

2. Introduction to 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

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 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);
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.

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.

The problem of Concurrency

- The OS is juggling **many things at once**, first running one process, then another, and so forth.
- Modern **multi-threaded programs** also exhibit the concurrency problem.

Concurrency Example

- ❑ A Multi-threaded Program (thread.c)

```
1      #include <stdio.h>
2      #include <stdlib.h>
3      #include "common.h"
4
5      volatile int counter = 0;
6      int loops;
7
8      void *worker(void *arg) {
9          int i;
10         for (i = 0; i < loops; i++) {
11             counter++;
12         }
13         return NULL;
14     }
15
16     int
17     main(int argc, char *argv[])
18     {
19         if (argc != 2) {
20             fprintf(stderr, "usage: threads <value>\n");
21             exit(1);
22     }
```

Concurrency Example (Cont.)

```
23         loops = atoi(argv[1]);
24         pthread_t p1, p2;
25         printf("Initial value : %d\n", counter);
26
27         Pthread_create(&p1, NULL, worker, NULL);
28         Pthread_create(&p2, NULL, worker, NULL);
29         Pthread_join(p1, NULL);
30         Pthread_join(p2, NULL);
31         printf("Final value : %d\n", counter);
32         return 0;
33     }
```

- ◆ The main program creates **two threads**.
 - Thread: a function running within the same memory space. Each thread starts running in a routine called `worker()`.
 - `worker()`: increments a counter

Concurrency Example (Cont.)

- loops determines how many times each of the two workers will **increment the shared counter** in a loop.
 - ◆ loops: 1000.

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000
```

- ◆ loops: 100000.

```
prompt> ./thread 100000
Initial value : 0
Final value : 143012 // huh??
prompt> ./thread 100000
Initial value : 0
Final value : 137298 // what the??
```

Why is this happening?

- ▣ Increment a shared counter → take three instructions.
 1. Load the value of the counter from memory into register.
 2. Increment it
 3. Store it back into memory
- ▣ These three instructions do not execute **atomically**. → Problem of **concurrency** happen.

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);
13          int rc = write(fd, "hello world\n", 13);
14          assert(rc == 13);
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

4. The Abstraction: The Process

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
 - 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

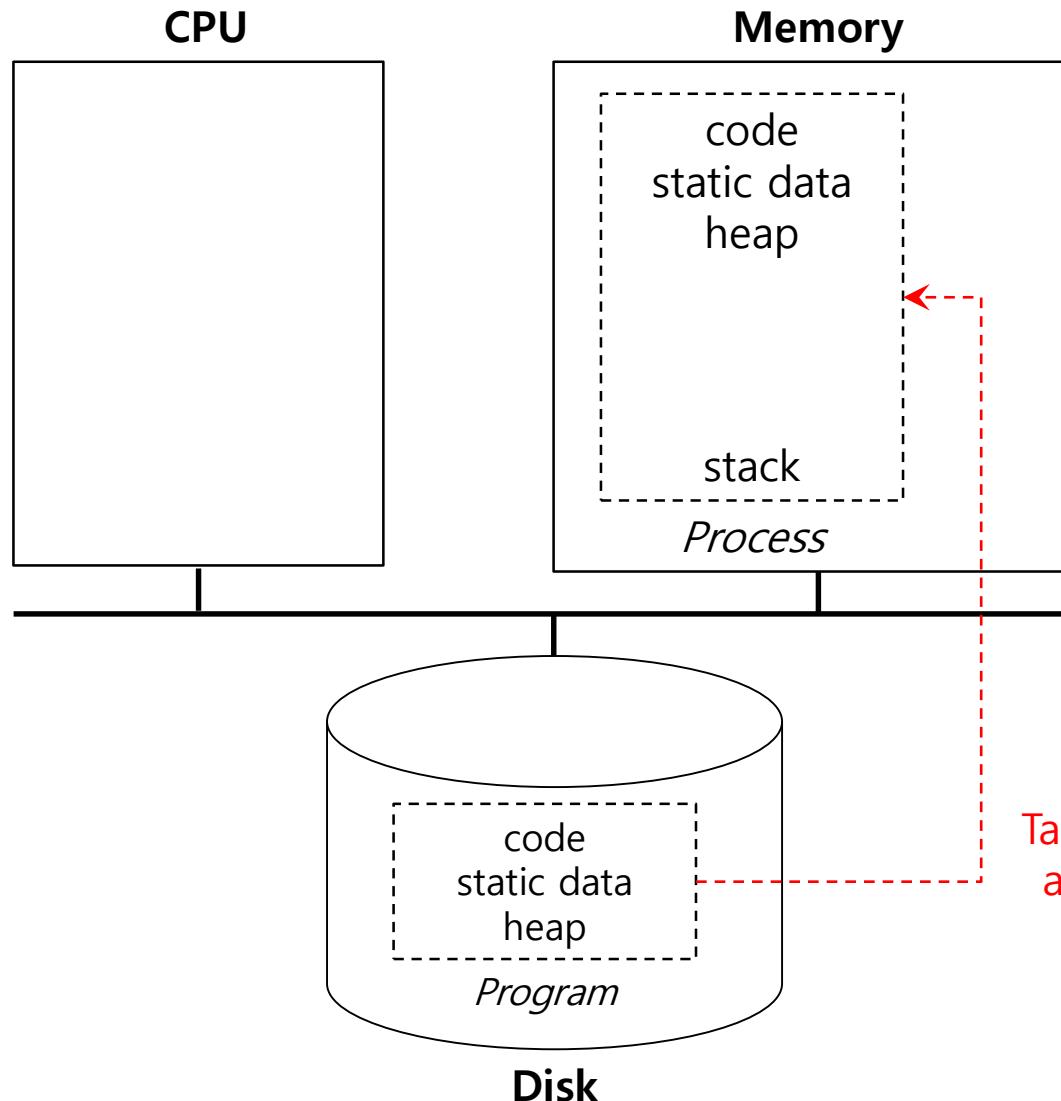
Process Creation

1. **Load** a program code into memory, into the address space of the process.
 - ◆ Programs initially reside on disk in *executable format*.
 - ◆ 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

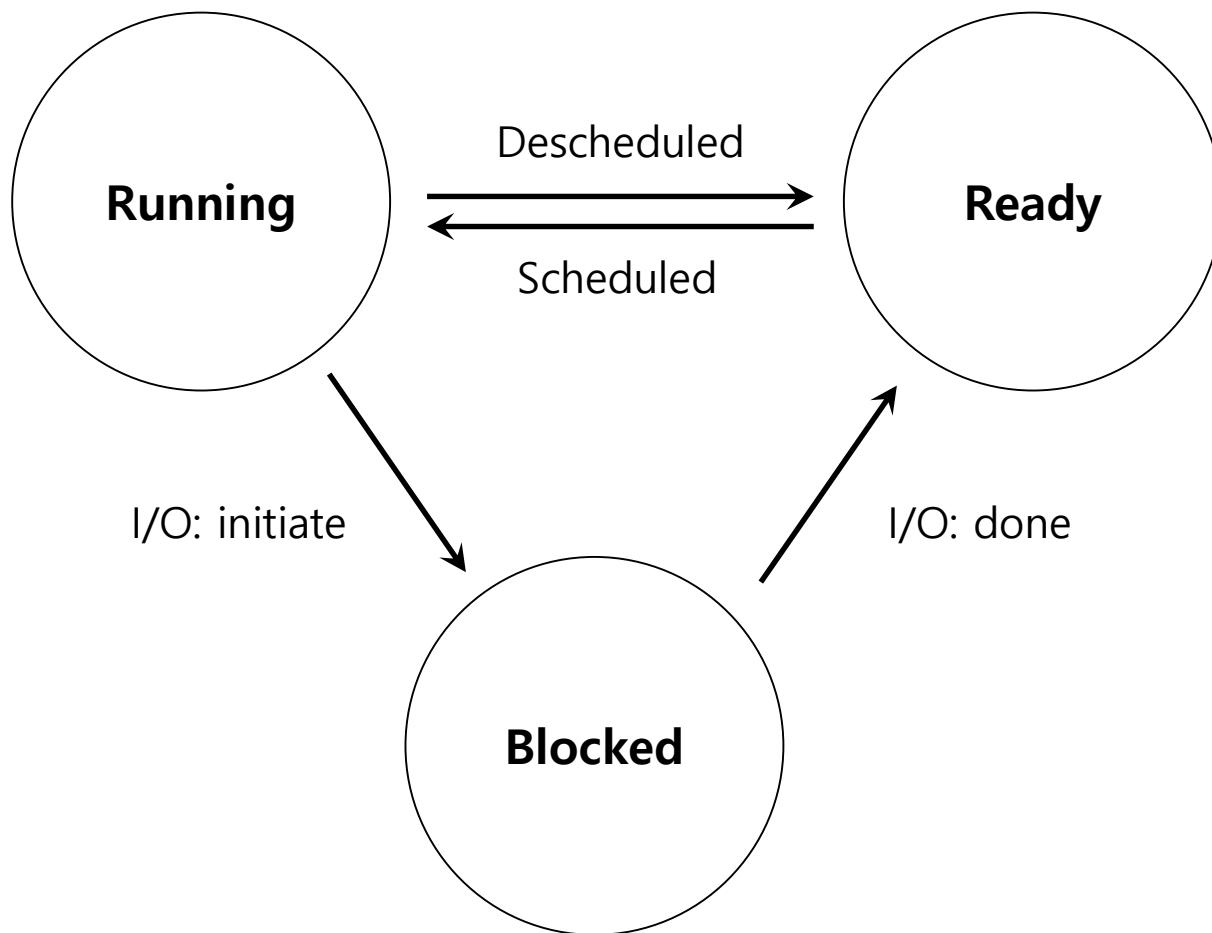


Loading:
Takes on-disk program
and reads it into the
address space of
process

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 };
```

