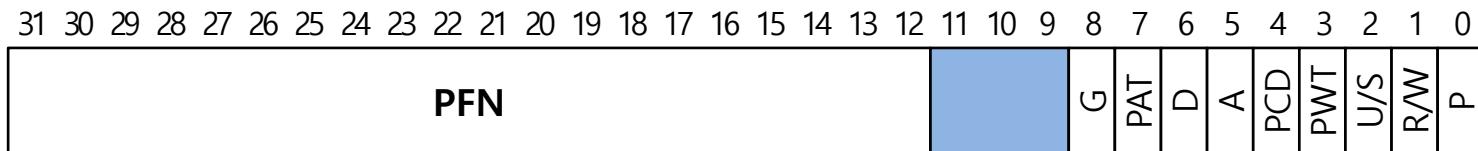
What Is In The Page Table?

- ▣ The page table is just a **data structure** that is used to map the virtual address to physical address.
 - ◆ Simplest form: a linear page table, an array
- ▣ The OS **indexes** the array by VPN, and looks up the page-table entry.

Common Flags Of Page Table Entry (PTE)

- ▣ **Valid Bit:** Indicating whether the particular translation is valid.
- ▣ **Protection Bit:** Indicating whether the page could be read from, written to, or executed from
- ▣ **Present Bit:** Indicating whether this page is in physical memory or on disk(swapped out)
- ▣ **Dirty Bit:** Indicating whether the page has been modified since it was brought into memory
- ▣ **Reference Bit(Accessed Bit):** Indicating that a page has been accessed

Example: x86 Page Table Entry



- ❑ P: present An x86 Page Table Entry(PTE)
- ❑ R/W: read/write bit
- ❑ U/S: supervisor
- ❑ A: accessed bit
- ❑ D: dirty bit
- ❑ PFN: the page frame number
- ❑ (PWT, PCD, PAT, and G): how hardware caching should work there
- ❑ Why no Valid bit?

Paging: Too Slow

- ▣ To find a location of the desired PTE, the **starting location** of the page table is **needed**.
 - ◆ Page tables are too big to be stored in MMU
- ▣ For every memory reference, paging requires the OS to perform one **extra memory reference**.

Accessing Memory With Paging

```
1 // Extract the VPN from the virtual address SHIFT=4  VPN_MASK=0x30
2 VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3 // Form the address of the page-table entry (PTE)
4 PTEAddr = PTBR + (VPN * sizeof(PTE))
5
6
7 // Fetch the PTE ~PTBR[VPN]
8 PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False):
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False):
14     RaiseException(PROTECTION_FAULT)
15 else:
16     // Access is OK: form physical address and fetch it
17     offset = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

A Memory Trace

- Example: A Simple Memory Access

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

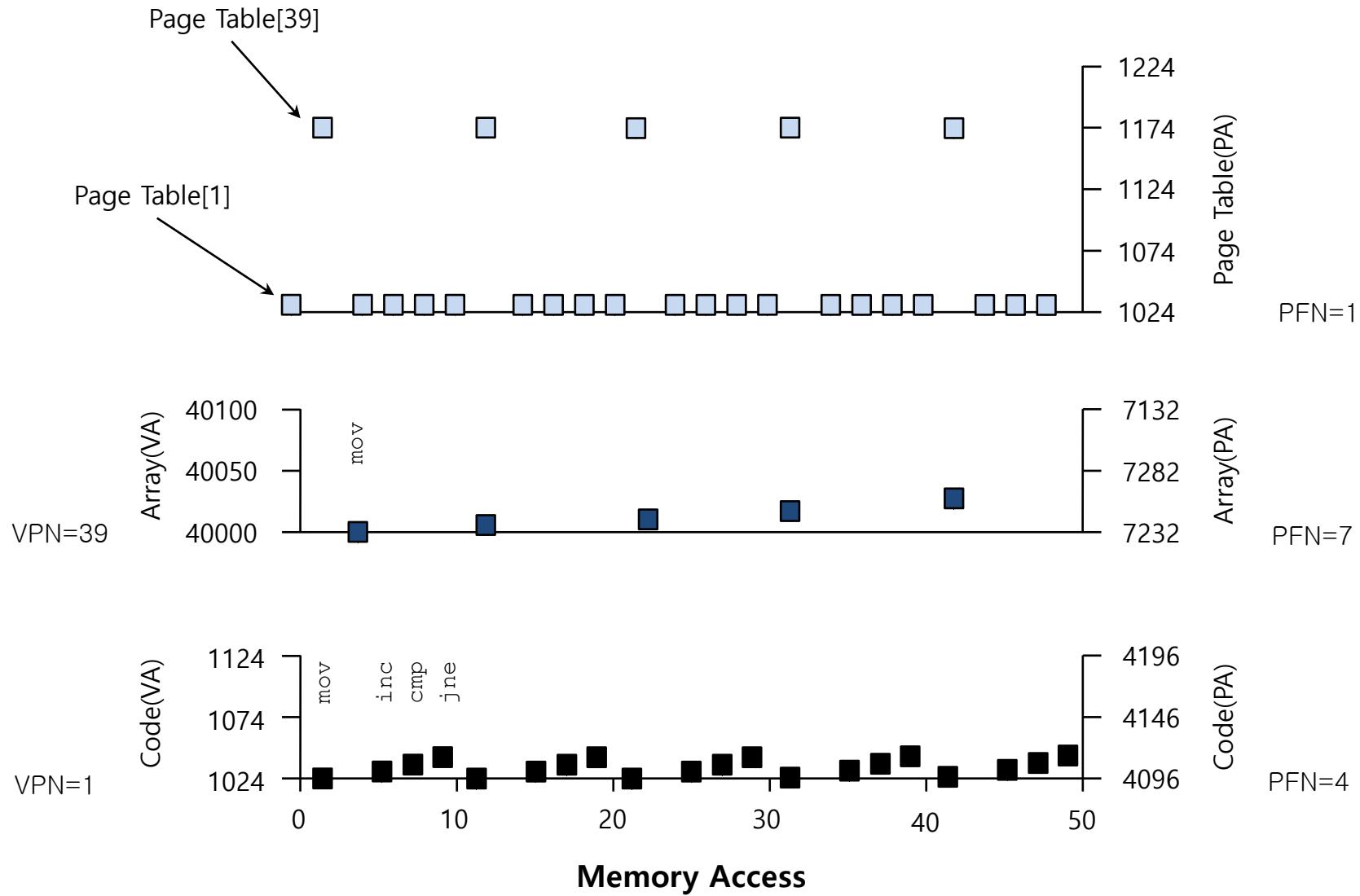
- Compile and execute

```
prompt> gcc -o array array.c -Wall -o
prompt>./array
```

- Resulting Assembly code (`objdump -d`)

```
0x1024 movl $0x0, (%edi,%eax,4)    #Power of CISC! edi+eax*4
0x1028 incl %eax                   #Increase counter
0x102c cmpl $0x03e8,%eax          #Check if last element
0x1030 jne 0x1024                 #Implicit (eflags) Zero bit access
```

A Virtual(And Physical) Memory Trace: 64KB VAS, 1KB pages

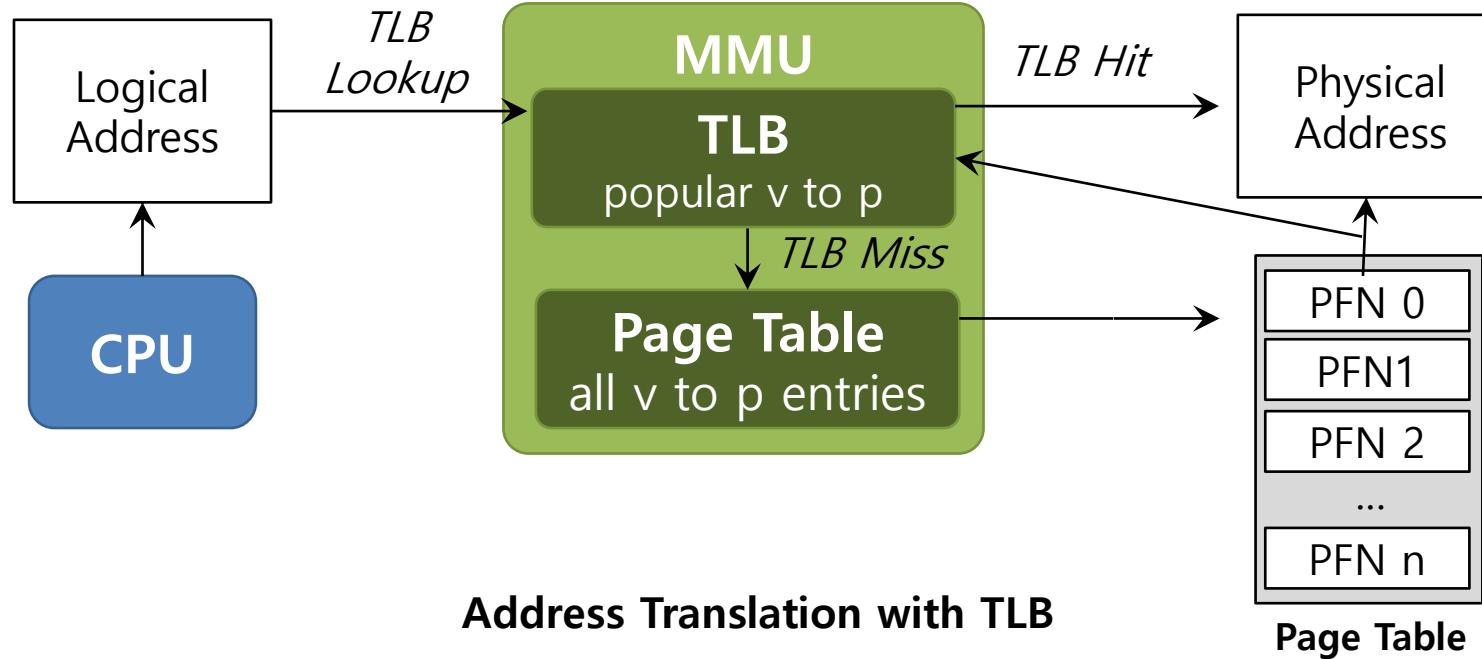


Translation Lookaside Buffers

Operating System: Three Easy Pieces

TLB

- Part of the chip's memory-management unit(MMU).
- A hardware cache of **popular** virtual-to-physical address translation.



TLB Basic Algorithms

```
1: VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2: (Success, TlbEntry) = TLB_Lookup(VPN)
3: if(Success == True){ // TLB Hit
4:     if(CanAccess(TlbEntry.ProtectBit) == True){
5:         offset = VirtualAddress & OFFSET_MASK
6:         PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:         AccessMemory(PhysAddr)
8:     }else RaiseException(PROTECTION_ERROR)
```

- ◆ (1 lines) extract the virtual page number (VPN).
- ◆ (2 lines) check if the TLB holds the translation for this VPN.
- ◆ (5-8 lines) extract the page frame number from the relevant TLB entry, and form the desired physical address and access memory.

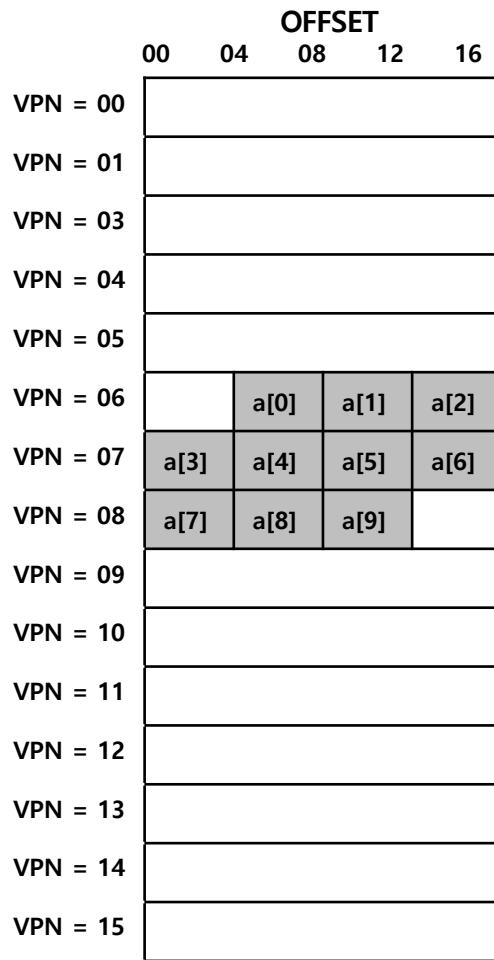
TLB Basic Algorithms (Cont.)

```
11:     }else{ //TLB Miss  
12:         PTEAddr = PTBR + (VPN * sizeof(PTE))  
13:         PTE = AccessMemory(PTEAddr)  
14:         if (PTE.Valid == False) {  
15:             RaiseException(SEGMENTATION_FAULT)  
16:         }else if (CanAccess(PTE.ProtectBits) == False) {  
17:             RaiseException(PROTECTION_FAULT)  
18:         }else{  
19:             TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)  
20:             RetryInstruction() }  
21:     }  
22: }
```

- ◆ (11-13 lines) Accesses the page table to find the translation.
- ◆ (13-17 lines) Check if PTE is ok
- ◆ (19 lines) updates the TLB with the translation.

Example: Accessing An Array

- How a TLB can improve its performance.



```
0:     int sum = 0 ;  
1:     for( i=0; i<10; i++) {  
2:             sum+=a[i] ;  
3:     }
```

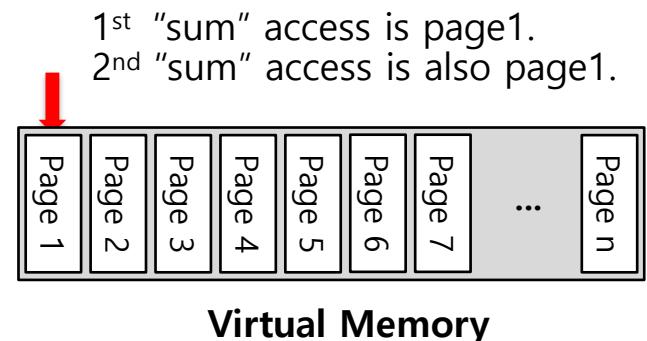
The TLB improves performance
due to spatial locality

3 misses and 7 hits.
Thus, **TLB hit rate** is 70%.

Locality

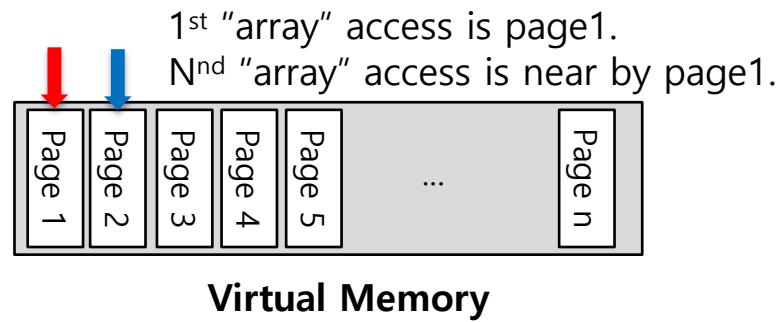
Temporal Locality

- An instruction or data item that has been recently accessed will likely be re-accessed soon in the future.



Spatial Locality

- If a program accesses memory at address x, it will likely soon access memory near x.



Who Handles The TLB Miss?

- ▣ Hardware handle the TLB miss entirely on **CISC**.
 - ◆ The hardware has to know exactly where the page tables are located in memory.
 - ◆ The hardware would “walk” the page table (TLB walker), find the correct page-table entry and **extract** the desired translation, **update** and **retry** instruction. This is the previous algorithm.
 - ◆ **hardware-managed TLB**.

Who Handles The TLB Miss? (Cont.)

- ▣ **RISC** have what is known as a software-managed TLB.
 - ◆ On a TLB miss, the hardware raises exception(trap handler).
 - Trap handler is code within the OS that is written with the express purpose of **handling TLB miss**.

TLB Control Flow algorithm(OS Handled)