Example) The xv6 kernel Proc Structure (Cont.)

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                      // Start of process memory
    uint sz;                         // Size of process memory
    char *kstack;                    // Bottom of kernel stack
                                    // for this process
    enum proc_state state;          // Process state
    int pid;                         // Process ID
    struct proc *parent;            // Parent process
    void *chan;                      // If non-zero, sleeping on chan
    int killed;                      // If non-zero, have been killed
    struct file *ofile[NOFILE];     // Open files
    struct inode *cwd;              // Current directory
    struct context context;         // Switch here to run process
    struct trapframe *tf;           // Trap frame for the
                                    // current interrupt
};
```

5. Interlude: Process API

The fork() System Call

- >Create a new process

- The newly-created process has its own copy of the **address space**, **registers**, and **PC**.

p1.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;
}
```

Calling fork() example (Cont.)

Result (Not deterministic)

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

or

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

The wait() System Call

- ▣ This system call won't return until the child has run and exited.

p2.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;
}
```

The wait() System Call (Cont.)

Result (Deterministic)

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

The exec() System Call

- Run a program that is different from the calling program

p3.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
        ...
    }
}
```

The exec() System Call (Cont.)

p3.c (Cont.)

```
...
    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;
}
```

Result

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

All of the above with redirection

p4.c

```
#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);
        ...
    }
}
```

All of the above with redirection (Cont.)

p4.c

```
...
// 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;
}
```

Result

```
prompt> ./p4
prompt> cat p4.output
32 109 846 p4.c
prompt>
```

6. Mechanism: Limited Direct Execution

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

- 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 argc / argv5. Clear registers6. Execute call main() <ol style="list-style-type: none">9. Free memory of process10. Remove from process list	<ol style="list-style-type: none">7. Run main()8. Execute return from main()

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
 - ◆ **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

System Call (Cont.)