```
1:      VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2:      (Success, TlbEntry) = TLB_Lookup(VPN)
3:      if (Success == True) { // TLB Hit
4:          if (CanAccess(TlbEntry.ProtectBits) == True) {
5:              Offset = VirtualAddress & OFFSET_MASK
6:              PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:              Register = AccessMemory(PhysAddr) }
8:          else{
9:              RaiseException(PROTECTION_FAULT) } }
10:     else { // TLB Miss
11:         RaiseException(TLB_MISS) }
```

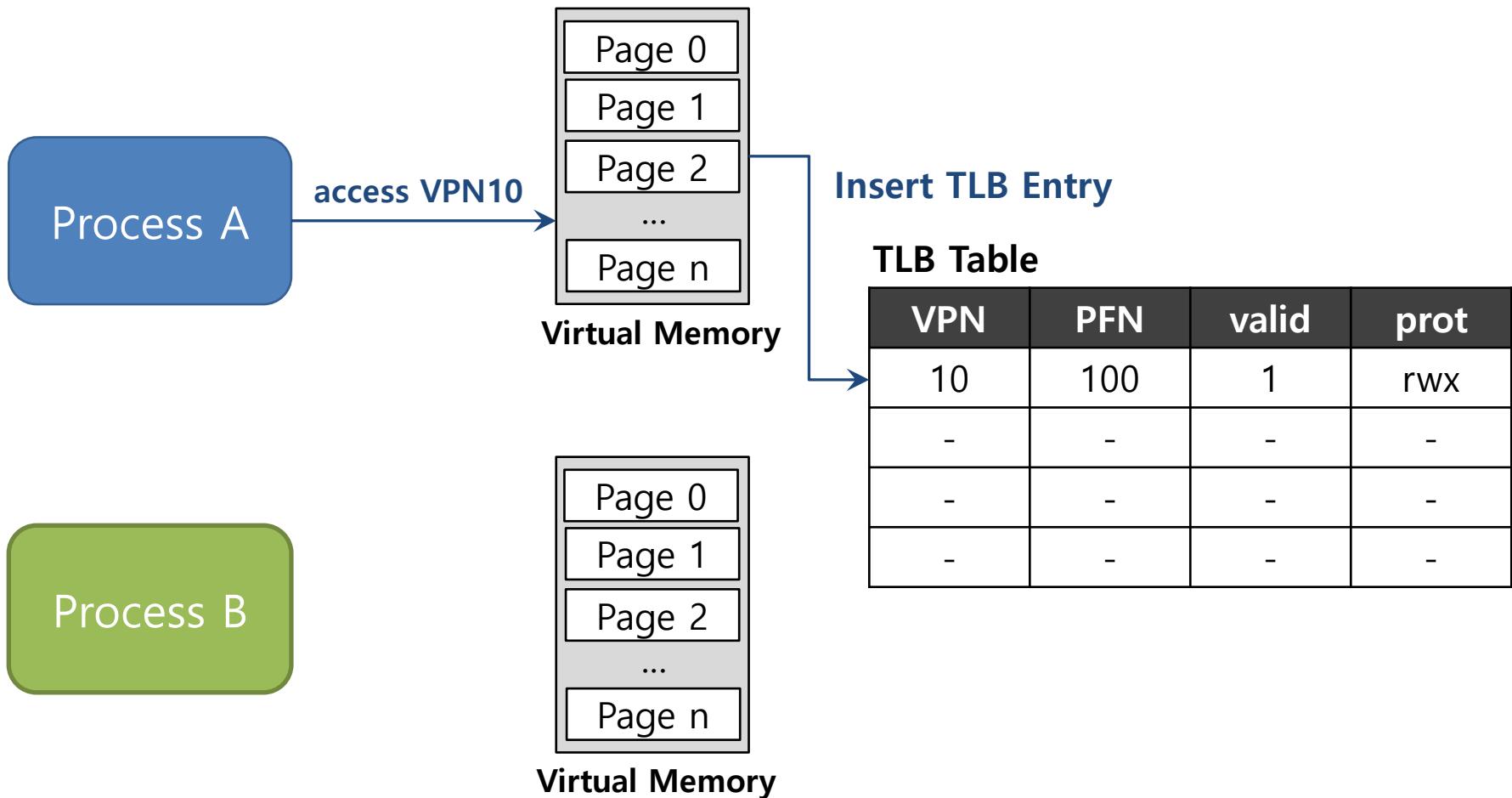
TLB entry

- TLB is managed by **Full Associative** method.
 - ◆ A typical TLB might have 32 to 128 entries.
 - ◆ Hardware search the entire TLB in parallel to find the desired translation (i.e., a cache without index bit, i.e., tag=VPN)
 - ◆ other bits: valid bits , protection bits, address-space identifier, dirty bit

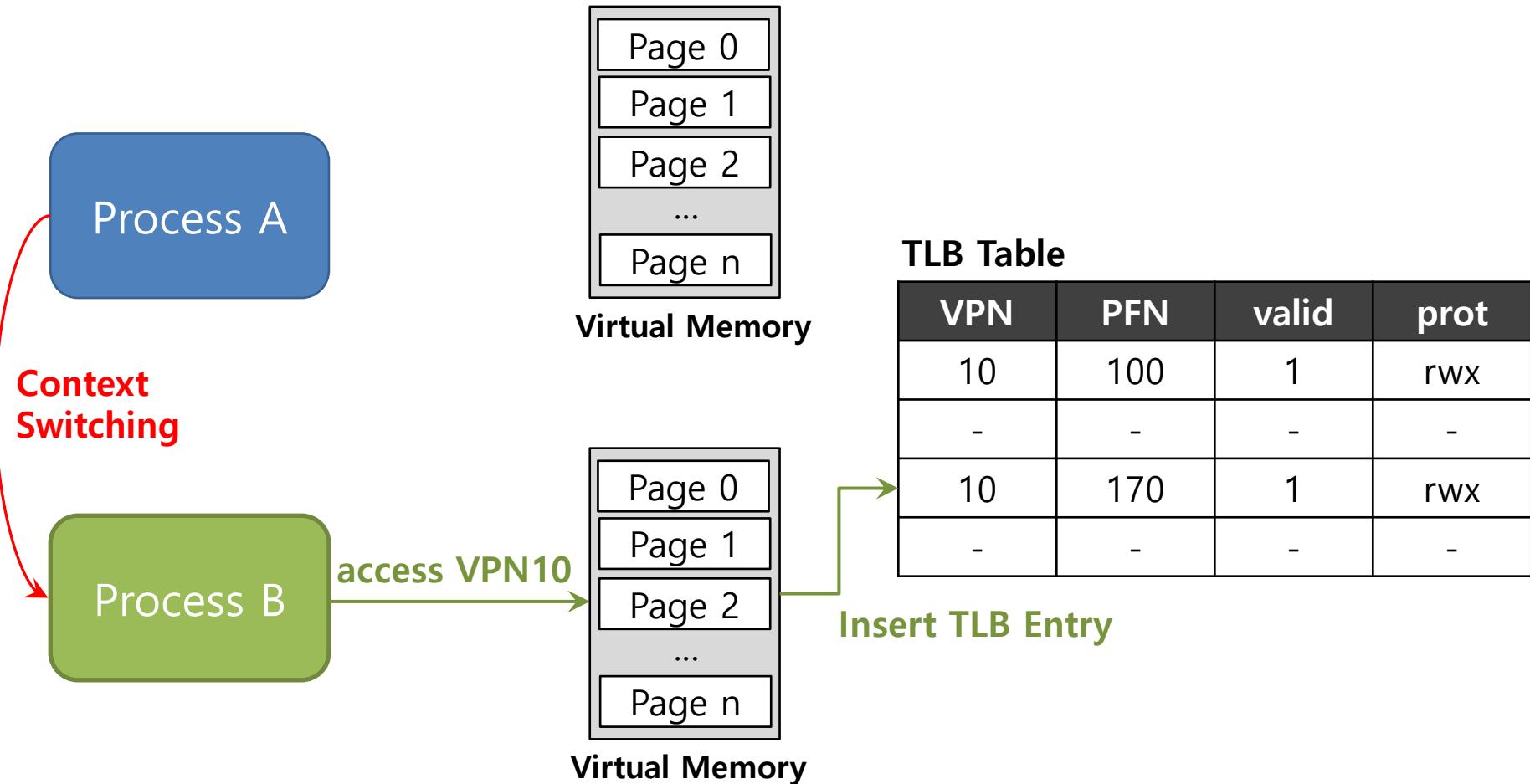


Typical TLB entry look like this

TLB Issue: Context Switching

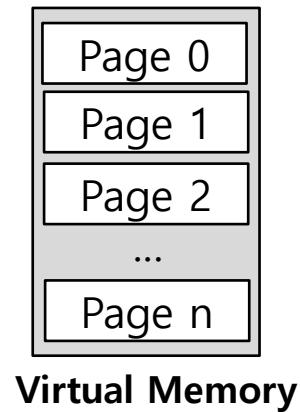


TLB Issue: Context Switching



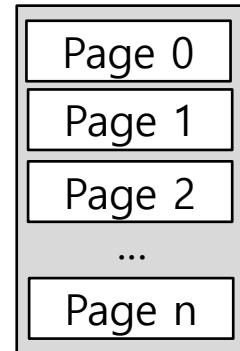
TLB Issue: Context Switching

Process A



Virtual Memory

Process B



Virtual Memory

TLB Table

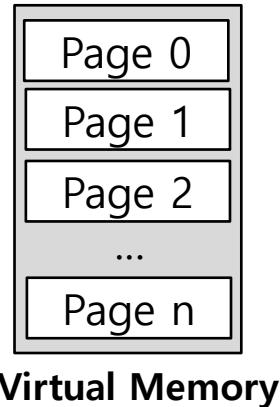
VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

To Solve Problem

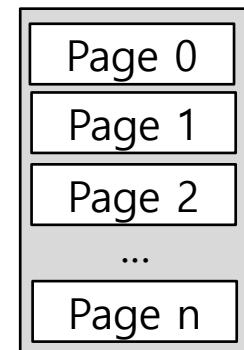
- Provide an address space identifier(ASID) field in the TLB.

Process A



Virtual Memory

Process B



Virtual Memory

TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

Another Case

- Two processes share a page.

- Process 1 is sharing physical page 101 with Process2.
- P1 maps this page into the 10th page of its address space.
- P2 maps this page to the 50th page of its address space.

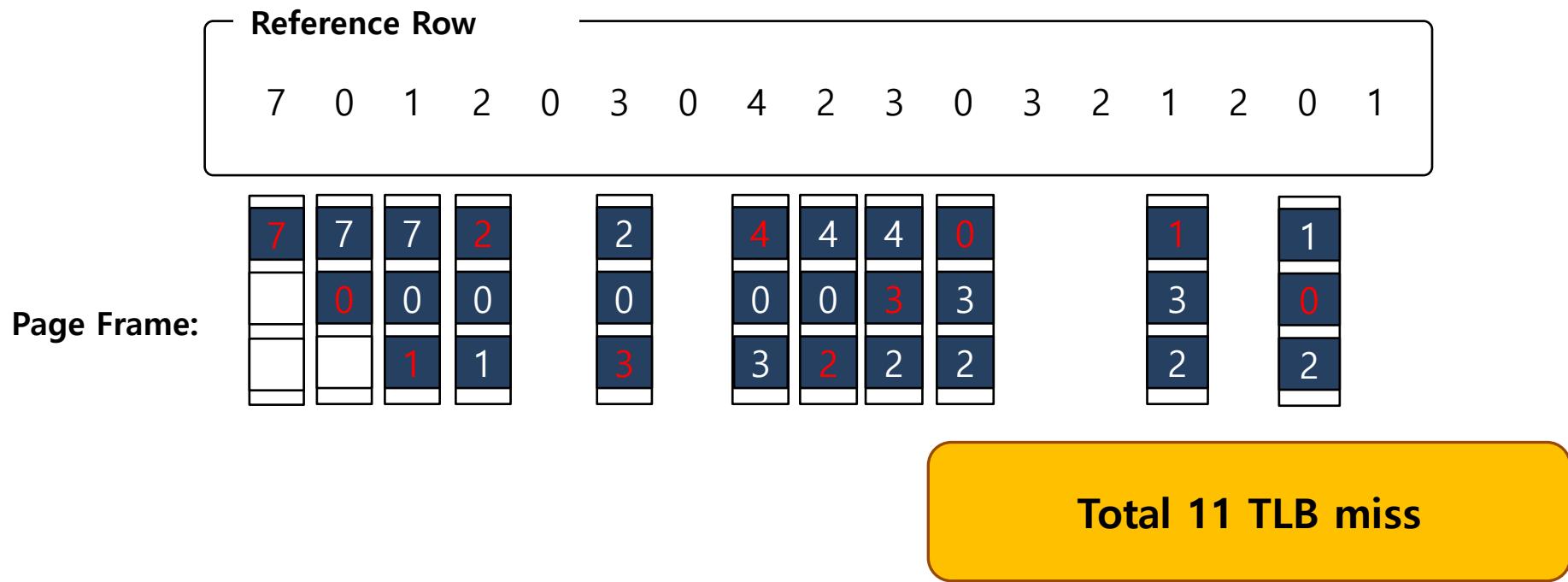
VPN	PFN	valid	prot	ASID
10	101	1	rwx	1
-	-	-	-	-
50	101	1	rwx	2
-	-	-	-	-

Sharing of pages is useful as it reduces the number of physical pages in use.

TLB Replacement Policy

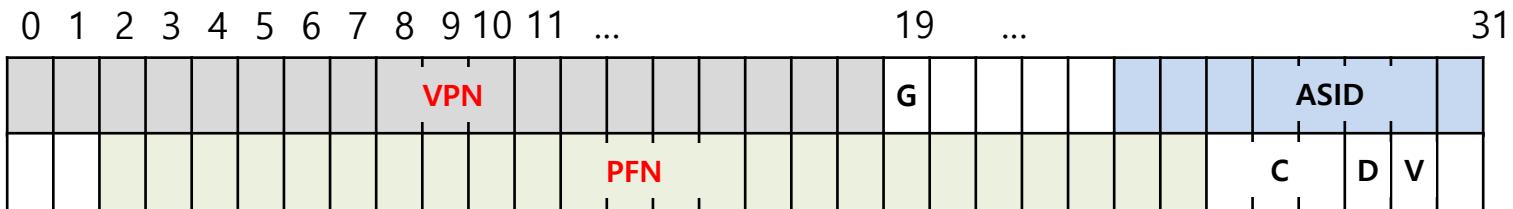
□ LRU(Least Recently Used)

- ◆ Evict an entry that has not recently been used.
- ◆ Take advantage of *locality* in the memory-reference stream.



A Real TLB Entry

All 64 bits of this TLB entry(example of MIPS R4000)



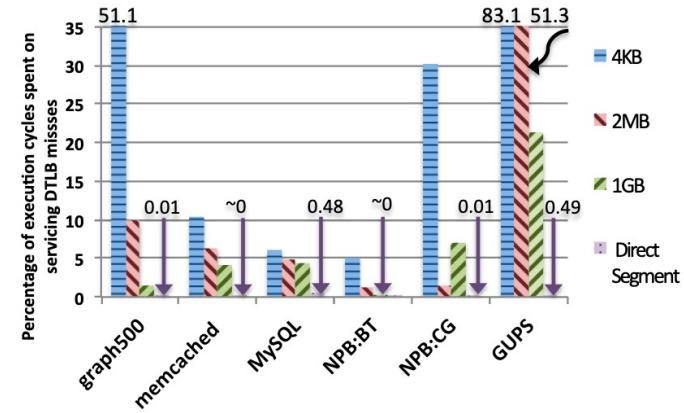
Flag	Content
19-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support up to 64GB of main memory($2^{24} * 4KB$ pages).
Global bit(G)	Used for pages that are globally-shared among processes (ignore ASID).
ASID	OS can use to distinguish between address spaces.
Coherence bit(C)	determine how a page is cached by the hardware.
Dirty bit(D)	marking when the page has been written.
Valid bit(V)	tells the hardware if there is a valid translation present in the entry.

ISA provides the mechanisms to control the TLB content:
TLBP (probe), **TLBR**(read), **TLBWI** (replace index),
TLBWRI (replace random)

TLB today

- Applications with very large memory footprint might be heavily impacted by TLB size (i.e., the locality of the code/data is too large exceeding **TLB-coverage**)

- ◆ Data/Instruction Separate TLBs
- ◆ Multiple level TLB
- ◆ Mega pages
- ◆ No TLB → no pages for a fraction of the address space



- TLB is an itchy issue for virtualization
 - ◆ Complex hardware support required to minimize performance overheads
 - ◆ Nested and deep multilevel page tables makes TLB performance critical

Recommended Reading (beyond class)

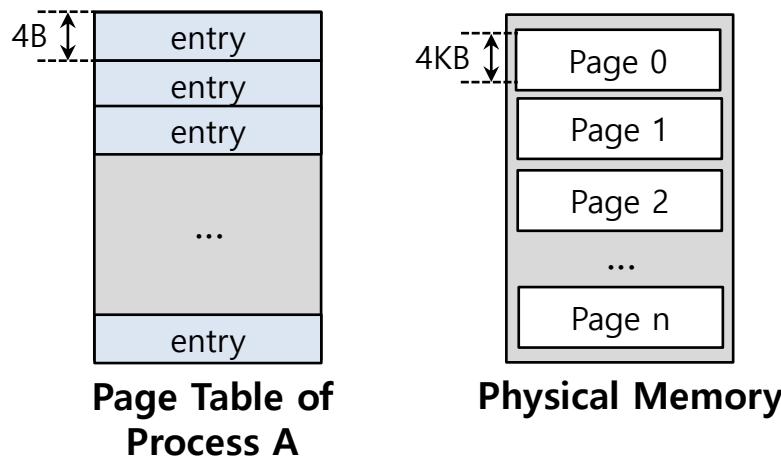
- ▣ Unikernels
 - ◆ <http://unikernel.org>
- ▣ Specialized single-address-space OS
 - ◆ One application
 - ◆ Convert the OS back in a set of library routines
 - ◆ Designed to be run in virtual environments
- ▣ Improved security and performance over conventional OS
 - ◆ Smaller attack surface
 - ◆ No virtual memory overhead
 - ◆ No VMM redirections in syscalls

Paging: Smaller Tables

Operating System: Three Easy Pieces

Paging: Linear Tables

- We usually have one page table for **every process** in the system.
 - ◆ Assume that 32-bit address space with 4KB pages and 4-byte page-table entry.

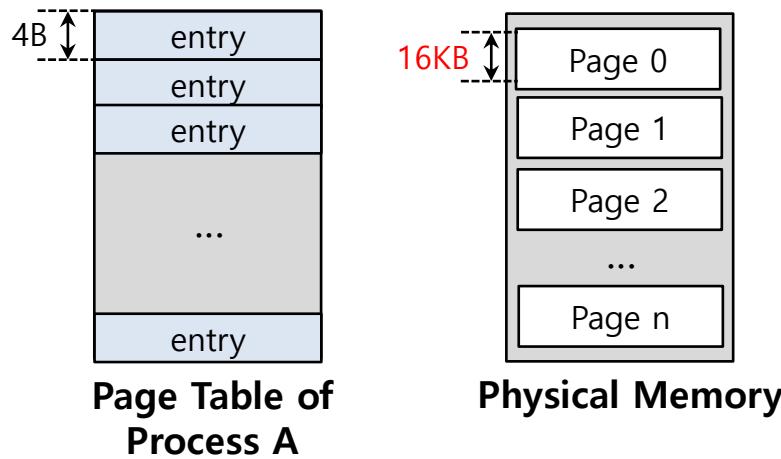


$$\text{Page table size} = \frac{2^{32}}{2^{12}} * 4\text{Byte} = 4\text{MByte}$$

Page tables are **too big** and thus consume too much memory.

Paging: Smaller Tables

- Page tables are too big and thus consume too much memory.
 - Assume that 32-bit address space with **16KB** pages and 4-byte page-table entry.

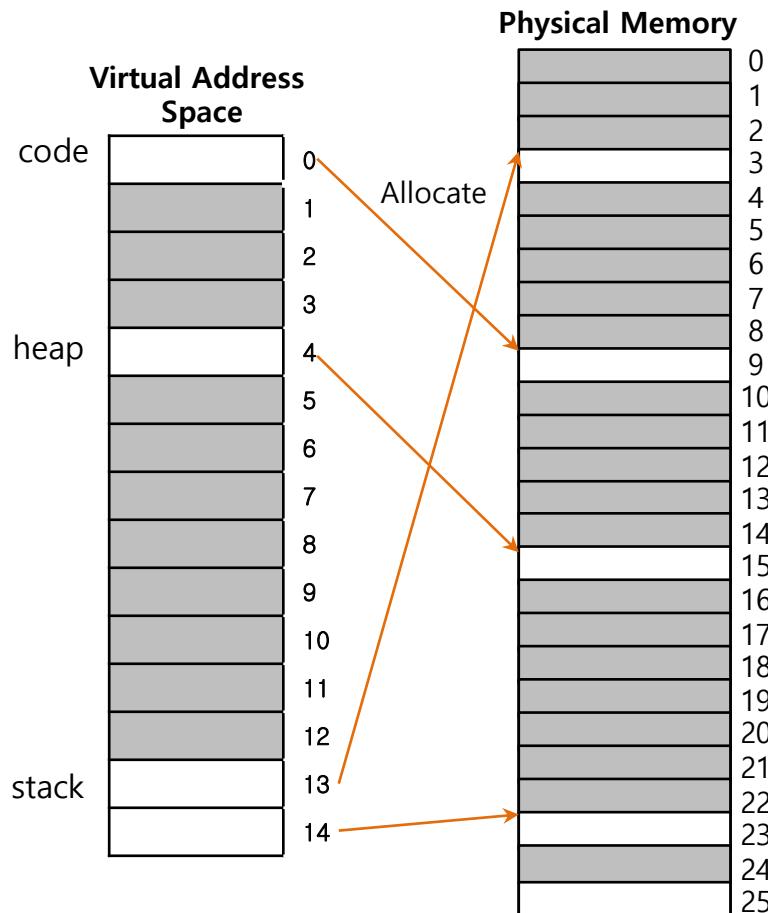


$$\frac{2^{32}}{2^{16}} * 4 = 1MB \text{ per page table}$$

But big pages might lead to **internal fragmentation**.

The Problem

- Single page table for the entries address space of process.



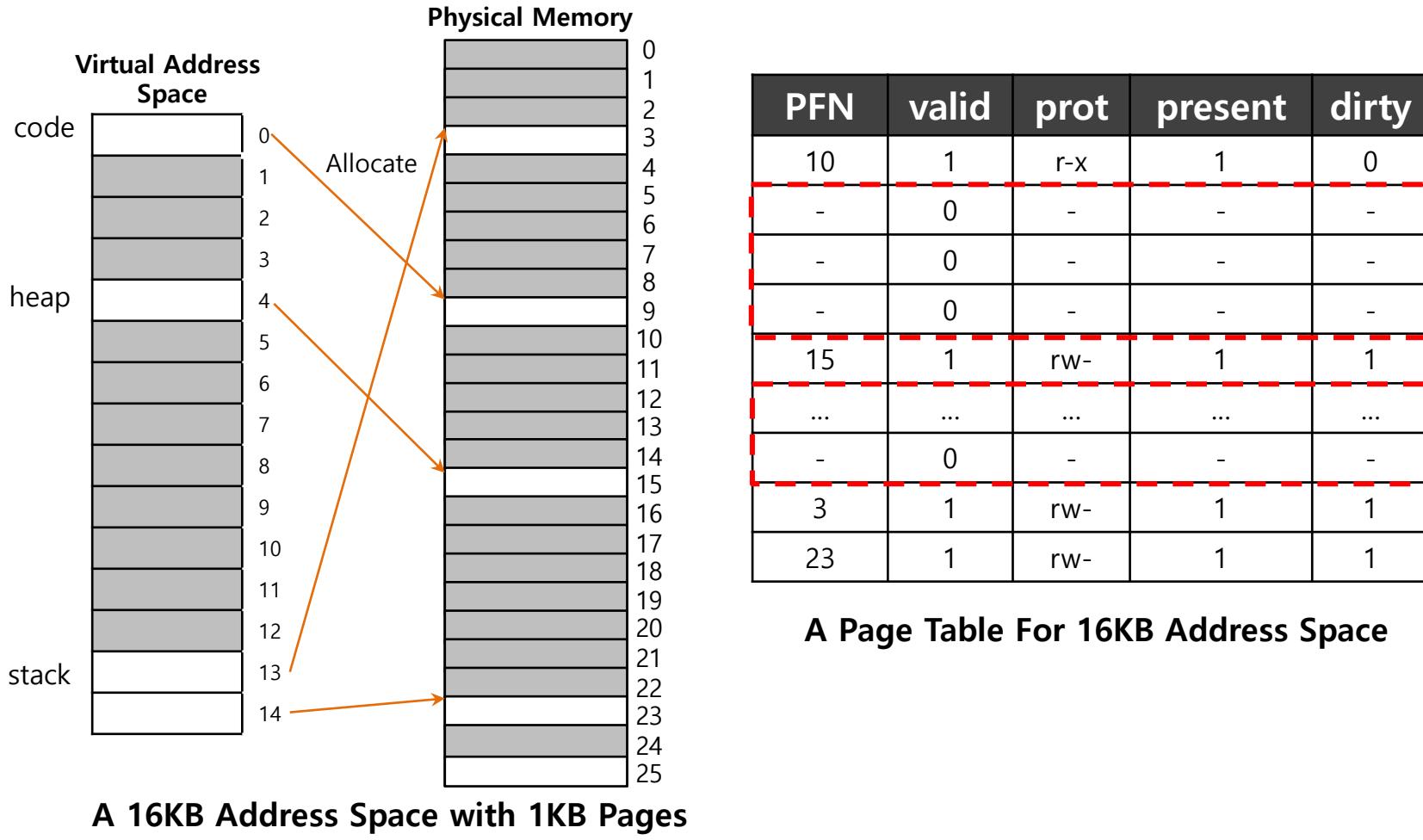
A 16KB Address Space with 1KB Pages

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
...
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

The Problem

- Most of the page table is **unused**, full of invalid entries.

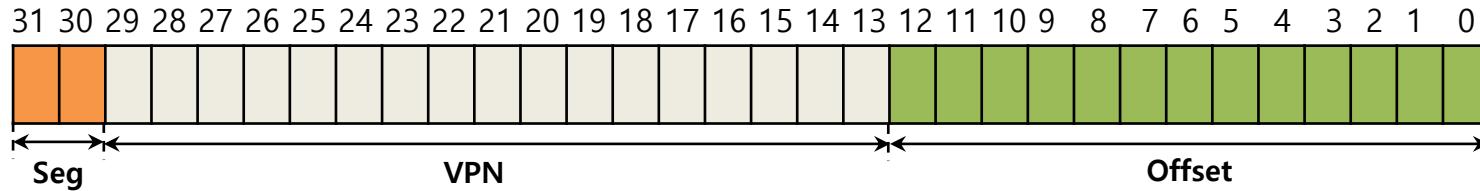


Hybrid Approach to the Problem: Paging and Segments

- ▣ In order to reduce the memory overhead of page tables.
 - ◆ Using base not to point to the segment itself but rather to hold the **physical address of the page table** of that segment.
 - ◆ The bounds register is used to indicate the end of the page table.

Simple Example of Hybrid Approach

- Each process has **three** page tables associated with it.
 - When process is running, the base register for each of these segments contains the physical address of a linear page table for that segment.



32-bit Virtual address space with 4KB pages

Seg value	Content
00	unused segment
01	code
10	heap
11	stack

TLB miss on Hybrid Approach

- ▣ The hardware get to **physical address** from **page table**.
 - ◆ The hardware uses the segment bits(SN) to determine which base and bounds pair to use.
 - ◆ The hardware then takes the **physical address** therein and **combines** it with the VPN as follows to form the address of the page table entry(PTE) .

```
01:      SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT  
02:      VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT  
03:      AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

Problem of Hybrid Approach

- ▣ Hybrid Approach inherits Segmentation issues
 - ◆ If we have a large but sparsely-used heap, we can still end up with a lot of page table waste (most of the free space should be tracked)
 - ◆ Causing **external fragmentation** to arise again
 - ◆ Page Tables have arbitrary size: it's hard to find space for them (or handle it dynamically)

Multi-level Page Tables

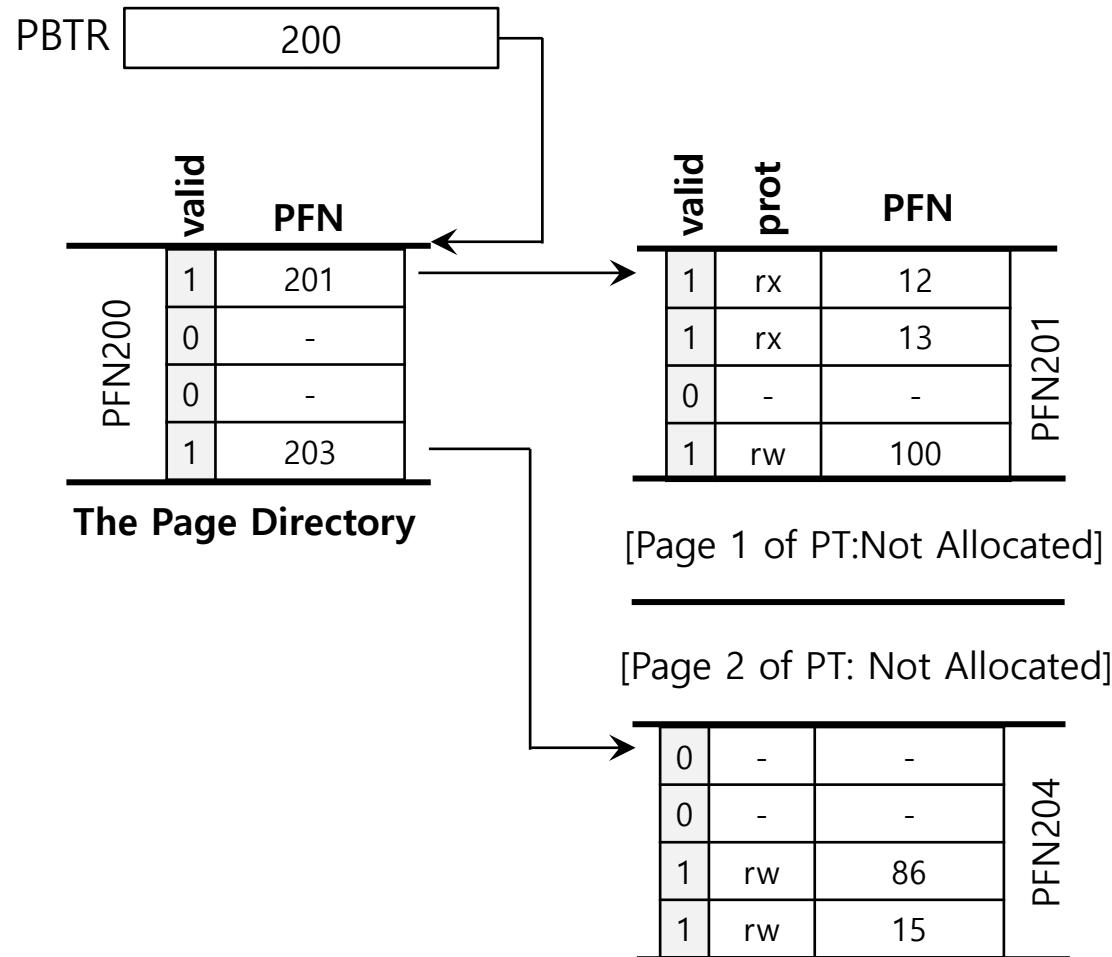
- ▣ Turns the linear page table into something like a tree.
 - ◆ Chop up the page table into page-sized units.
 - ◆ If an entire page of page-table entries is invalid, don't allocate that page of the page table at all.
 - ◆ To track whether a page of the page table is valid, use a new structure, called **page directory**.

Multi-level Page Tables: Page directory

Linear Page Table

PTBR		
valid	prot	PFN
1	rx	12
1	rx	13
0	-	-
1	rw	100
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
1	rw	86
1	rw	15

Multi-level Page Table



Linear (Left) And Multi-Level (Right) Page Tables

Multi-level Page Tables: Page directory entries

- ▣ The page directory contains one entry per page of the page table.
 - ◆ It consists of a number of **page directory entries (PDE)**.
- ▣ PDE (minimally) has a valid bit and page frame number (PFN).

Multi-level Page Tables: Advantage & Disadvantage

❑ Advantage

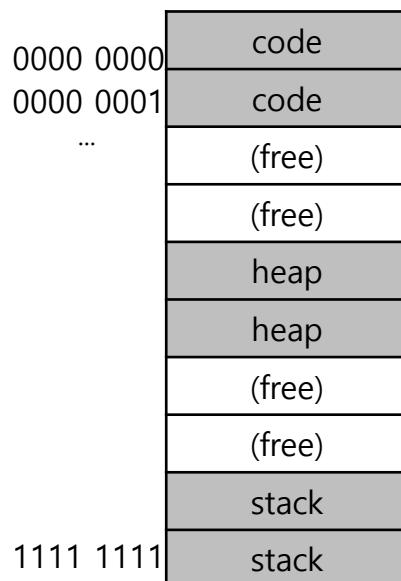
- ◆ Only allocates page-table space in proportion to the amount of address space you are using.
- ◆ The OS can grab the next free page when it needs to allocate or grow a page table.

❑ Disadvantage

- ◆ Multi-level table is a small example of a **time-space trade-off**.
- ◆ **Complexity**.

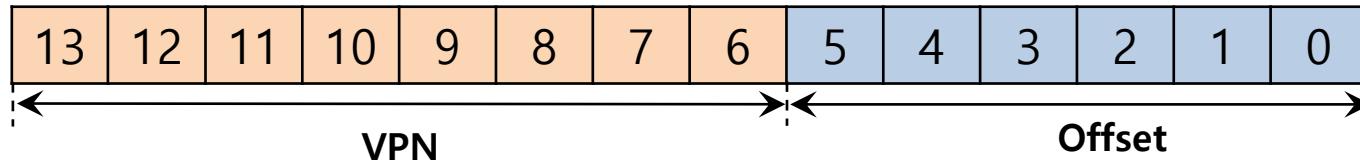
A Detailed Multi-Level Example

- To understand the idea behind multi-level page tables better, let's do an example.



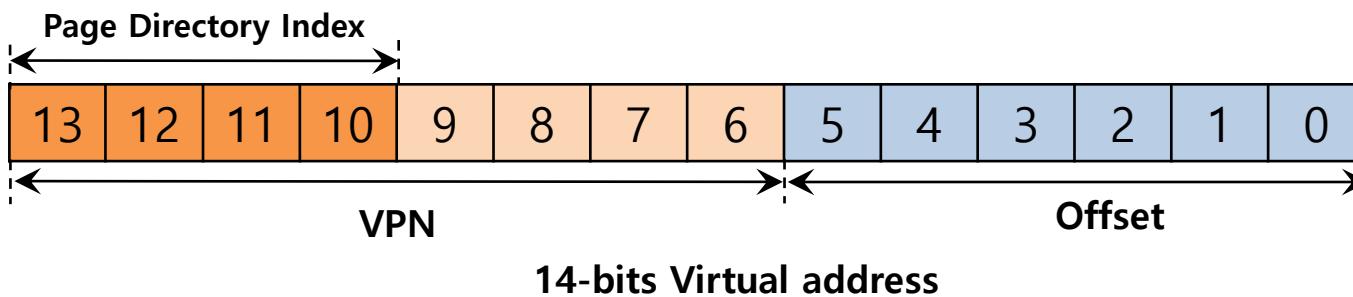
Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
# Page table entry	$2^8(256)$

A 16-KB Address Space With 64-byte Pages



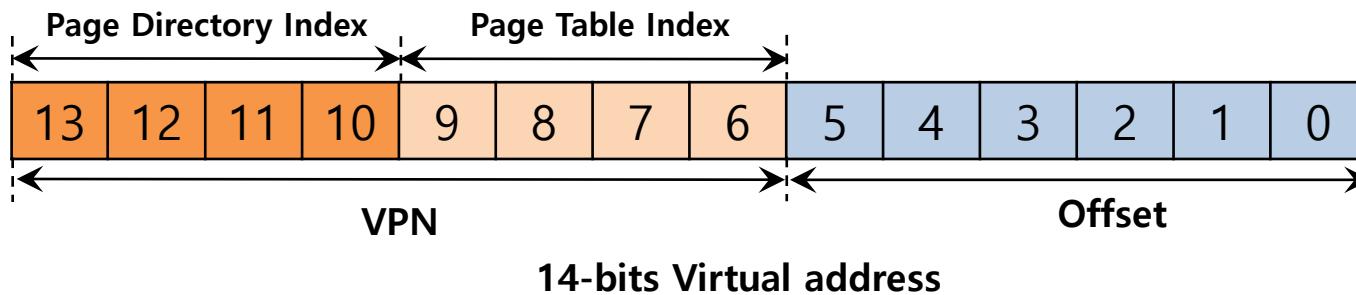
A Detailed Multi-Level Example: Page Directory Index

- The page directory needs one entry per page of the page table
 - ◆ it has 16 entries.
- The PDE is **invalid** → Raise an exception (The access is invalid)



A Detailed Multi-Level Example: Page Table Idx

- ❑ The PDE is valid, we have more work to do.
 - ◆ To fetch the page table entry(PTE) from the page of the page table pointed to by this page-directory entry.
- ❑ This **page-table index** can then be used to index into the page table itself.

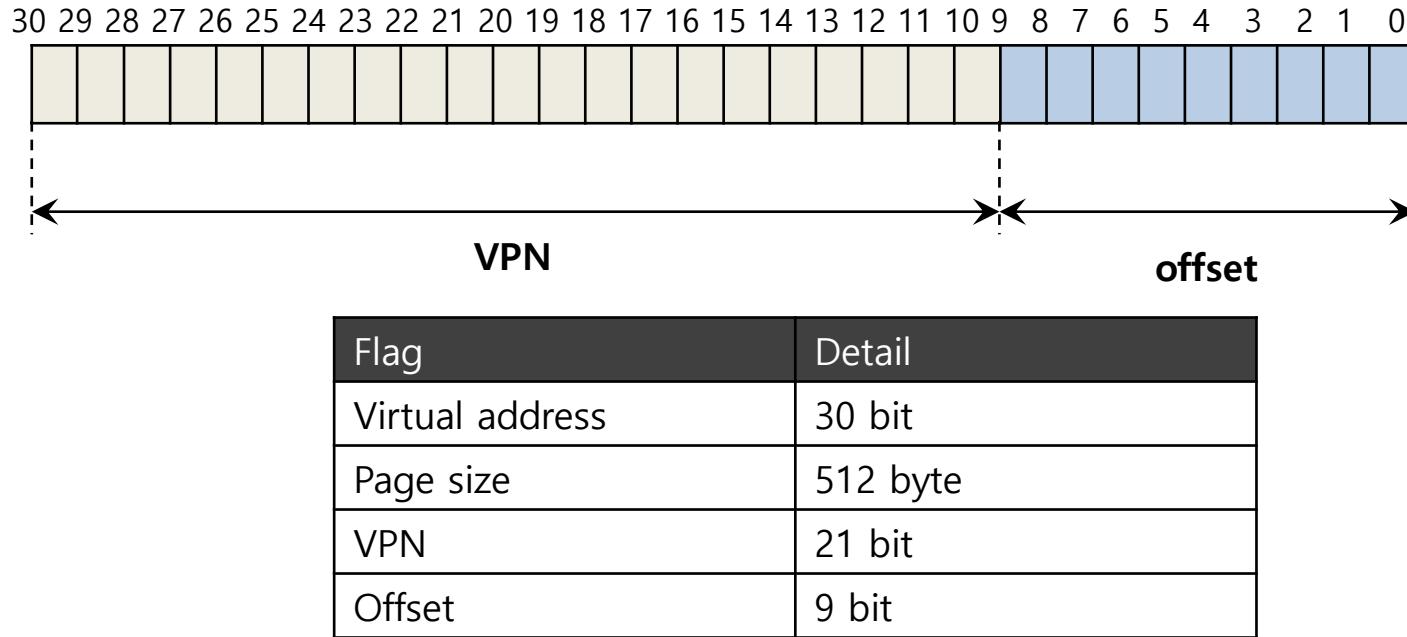


Example

	Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
	PFN	valid?	PFN	valid	prot	PFN	valid	prot
0000 0000	100	1	10	1	r-x	—	0	—
0000 0001	—	0	23	1	r-x	—	0	—
0000 0010	—	0	—	0	—	—	0	—
0000 0011	(free)	0	—	0	—	—	0	—
0000 0100	heap	0	80	1	rw-	—	0	—
0000 0101	heap	0	59	1	rw-	—	0	—
0000 0110	(free)	0	—	0	—	—	0	—
0000 0111	(free)	0	—	0	—	—	0	—
.....	... all free ...	0	—	0	—	—	0	—
1111 1100	(free)	0	—	0	—	—	0	—
1111 1101	(free)	0	—	0	—	—	0	—
1111 1110	stack	0	—	0	—	—	0	—
1111 1111	stack	0	—	0	—	—	0	—
	—	0	—	0	—	—	0	—
	—	0	—	0	—	—	0	—
	—	0	—	0	—	55	1	rw-
	101	1	—	0	—	45	1	rw-