▫ Trap instruction

- ◆ Jump into the kernel
- ◆ Raise the privilege level to kernel mode

▫ Return-from-trap instruction

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

Limited Direction Execution Protocol

OS @ boot
(kernel mode)

initialize trap table

Hardware

remember address of ...
syscall handler

OS @ run
(kernel 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

Hardware

restore regs from kernel stack
move to user mode
jump to main

Program
(user mode)

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

Limited Direction Execution Protocol (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
<i>(Cont.)</i>		
	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())
Free memory of process Remove from process list		

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.
- ◆ The timer raises an interrupt every so many milliseconds.
- ◆ 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  
10    movl %esp, 4(%eax)         # and stack  
11    movl %ebx, 8(%eax)         # and other registers  
12    movl %ecx, 12(%eax)  
13    movl %edx, 16(%eax)  
14    movl %esi, 20(%eax)  
15    movl %edi, 24(%eax)  
16    movl %ebp, 28(%eax)  
17  
18     # Load new registers  
19    movl 4(%esp), %eax          # put new ptr into eax  
20    movl 28(%eax), %ebp          # restore other registers  
21    movl 24(%eax), %edi  
22    movl 20(%eax), %esi  
23    movl 16(%eax), %edx  
24    movl 12(%eax), %ecx  
25    movl 8(%eax), %ebx  
26    movl 4(%eax), %esp          # stack is switched here  
27    pushl 0(%eax)              # return addr put in place  
28    ret                         # finally return into new ctxt
```

Worried About Concurrency?

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

7. Scheduling: Introduction

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.

Scheduling Metrics