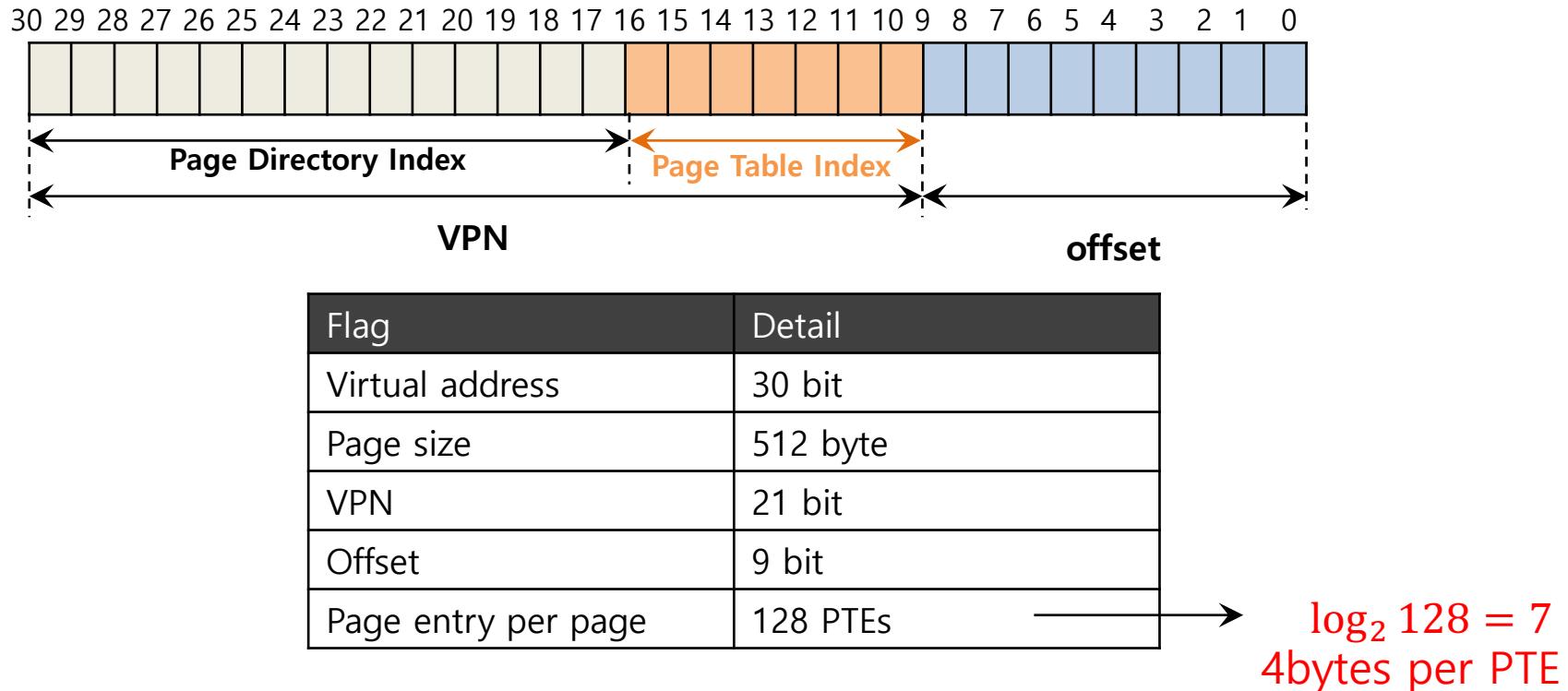
More than Two Levels

- In some cases, a deeper tree is possible (and needed).



More than Two Level : Page Table Index

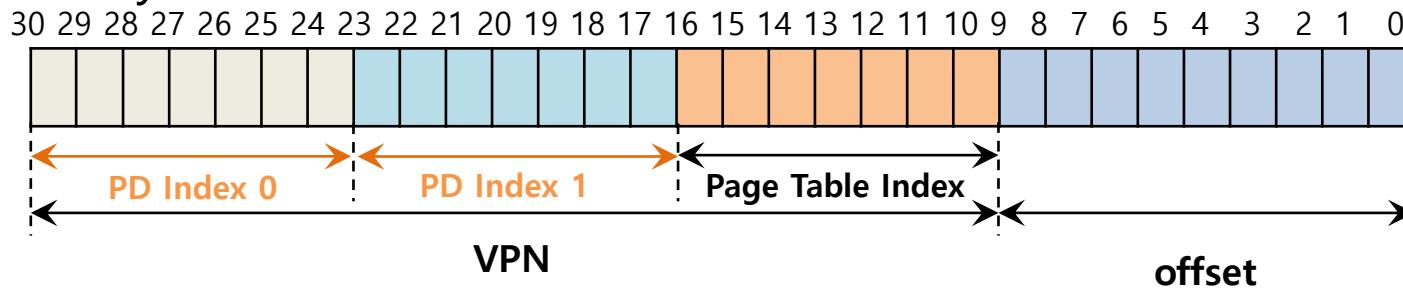
- In some cases, a deeper tree is possible (and needed).



$$PD \text{ has } 2^{14} \text{ entries} = 16K \cdot 4 = 64KB$$

More than Two Level : Page Directory Within a Page

- If our page directory has 2^{14} entries, it spans not one page but 128
assuming size_of(PDE) == size_of(PTE)
- To remedy this problem, we build a **further level** of the tree, by splitting the page directory itself into multiple pages of the page directory.



Multi-level Page Table Control Flow

```
01:     VPN = (VirtualAddress & VPN_MASK) >> SHIFT
02:     (Success,TlbEntry) = TLB_Lookup(VPN)
03:     if(Success == True)           //TLB Hit
04:         if(CanAccess(TlbEntry.ProtectBits) == True)
05:             Offset = VirtualAddress & OFFSET_MASK
06:             PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
07:             Register = AccessMemory(PhysAddr)
08:         else RaiseException(PROTECTION_FAULT);
09:     else // perform the full multi-level lookup
```

- (1 lines) extract the virtual page number(VPN)
- (2 lines) check if the TLB holds the translation for this VPN
- (5-8 lines) extract the page frame number from the relevant TLB entry, and form the desired physical address and access memory

Multi-level Page Table Control Flow

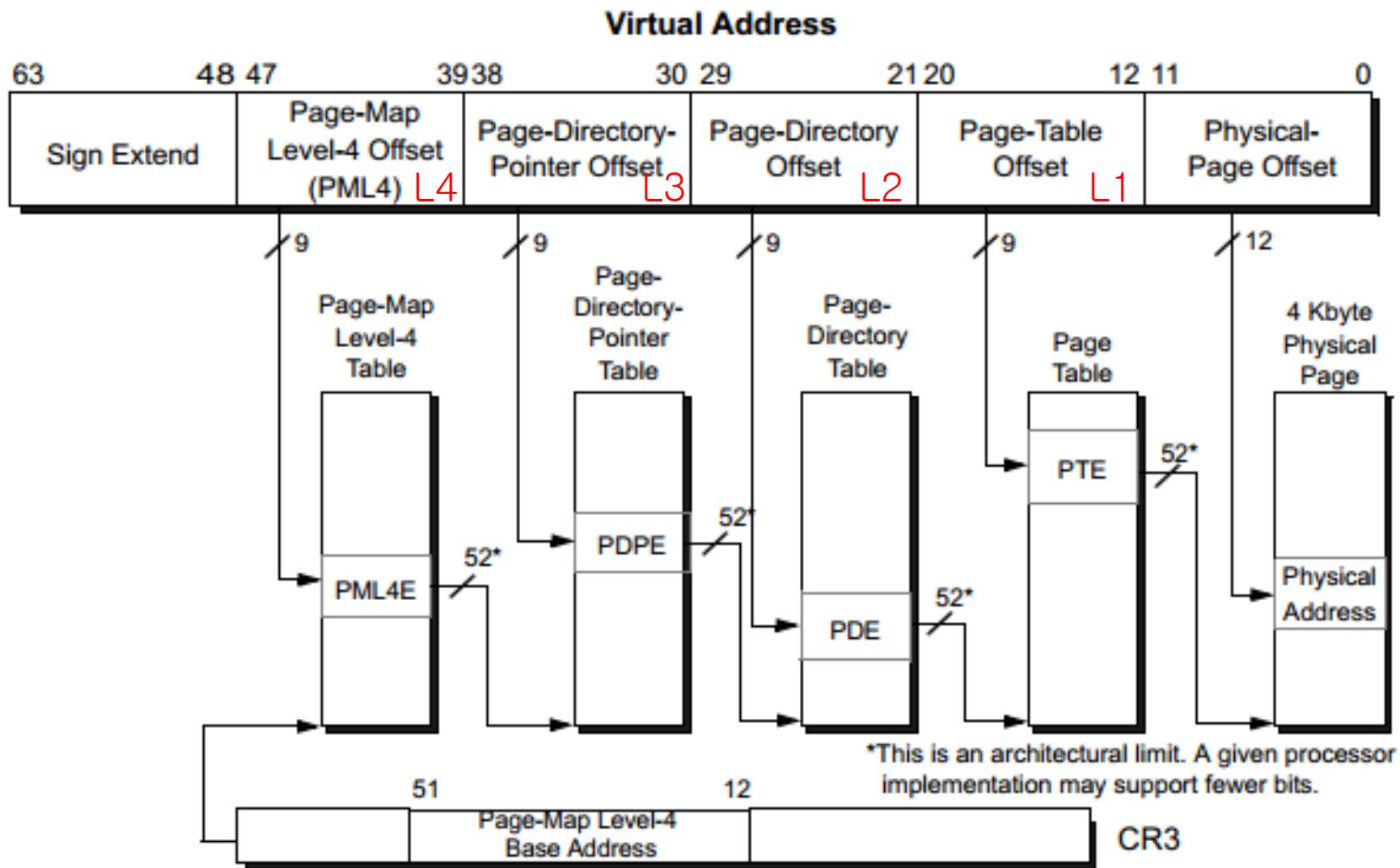
```
11:     else
12:             PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13:             PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14:             PDE = AccessMemory(PDEAddr)
15:             if (PDE.Valid == False)
16:                 RaiseException(SEGMENTATION_FAULT)
17:             else // PDE is Valid: now fetch PTE from PT
```

- ◆ (11 lines) extract the Page Directory Index(PDIndex)
- ◆ (13 lines) get Page Directory Entry(PDE)
- ◆ (15-17 lines) Check PDE valid flag. If valid flag is true, fetch Page Table entry from Page Table

The Translation Process: Remember the TLB

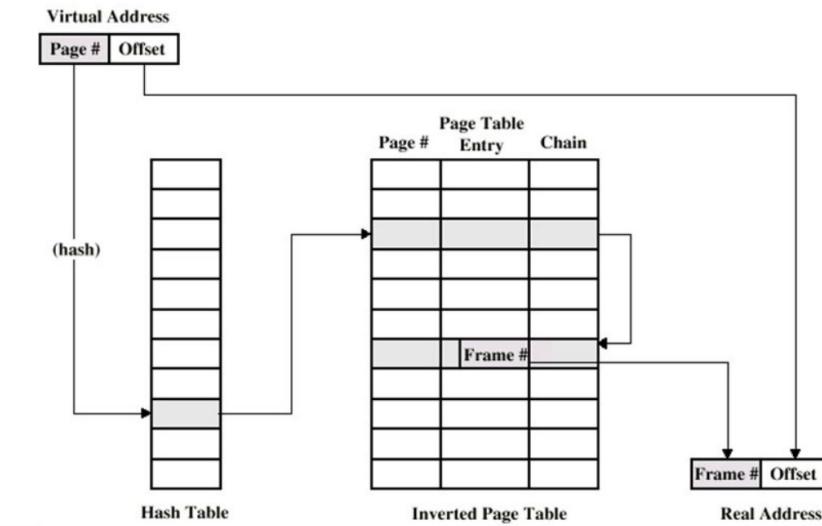
```
18:     PTIndex = (VPN & PT_MASK) >> PT_SHIFT
19:     PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
20:     PTE = AccessMemory(PTEAddr)
21:     if(PTE.Valid == False)
22:         RaiseException(SEGMENTATION_FAULT)
23:     else if(CanAccess(PTE.ProtectBits) == False)
24:         RaiseException(PROTECTION_FAULT);
25:     else
26:         TLB_Insert(VPN, PTE.PFN , PTE.ProtectBits)
27:         RetryInstruction()
```

x86-64: today 4 levels(256TB), planned 5 (128 PB)



Inverted Page Table

- Keeping a single page table that has an entry for each physical page of the system.
- The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.
- Used with a hashing table (hash anchor table) in order to allow practical implementations



Swapping Page Tables to Disk

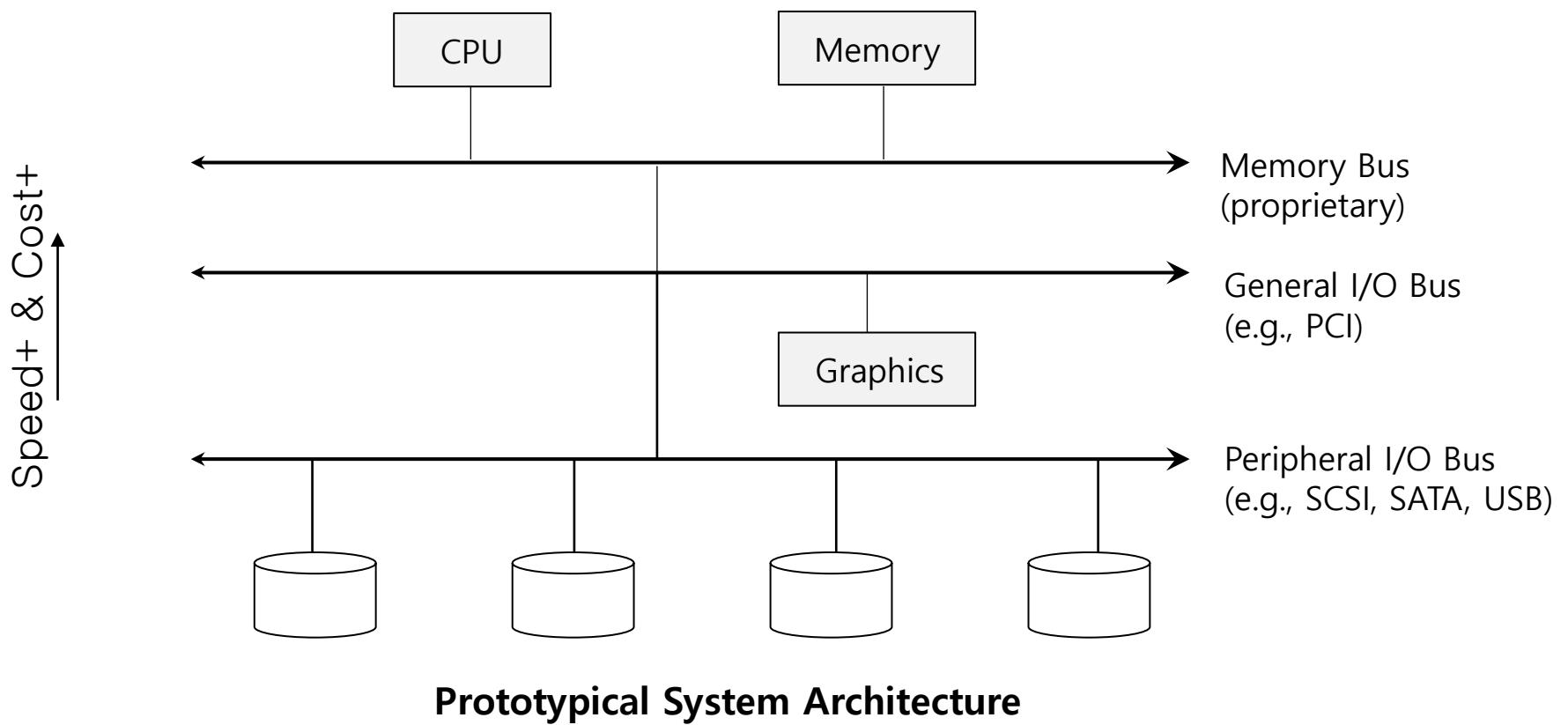
- When memory conditions are harsh, kernel allocated pages can be swapped to disk
- Page tables (a portion of it) might reside only in disk at a given time
- Ignored for sake of simplicity until now (we have no clue about swapping yet)

I/O Devices

Operating System: Three Easy Pieces

- ▣ I/O is **critical** to computer system to **interact with systems**.
- ▣ Issue :
 - ◆ How should I/O be integrated into systems?
 - ◆ What are the general mechanisms?
 - ◆ How can we make the efficiently?

Structure of input/output (I/O) device



CPU is attached to the main memory of the system via some kind of memory bus.
Some devices are connected to the system via a general I/O bus.

Why not a flat design? (like in the early days)

❑ Buses

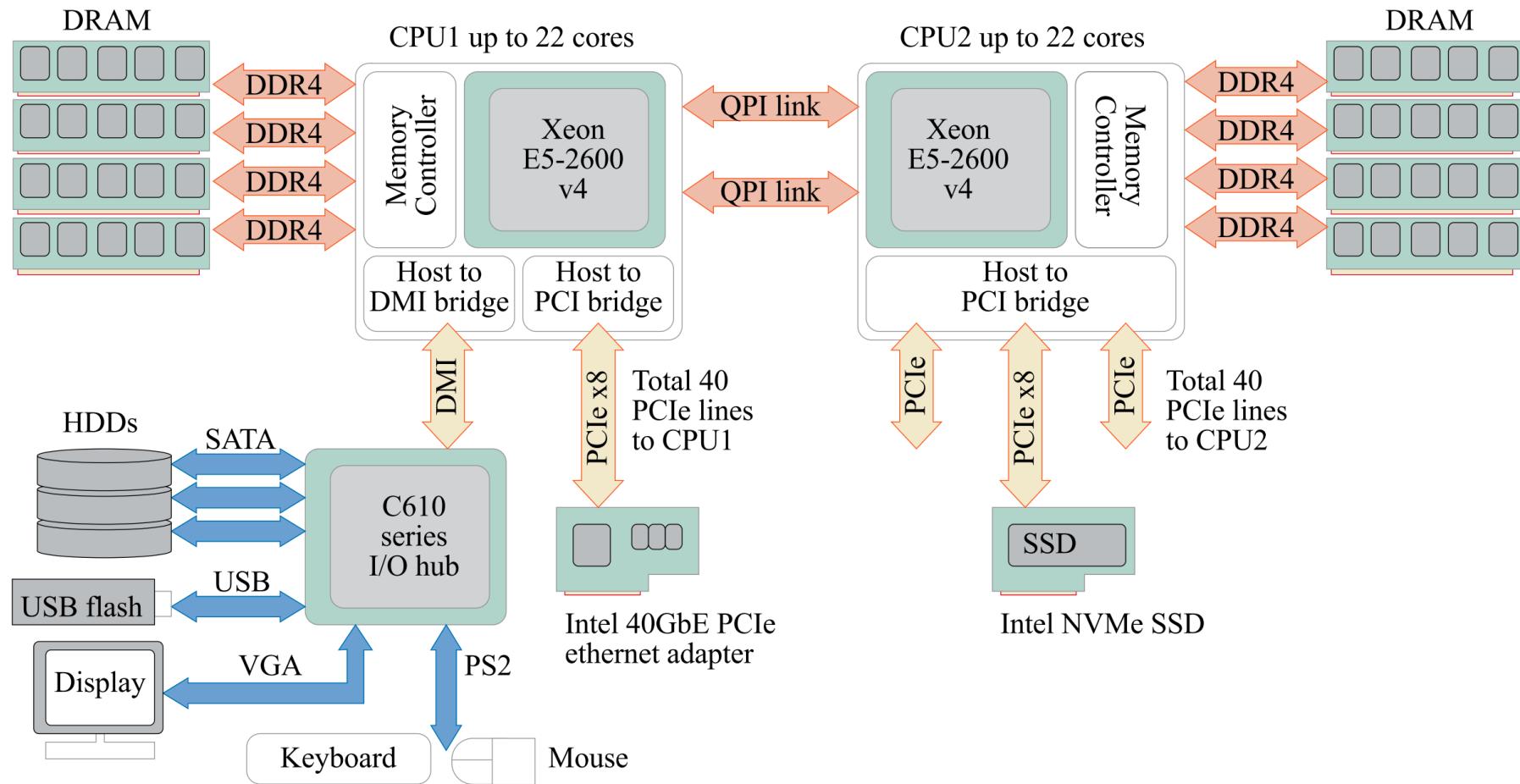
- ◆ Data paths that provided to enable information between CPU(s), RAM, and I/O devices.

❑ I/O bus

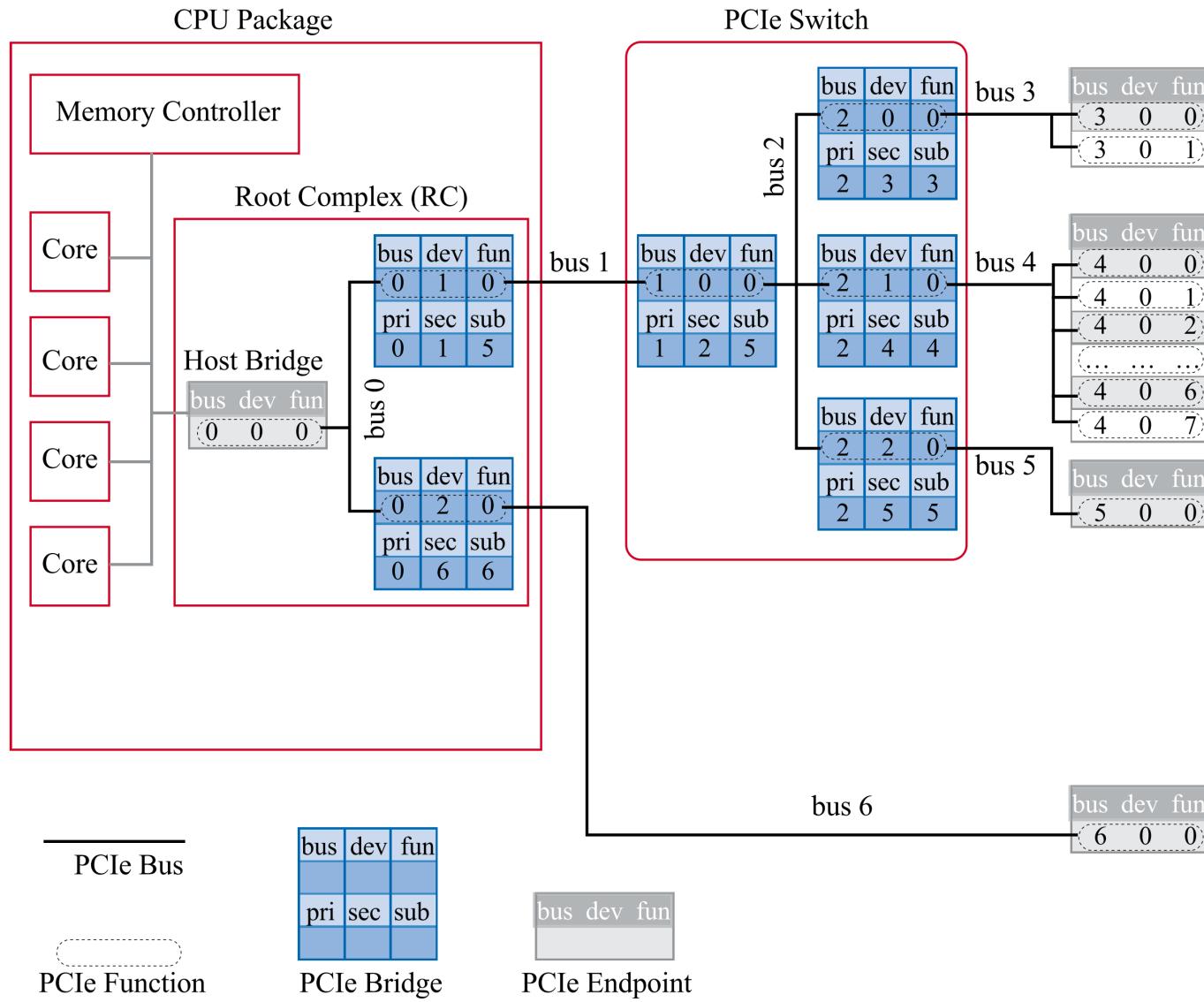
- ◆ Data path that connects a CPU to an I/O device.
 - ◆ I/O bus is connected to I/O device by three hardware components: I/O ports, interfaces and device controllers.
-
- ❑ In current system (due to scalability limitations of the buses) most high-speed buses have migrated to **point-to-point networks**

Todays Systems

- Two-socket server with Xeon E5-2600 v4

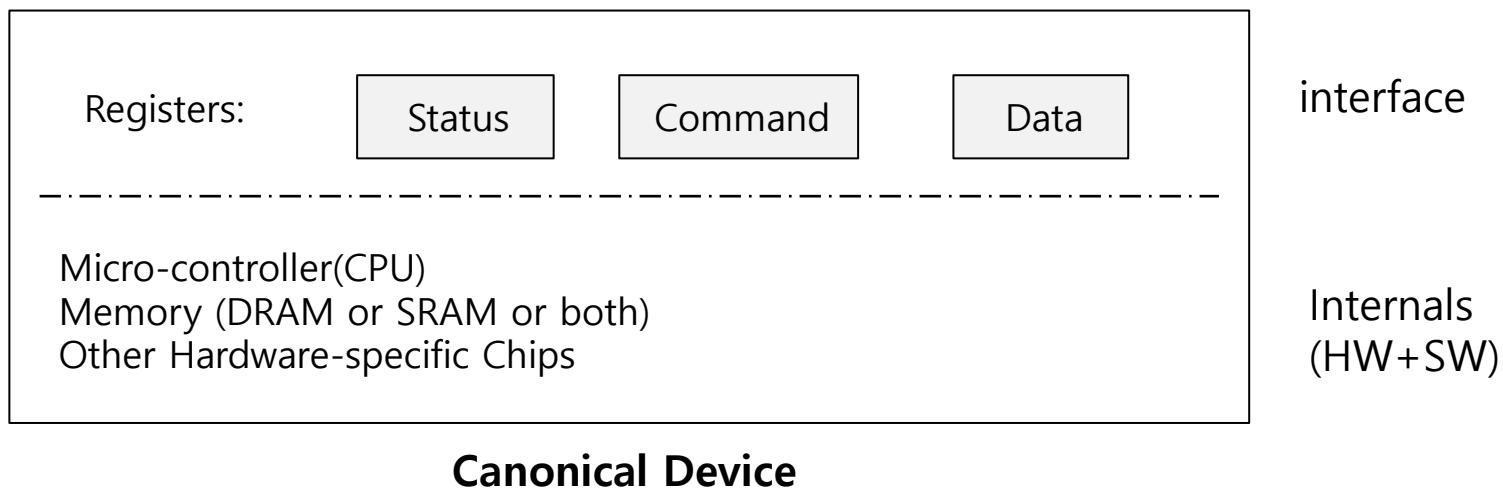


PCIe

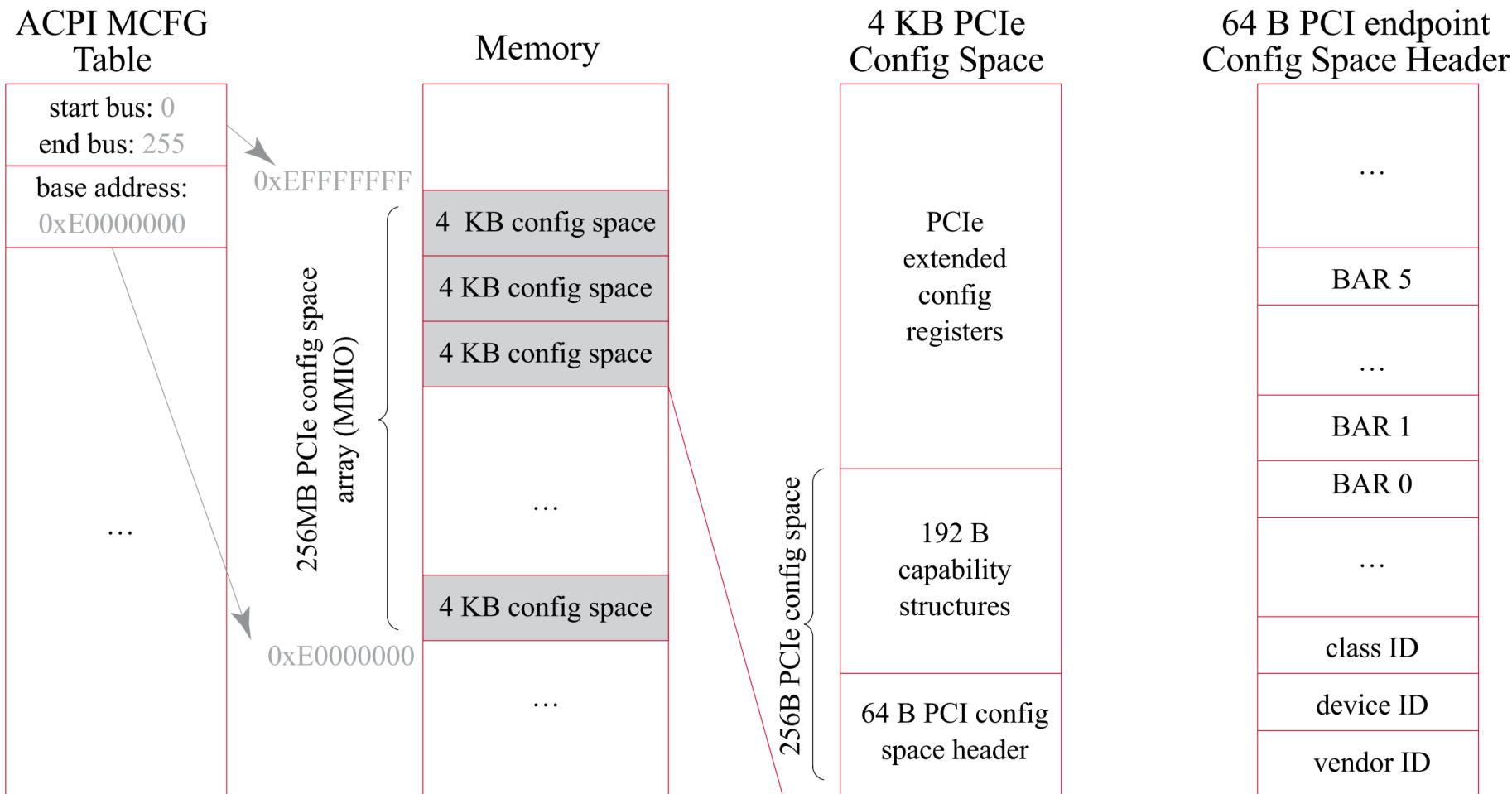


Canonical Device

- Canonical Devices has two important components.
 - Hardware interface allows the system software to control its operation.
 - Internals which is implementation specific.