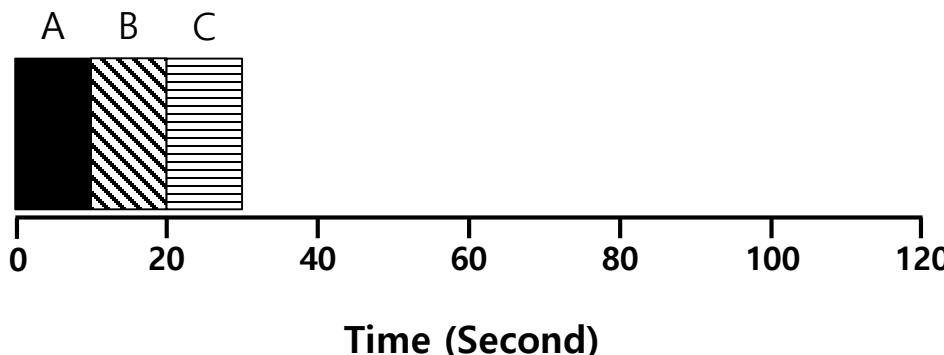
- ❑ Performance metric: Turnaround time
 - ◆ The time at which **the job completes** minus the time at which **the job arrived** in the system.

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

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

First In, First Out (FIFO)

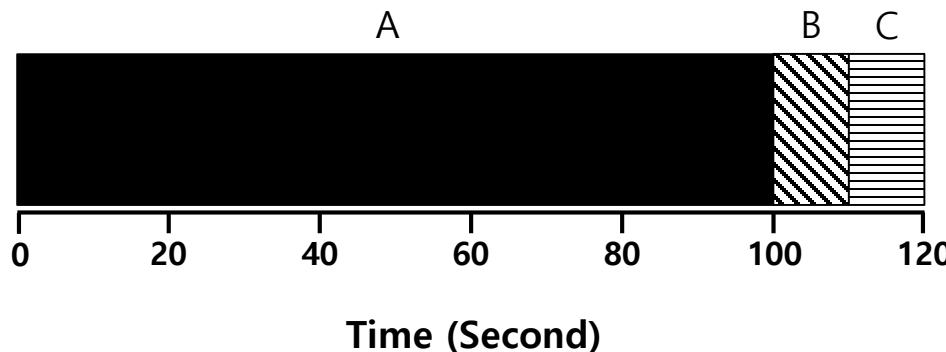
- ▣ First Come, First Served (FCFS)
 - ◆ Very simple and easy to implement
- ▣ Example:
 - ◆ A arrived just before B which arrived just before C.
 - ◆ Each job runs for 10 seconds.



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

Why FIFO is not that great? – Convoy effect

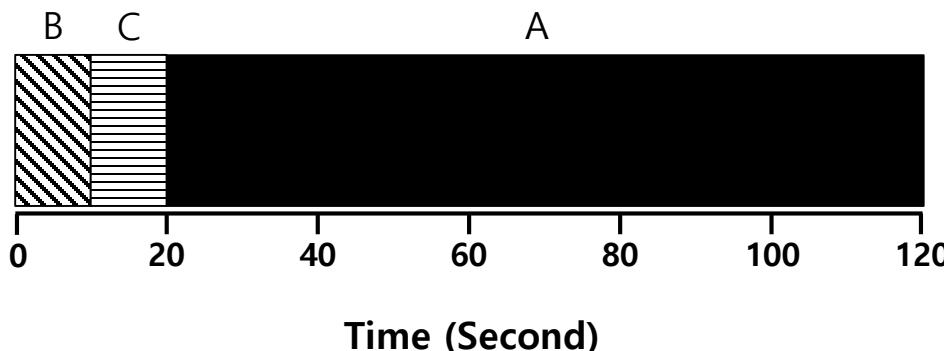
- Let's relax assumption 1: Each job **no longer** runs for the same amount of time.
- Example:
 - A arrived just before B which arrived just before C.
 - A runs for 100 seconds, B and C run for 10 each.



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

Shortest Job First (SJF)

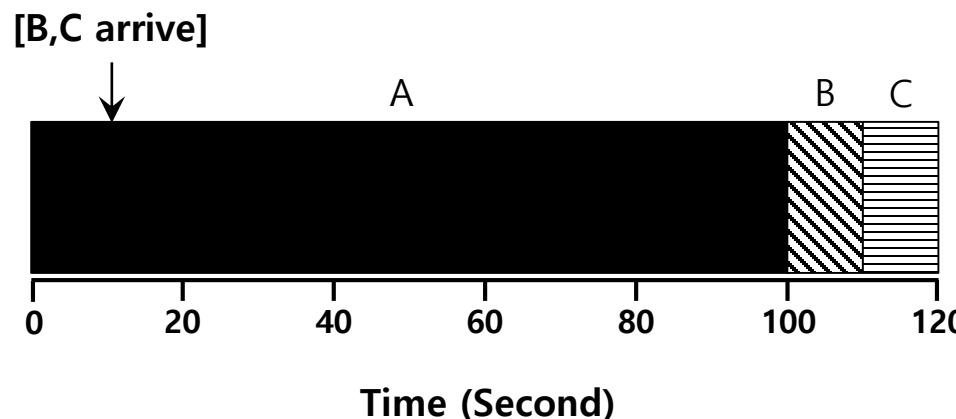
- ▣ Run the shortest job first, then the next shortest, and so on
 - ◆ Non-preemptive scheduler
- ▣ Example:
 - ◆ A arrived just before B which arrived just before C.
 - ◆ A runs for 100 seconds, B and C run for 10 each.



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

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



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

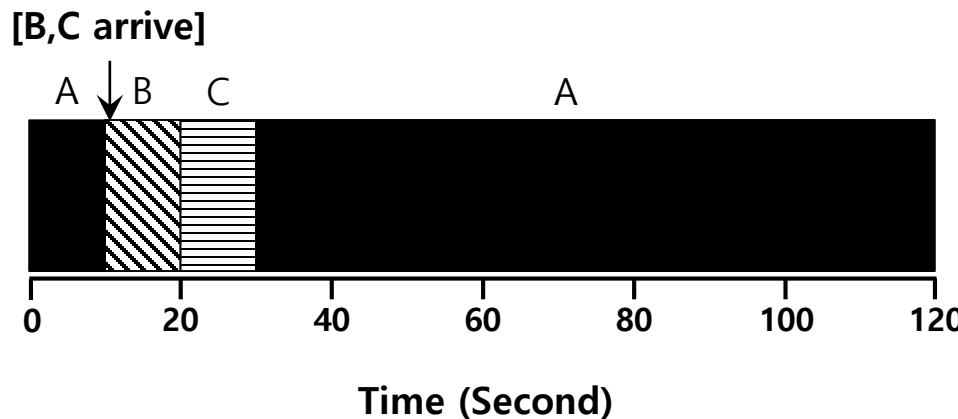
Shortest Time-to-Completion First (STCF)

- ▣ 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

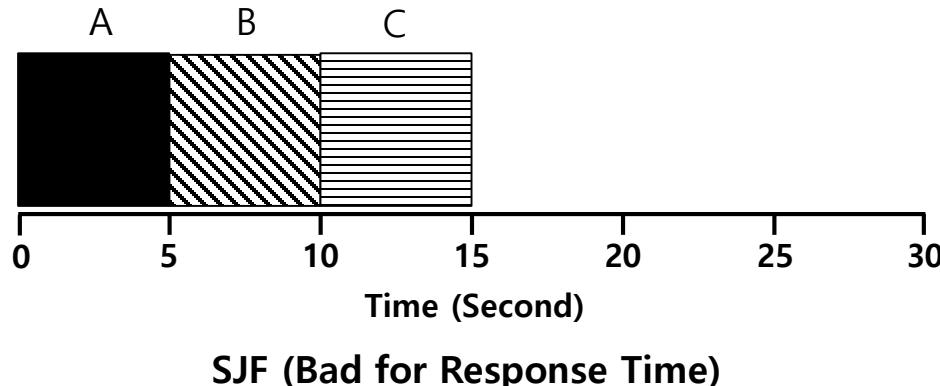
▣ Time slicing Scheduling

- ◆ 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.

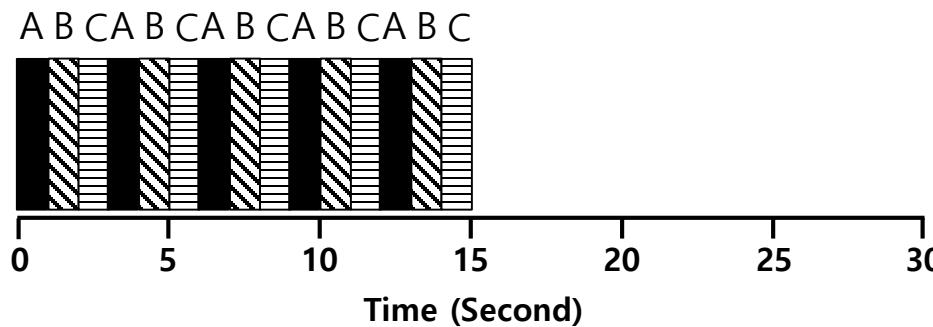
RR is fair, but performs poorly on metrics such as turnaround time

RR Scheduling Example

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



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



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

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

The length of the time slice is critical.

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

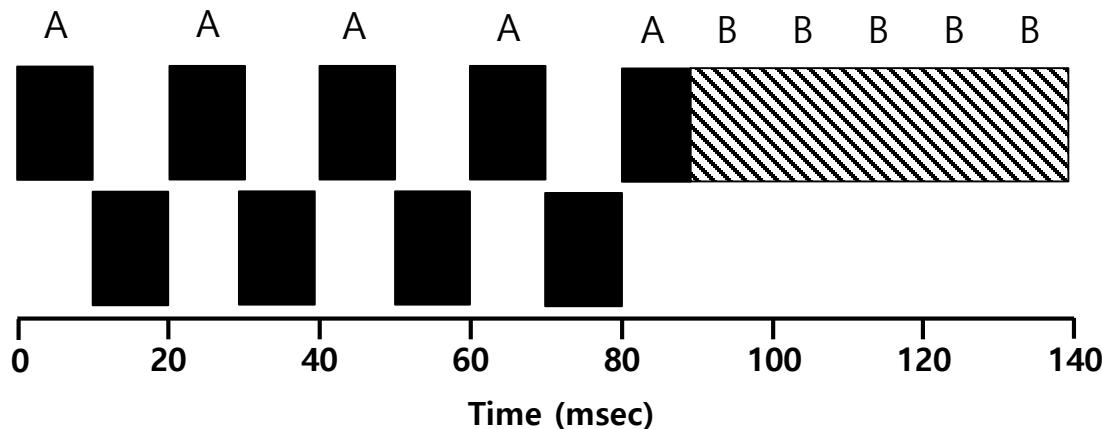
- ▣ The longer time slice
 - ◆ Amortize the cost of switching
 - ◆ Worse response time

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

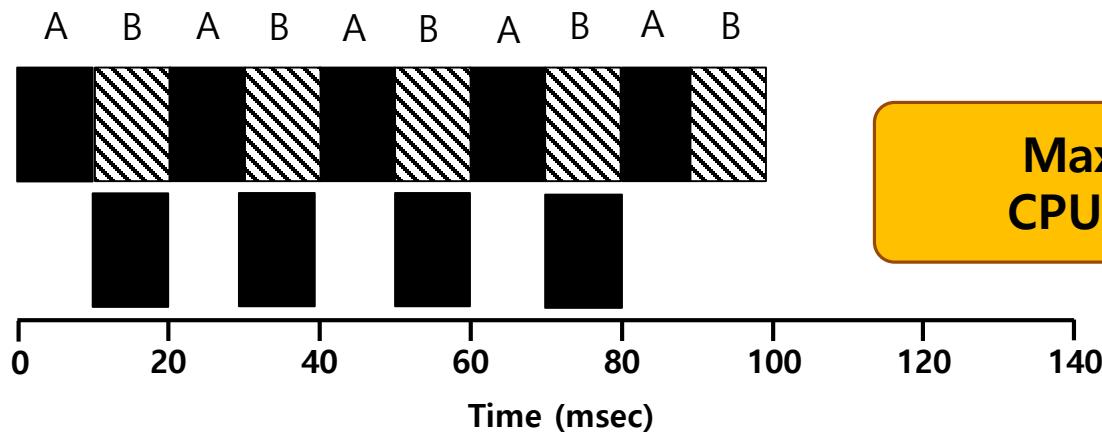
Incorporating I/O

- ▣ 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.

8: Scheduling: The Multi-Level Feedback Queue

Multi-Level Feedback Queue (MLFQ)

- ▣ A Scheduler that learns from the past to predict the future.
- ▣ Objective:
 - ◆ Optimize **turnaround time** → Run shorter jobs first
 - ◆ Minimize **response time** without *a priori knowledge of job length*.

MLFQ: Basic Rules

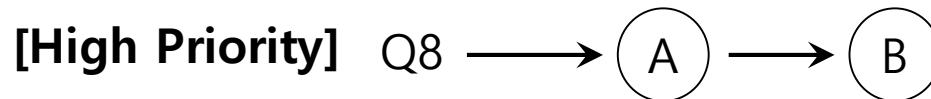
- ▣ MLFQ has a number of distinct **queues**.
 - ◆ Each queue is assigned a different priority level.
- ▣ A job that is ready to run is on a single queue.
 - ◆ A job **on a higher queue** is chosen to run.
 - ◆ Use round-robin scheduling among jobs in the same queue

Rule 1: If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).
Rule 2: If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR.

MLFQ: Basic Rules (Cont.)

- ▣ MLFQ varies the priority of a job based on *its observed behavior*.
- ▣ Example:
 - ◆ A job repeatedly relinquishes the CPU while waiting IOs → Keep its priority high
 - ◆ A job uses the CPU intensively for long periods of time → Reduce its priority.

MLFQ Example



Q7

Q6

Q5



Q3

Q2



MLFQ: How to Change Priority

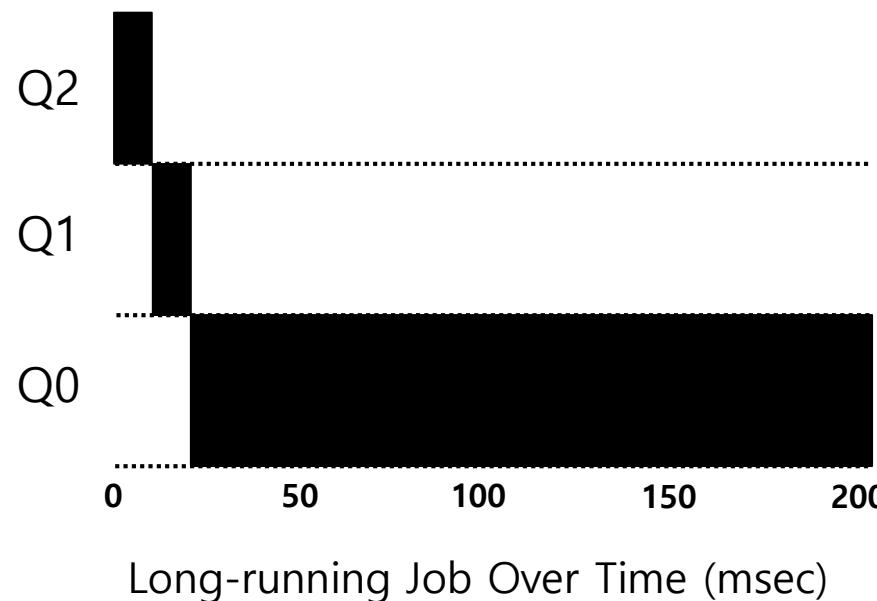
- MLFQ priority adjustment algorithm:

- ◆ **Rule 3:** When a job enters the system, it is placed at the highest priority