PCIe Devices (the real thing)



Hardware interface of Canonical Device

- **status register**

- ◆ See the current status of the device

- **command register**

- ◆ Tell the device to perform a certain task

- **data register**

- ◆ Pass data to the device, or get data from the device

By reading and writing above **three registers**,
the operating system can **control device behavior**.

Hardware interface of Canonical Device (Cont.)

▣ Typical interaction example (Programmed I/O or PIO)

```
while ( STATUS == BUSY)
    ; //wait until device is not busy

write data to data register
write command to command register

    Doing so starts the device and executes the command

while ( STATUS == BUSY)
    ; //wait until device is done with your request
```

Polling

- ▣ Operating system waits until the device is ready by **repeatedly** reading the status register.
 - ◆ Positive aspect is simple and working.
 - ◆ **However, it wastes CPU time just waiting for the device.**
 - Switching to another ready process is better utilizing the CPU.

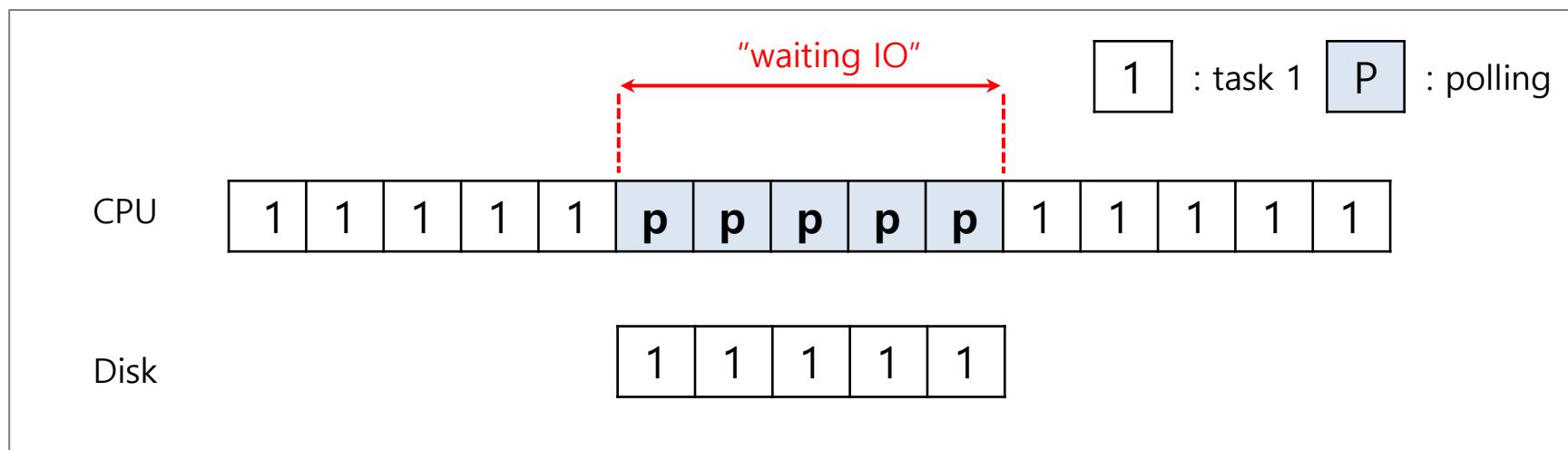


Diagram of CPU utilization by polling

Interrupts

- Put the I/O request process to sleep and context switch to another.
- When the device is finished, wake the process waiting for the I/O by **interrupt** (via interrupt handler or *Interrupt Service Routine ISR*)
 - ◆ Positive aspect is allowed to **CPU and the disk are properly utilized.**

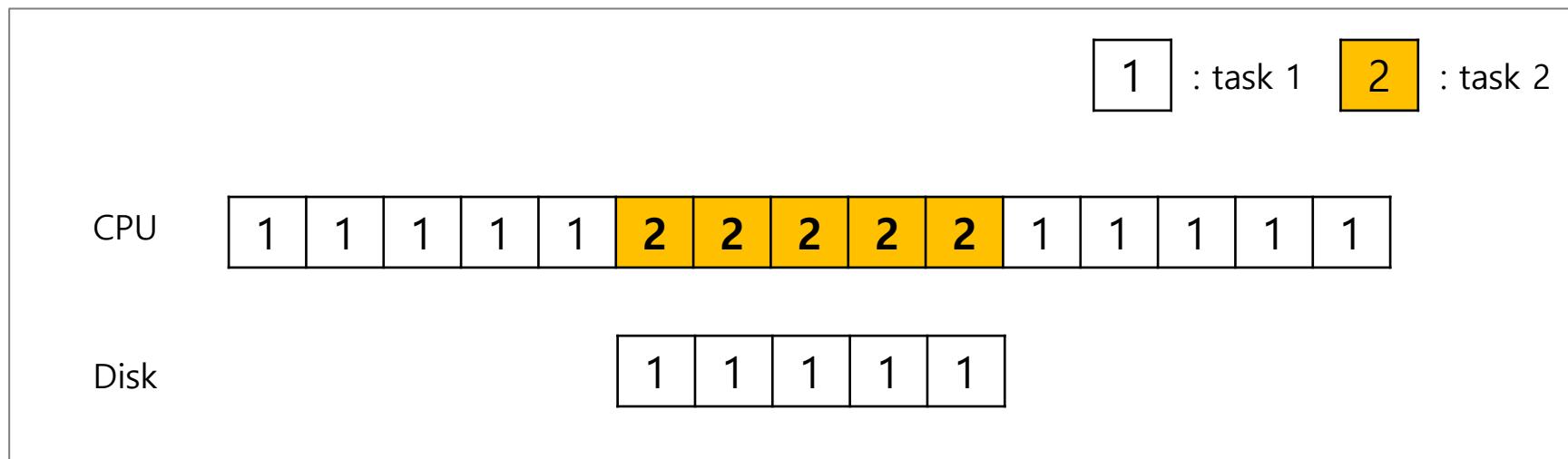


Diagram of CPU utilization by interrupt

Polling vs interrupts

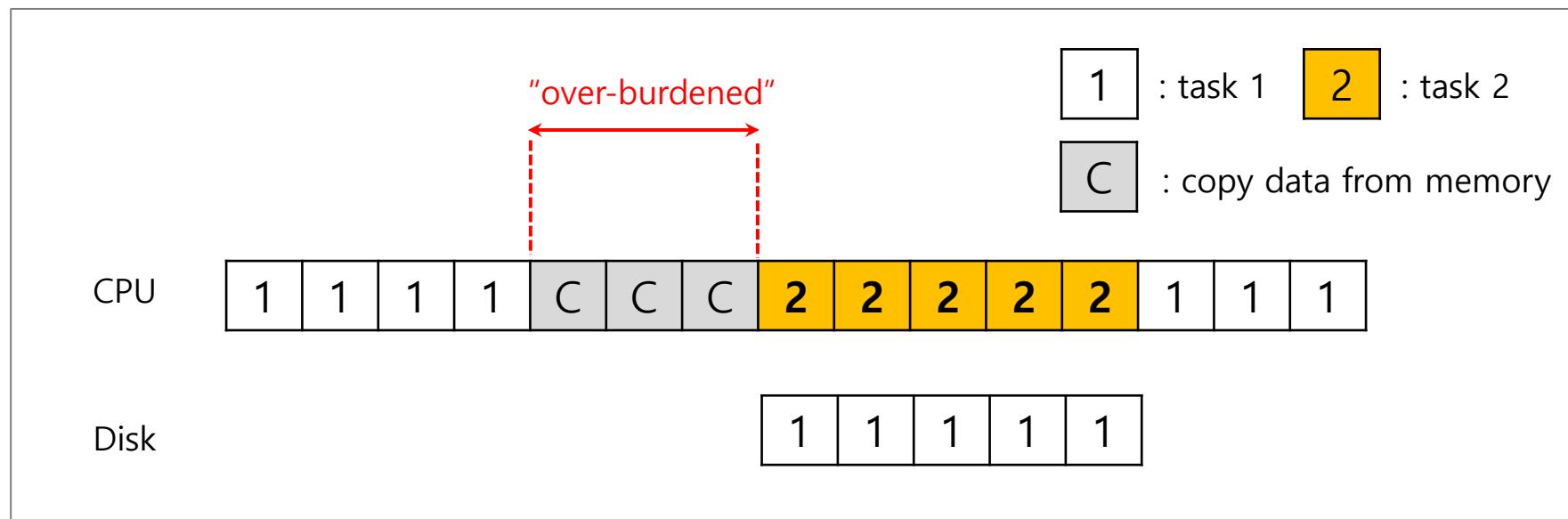
- ▣ *However, “interrupts is not always the best solution”*
 - ◆ If, device performs very quickly (for example, at first poll the operation is done), interrupt will “slow down” the system.
 - ◆ Because **context switch is expensive (switching to another process)**

If a device is fast → **poll** is best.
If it is slow → **interrupts** is better.

- ▣ Hybrid approach
 - ◆ If poll too slow go to interrupts
- ▣ **Coalescing interrupts**
- ▣ Under DDoS attacks go PIO

CPU is once again over-burdened

- CPU wastes a lot of time to copy a *large chunk of data* from memory to the device.



DMA (Direct Memory Access)

- ❑ **Copy data** in memory by knowing “where the data lives in memory, & how much data to copy”
- ❑ Tell the DMA controller to do the “hard-work”
- ❑ When completed, DMA raises an interrupt, I/O begins on Disk.