- ◆ **Rule 4a:** If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down on queue).
- ◆ **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the same priority level

In this manner, MLFQ approximates SJF

Example 1: A Single Long-Running Job

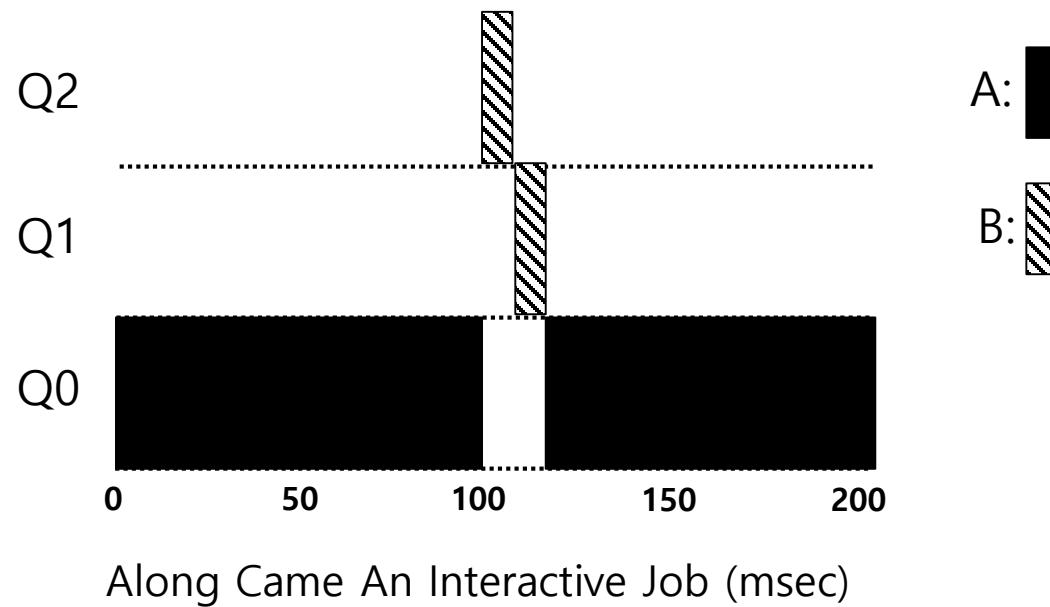
- A three-queue scheduler with time slice 10ms



Example 2: Along Came a Short Job

- Assumption:

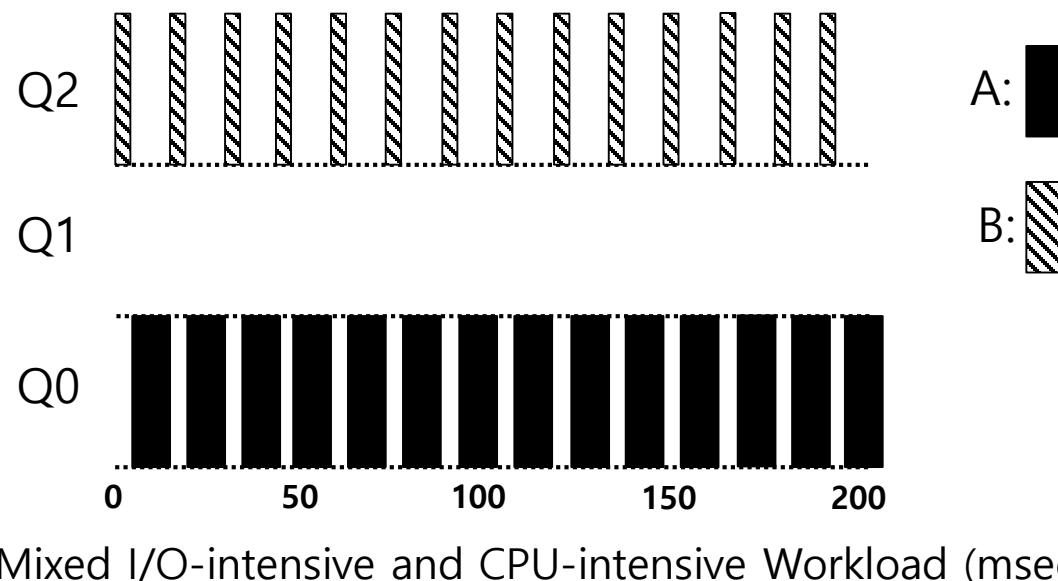
- Job A: A long-running CPU-intensive job
- Job B: A short-running interactive job (20ms runtime)
- A has been running for some time, and then B arrives at time T=100.



Example 3: What About I/O?

- Assumption:

- Job A: A long-running CPU-intensive job
- Job B: An interactive job that need the CPU only for 1ms before performing an I/O



The MLFQ approach keeps an interactive job at the highest priority

Problems with the Basic MLFQ

- Starvation
 - ◆ If there are “too many” interactive jobs in the system.
 - ◆ Long-running jobs will never receive any CPU time.

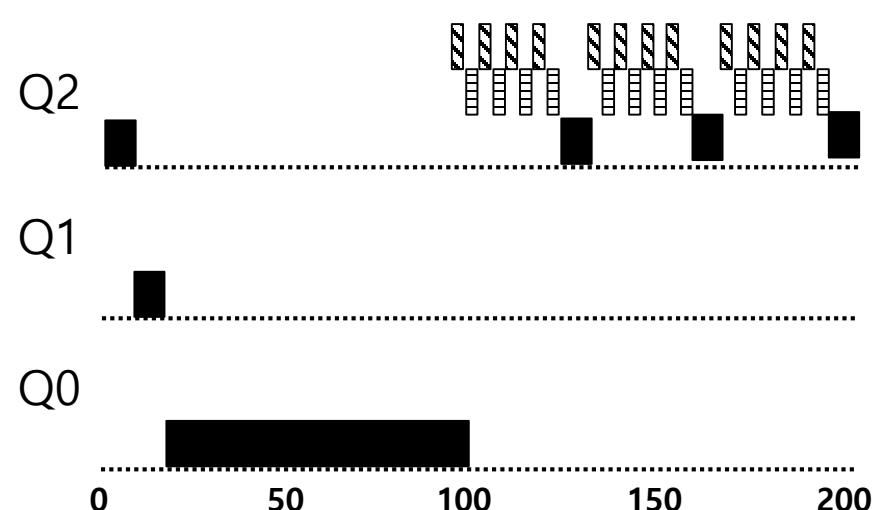
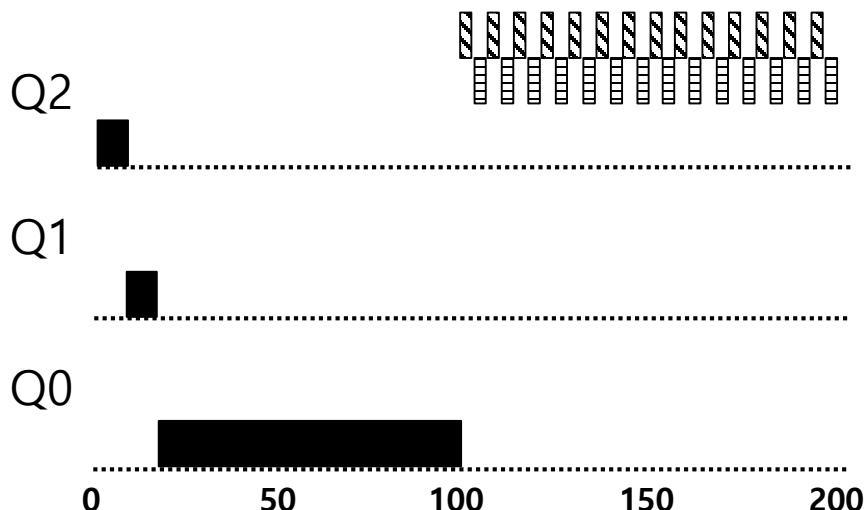
- Game the scheduler
 - ◆ After running 99% of a time slice, issue an I/O operation.
 - ◆ The job gain a higher percentage of CPU time.

- A program may change its behavior over time.

- ◆ CPU bound process → I/O bound process

The Priority Boost

- **Rule 5:** After some time period S , move all the jobs in the system to the topmost queue.
 - ◆ Example:
 - A long-running job(A) with two short-running interactive job(B, C)

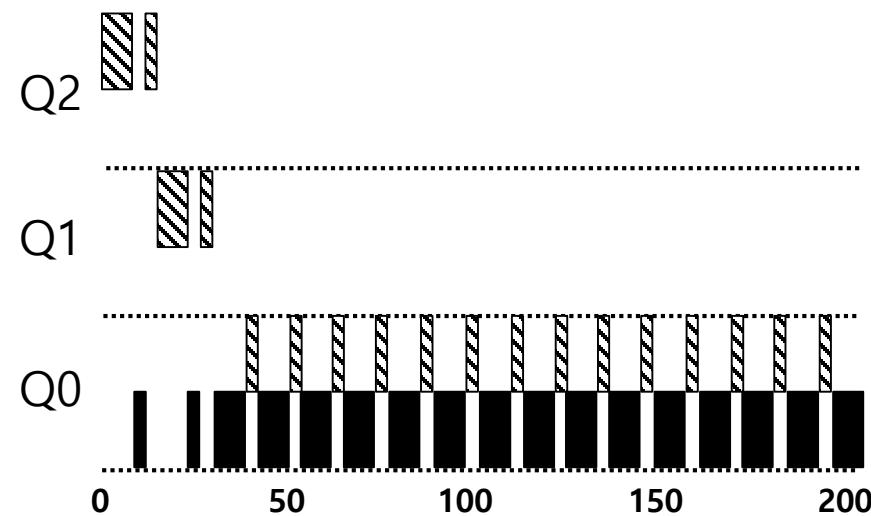
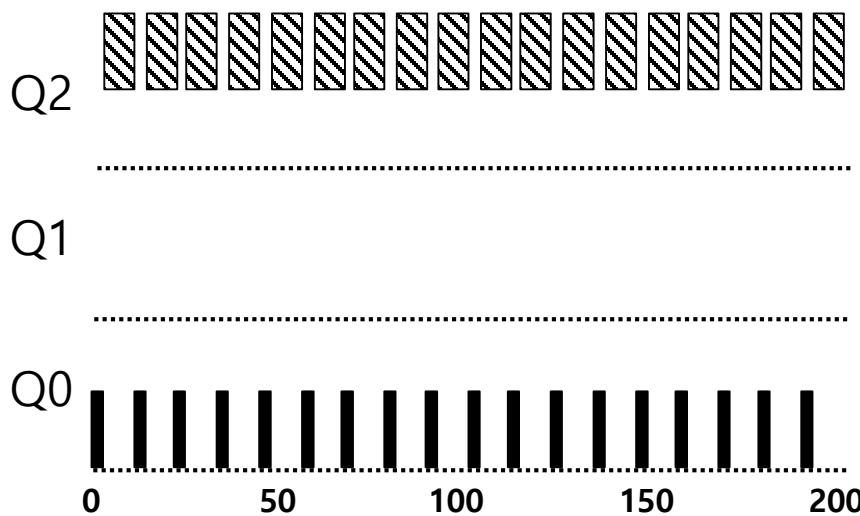


Without(Left) and With(Right) Priority Boost

A: B: C:

Better Accounting

- How to prevent gaming of our scheduler?
- Solution:
 - ◆ **Rule 4** (Rewrite Rules 4a and 4b): Once a job **uses up its time allotment** at a given level (regardless of how many times it has given up the CPU), **its priority is reduced**(i.e., it moves down on queue).

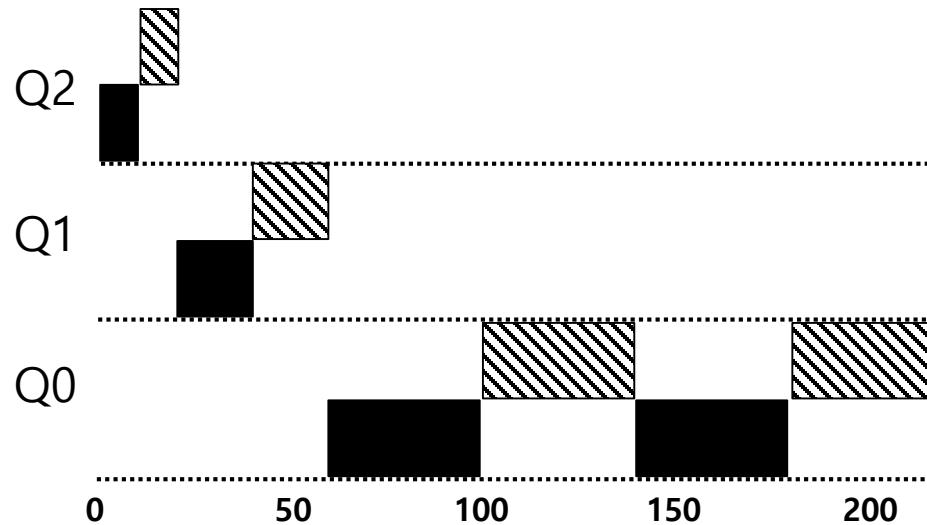


Without(Left) and With(Right) Gaming Tolerance

Tuning MLFQ And Other Issues

Lower Priority, Longer Quanta

- ◆ The high-priority queues → Short time slices
 - E.g., 10 or fewer milliseconds
- ◆ The Low-priority queue → Longer time slices
 - E.g., 100 milliseconds



Example) 10ms for the highest queue, 20ms for the middle,
40ms for the lowest

The Solaris MLFQ implementation

- ▣ For the Time-Sharing scheduling class (TS)
 - ◆ 60 Queues
 - ◆ Slowly increasing time-slice length
 - The highest priority: 20msec
 - The lowest priority: A few hundred milliseconds
 - ◆ Priorities boosted around every 1 second or so.

MLFQ: Summary

- The refined set of MLFQ rules:

- ◆ **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- ◆ **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.
- ◆ **Rule 3:** When a job enters the system, it is placed at the highest priority.
- ◆ **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced(i.e., it moves down on queue).
- ◆ **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

9: Scheduling: Proportional Share

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 A B A B A

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

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)

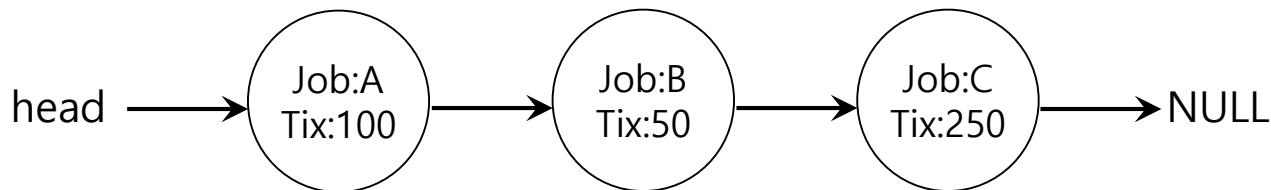
Ticket Mechanisms (Cont.)

- ▣ 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:



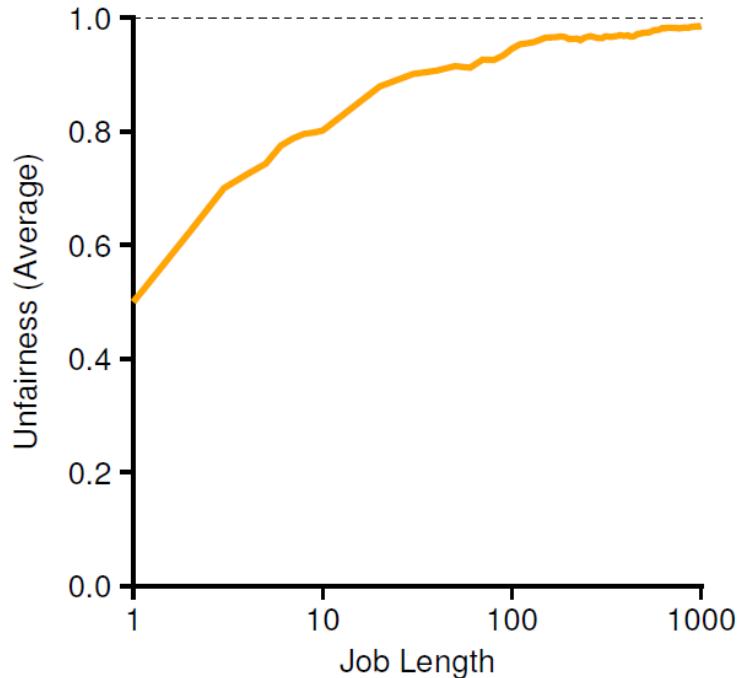
```
1 // counter: used to track if we've found the winner yet
2 int counter = 0;
3
4 // winner: use some call to a random number generator to
5 // get a value, between 0 and the total # of tickets
6 int winner = getrandom(0, totaltickets);
7
8 // current: use this to walk through the list of jobs
9 node_t *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