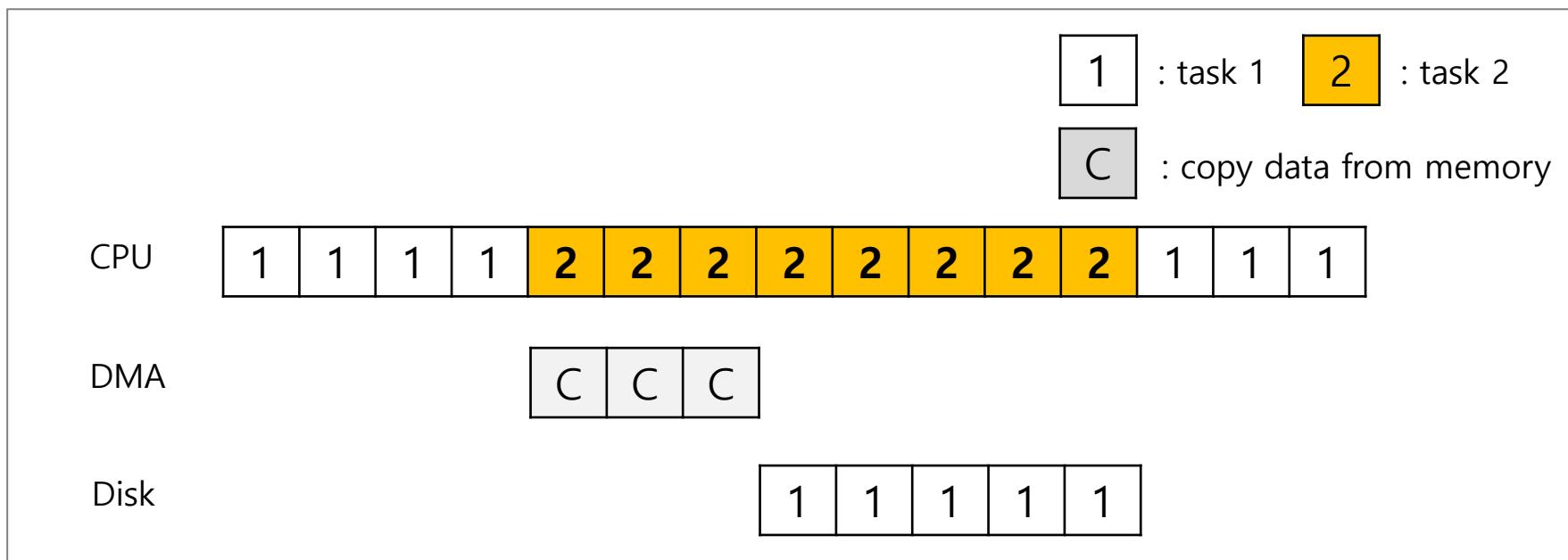


Diagram of CPU utilization by DMA

Device interaction

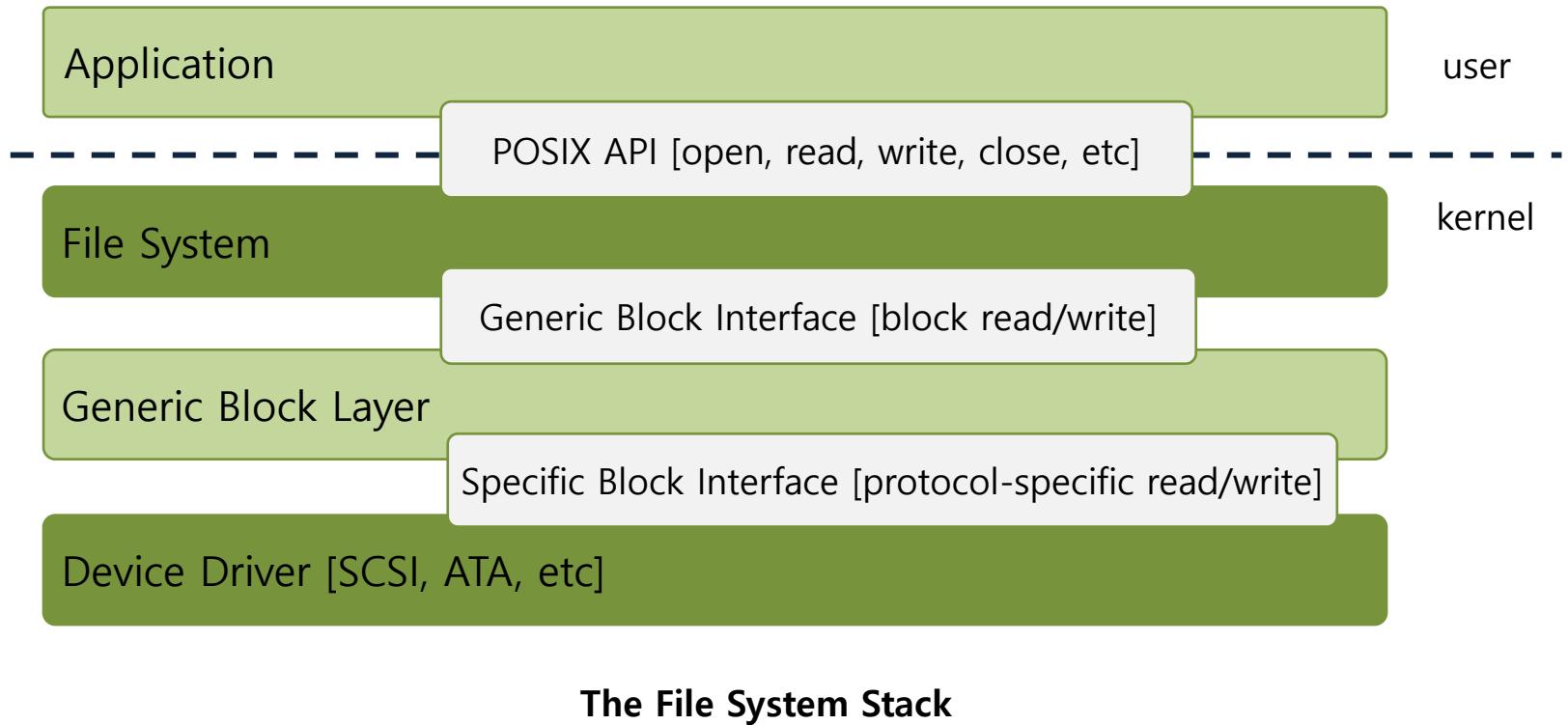
- ▣ How the CPU communicates with the **device**?
- ▣ Approaches
 - ◆ **I/O instructions**: a way for the OS to send data to specific device registers.
 - Ex) `in` and `out` instructions on x86
 - Separate I/O and memory buses in early days
 - ◆ **memory-mapped I/O**
 - Device registers available as if they were memory locations.
 - The OS `load` (to read) or `store` (to write) to the device instead of main memory

Fitting Into The OS: The Device Driver

- ▣ How the OS interact with **different specific interfaces?**
 - ◆ Ex) We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drivers, and so on.
- ▣ **Solution: Abstraction**
 - ◆ Abstraction encapsulate **any specifics of device interaction.**
 - ◆ Only the lowest level should be aware of the specifics: called Device driver

File system Abstraction

- File system **specifics** of which disk class it is using.
 - Ex) It issues **block read** and **write** request to the generic block layer.



Problem of File system Abstraction

- ▣ If there is a device having many special capabilities, these capabilities **will go unused** in the generic interface layer.
 - ◆ Ex) SCSI devices have a **rich error reporting** that are mostly **unused** in Linux because IDE/ATA had very limited capabilities
- ▣ **Over 70% of OS** code is found in device drivers.
 - ◆ Any device drivers are needed because you might plug it to your system.
 - ◆ They are primary contributor to **kernel crashes**, making **more bugs**.
 - ◆ **Driver signing** in current windows system has improved its resiliency greatly

Case Study: A Simple IDE Disk Driver (xv6 uses QEMU IDE)

- ▣ Four types of register
 - ◆ Control, command block, status and error
 - ◆ Mapped to I/O addresses
 - ◆ `in` and `out` I/O instruction
- ▣ Book code doesn't not match with current version
- ▣ *Integrated Drive Electronics* (IDE) was developed in 1987
- ▣ Current xv6-riscv uses virtio disks

- Control Register:

Address 0x3F6 = 0x80 (0000 1RE0): R=reset, E=0 means "enable interrupt"

- Command Block Registers:

Address 0x1F0 = Data Port

Address 0x1F1 = Error

Address 0x1F2 = Sector Count

Address 0x1F3 = LBA low byte (Logical Block Address)

Address 0x1F4 = LBA mid byte

Address 0x1F5 = LBA hi byte

Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive

Address 0x1F7 = Command/status

- Status Register (Address 0x1F7):

7	6	5	4	3	2	1	0
BUSY	READY	FAULT	SEEK	DRQ	CORR	IDDEX	ERROR

- Error Register (Address 0x1F1): (check when Status ERROR==1)

7	6	5	4	3	2	1	0
BBK	UNC	MC	IDNF	MCR	ABRT	T0NF	AMNF

- ◆ BBK = Bad Block
- ◆ UNC = Uncorrectable data error
- ◆ MC = Media Changed
- ◆ IDNF = ID mark Not Found
- ◆ MCR = Media Change Requested
- ◆ ABRT = Command aborted
- ◆ T0NF = Track 0 Not Found
- ◆ AMNF = Address Mark Not Found

- ❑ **Wait for drive to be ready.** Read Status Register (0x1F7) until drive is not busy and READY.
- ❑ **Write parameters to command registers.** Write the sector count, logical block address (LBA) of the sectors to be accessed, and drive number (master=0x00 or slave=0x10, as IDE permits just two drives) to command registers (0x1F2-0x1F6).
- ❑ **Start the I/O.** by issuing read/write to command register. Write READ—WRITE command to command register (0x1F7).
- ❑ **Data transfer (for writes):** Wait until drive status is READY and DRQ (drive request for data); write data to data port.
- ❑ **Handle interrupts.** In the simplest case, handle an interrupt for each sector transferred; more complex approaches allow batching and thus one final interrupt when the entire transfer is complete.
- ❑ **Error handling.** After each operation, read the status register. If the ERROR bit is on, read the error register for details.

xv6: I/O buffer (node struct)

```
struct buf {           //chunk of 512B to read/write
    int flags;
    uint dev;
    uint sector;
    struct buf *prev; // LRU cache list
    struct buf *next;
    struct buf *qnext; // disk queue
    uchar data[512];
};

#define B_BUSY 0x1 // buffer is locked by some process #d
efine B_VALID 0x2 // buffer has been read from disk #defin
e B_DIRTY 0x4 // buffer needs to be written to disk
```

xv6 code: Queues request (if IDE not avail) or issue the req.

```
void ide_rw(struct buf *b) {  
    acquire(&ide_lock);  
  
    for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)  
        ;                                // walk queue (beware 2nd term)  
    *pp = b;                            // add request to end  
    if (ide_queue == b)                // if q was empty (only has b)  
        ide_start_request(b);          // send req to disk  
    while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)  
        sleep(b, &ide_lock);          // wait for completion and rel. lock  
    release(&ide_lock);  
}
```

xv6 code: intercedes with the driver

```
static void ide_start_request(struct buf *b) {  
    ide_wait_ready();  
  
    outb(0x3f6, 0); // generate interrupt  
    outb(0x1f2, 1); // how many sectors to read/write?  
    outb(0x1f3, b->sector & 0xff); // LBA goes here ...  
    outb(0x1f4, (b->sector >> 8) & 0xff); // ... and here  
    outb(0x1f5, (b->sector >> 16) & 0xff); // ... and here!  
    outb(0x1f6, 0xe0 | ((b->dev & 1)<<4) | ((b->sector>>24) & 0x0f)); //M or S?  
    if(b->flags & B_DIRTY){  
        outb(0x1f7, IDE_CMD_WRITE); // this is a WRITE 0x20 (ide.c)  
        outsl(0x1f0, b->data, 512/4); // transfer data too!  
    } else {  
        outb(0x1f7, IDE_CMD_READ); // this is a READ (no data) 0x30(ide.c)  
    }  
}
```

Ports and Outs/ins

```
static inline void
outb(ushort port, uchar data)
{
    asm volatile("out %0,%1" : : "a" (data), "d" (port));
}
```

```
static inline uchar
inb(ushort port)
{
    uchar data;

    asm volatile("in %1,%0" : "=a" (data) : "d" (port));
    return data;
}
```

```
static inline void
outsl(int port, const void *addr, int cnt)
{
    asm volatile("cld; rep outsl" :
                "=S" (addr), "=c" (cnt) :
                "d" (port), "0" (addr), "1" (cnt) :
                "cc");
}
```

xv6 code: just check the device is ready and not busy

```
static int ide_wait_ready() {  
    while (((int r = inb(0x1f7)) & IDE_BSY) ||  
           !(r & IDE_DRDY))  
    ; // loop until drive isn't busy  
}
```

Device should be initialized somewhere else (at boot)

xv6 code: interrupt handler

```
void ide_intr() { //called from traps()

    struct buf *b;

    acquire(&ide_lock);

    //take b as the first element in the ide_queue (not shown)

    if (!(b->flags & B_DIRTY) && ide_wait_ready(1) >= 0)

        insl(0x1f0, b->data, 512/4); // if READ: get data

    b->flags |= B_VALID;

    b->flags &= ~B_DIRTY;

    wakeup(b); // wakeI/O Devices waiting process (equivalent to signal)

    if ((ide_queue = b->qnext) != 0) // start next request

        ide_start_request(ide_queue); // (if one exists)

    release(&ide_lock);

}
```

Sleep & wakeup

```
// Atomically release lock and sleep on chan.  
// Reacquires lock when awakened.  
void  
sleep(void *chan, struct spinlock *lk)  
{  
    if(proc == 0)  
        panic("sleep");  
  
    if(lk == 0)  
        panic("sleep without lk");  
  
    // Must acquire ptable.lock in order to  
    // change p->state and then call sched.  
    // Once we hold ptable.lock, we can be  
    // guaranteed that we won't miss any wakeup  
    // (wakeup runs with ptable.lock locked),  
    // so it's okay to release lk.  
    if(lk != &ptable.lock){ //DOC: sleeplock0  
        acquire(&ptable.lock); //DOC: sleeplock1  
        release(lk);  
    }  
  
    // Go to sleep.  
    proc->chan = chan;  
    proc->state = SLEEPING;  
    sched();  
  
    // Tidy up.  
    proc->chan = 0;  
  
    // Reacquire original lock.  
    if(lk != &ptable.lock){ //DOC: sleeplock2  
        release(&ptable.lock);  
        acquire(lk);  
    }  
}
```

```
// Wake up all processes sleeping on chan.  
// The ptable lock must be held.  
static void  
wakeup1(void *chan)  
{  
    struct proc *p;  
  
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)  
        if(p->state == SLEEPING && p->chan == chan)  
            p->state = RUNNABLE;  
}  
  
// Wake up all processes sleeping on chan.  
void  
wakeup(void *chan)  
{  
    acquire(&ptable.lock);  
    wakeup1(chan);  
    release(&ptable.lock);  
}
```

Current xv6 code (kernel/ide.c)

```
// Sync buf with disk.  
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.  
// Else if B_VALID is not set, read buf from disk, set B_VALID.  
void  
idewr(struct buf *b)  
{  
    struct buf **pp;  
  
    if(!(b->flags & B_BUSY))  
        panic("idewr: buf not busy");  
    if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)  
        panic("idewr: nothing to do");  
    if(b->dev != 0 && !havedisk1)  
        panic("idewr: ide disk 1 not present");  
  
    acquire(&idelock);  
  
    // Append b to idequeue.  
    b->qnext = 0;  
    for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  
        ;  
    *pp = b;  
  
    // Start disk if necessary.  
    if(idequeue == b)  
        idestart(b);  
  
    // Wait for request to finish.  
    // Assuming will not sleep too long: ignore proc->killed.  
    while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){  
        sleep(b, &idelock);  
    }  
  
    release(&idelock);  
}
```

```
// Start the request for b. Caller must hold idelock.  
static void  
idestart(struct buf *b)  
{  
    if(b == 0)  
        panic("idestart");  
  
    idewait(0);  
    outb(0x3f6, 0); // generate interrupt  
    outb(0x1f2, 1); // number of sectors  
    outb(0x1f3, b->sector & 0xff);  
    outb(0x1f4, (b->sector >> 8) & 0xff);  
    outb(0x1f5, (b->sector >> 16) & 0xff);  
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));  
    if(b->flags & B_DIRTY){  
        outb(0x1f7, IDE_CMD_WRITE);  
        outsl(0x1f0, b->data, 512/4);  
    } else {  
        outb(0x1f7, IDE_CMD_READ);  
    }  
}
```

Current xv6 code (kernel/ide.c)

```
// Wait for IDE disk to become ready.
static int
idewait(int checkerr)
{
    int r;

    while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
    ;
    if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
        return -1;
    return 0;
}

void
ideinit(void)
{
    int i;

    initlock(&idelock, "ide");
    picenable(IRQ_IDE);
    ioapicenable(IRQ_IDE, ncpu - 1);
    idewait(0);

    // Check if disk 1 is present
    outb(0x1f6, 0xe0 | (1<<4));
    for(i=0; i<1000; i++){
        if(inb(0x1f7) != 0){
            havedisk1 = 1;
            break;
        }
    }

    // Switch back to disk 0.
    outb(0x1f6, 0xe0 | (0<<4));
}
```

```
case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;

// Interrupt handler.
void
ideintr(void)
{
    struct buf *b;

    // Take first buffer off queue.
    acquire(&idelock);
    if((b = idequeue) == 0){
        release(&idelock);
        // cprintf("spurious IDE interrupt\n");
        return;
    }
    idequeue = b->gnext;

    // Read data if needed.
    if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
        insl(0x1f0, b->data, 512/4);

    // Wake process waiting for this buf.
    b->flags |= B_VALID;
    b->flags &= ~B_DIRTY;
    wakeup(b);

    // Start disk on next buf in queue.
    if(idequeue != 0)
        idestart(idequeue);

    release(&idelock);
}
```