- ▣ **Stride** of each process
 - ◆ (A large number) / (the number of tickets of the process)
 - ◆ Example: A large number = 10,000
 - Process A has 100 tickets → stride of A is 100
 - Process B has 50 tickets → stride of B is 200
- ▣ 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!

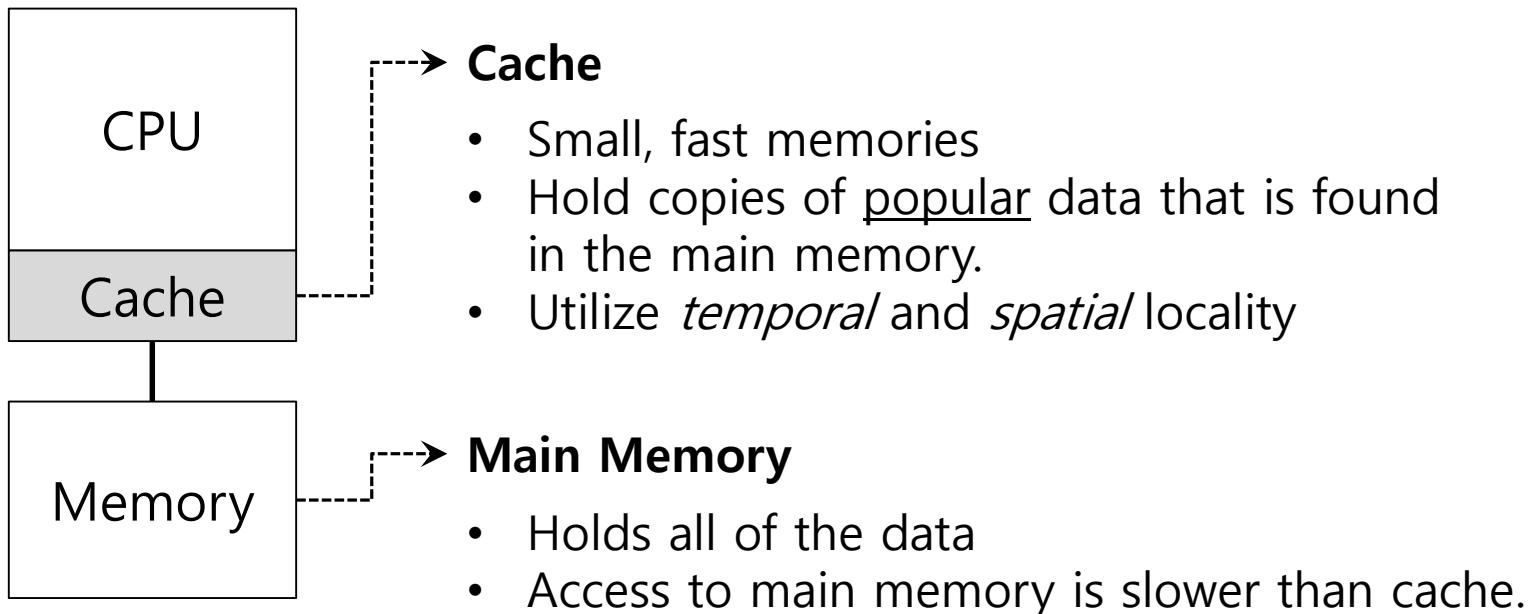
10. Multiprocessor Scheduling (Advanced)

Multiprocessor Scheduling

- ❑ The rise of the **multicore processor** is the source of multiprocessor-scheduling proliferation.
 - ◆ **Multicore:** Multiple CPU cores are packed onto a single chip.
- ❑ Adding more CPUs does not make that single application run faster.
→ You'll have to rewrite application to run in parallel, using **threads**.

How to schedule jobs on **Multiple CPUs?**

Single CPU with cache

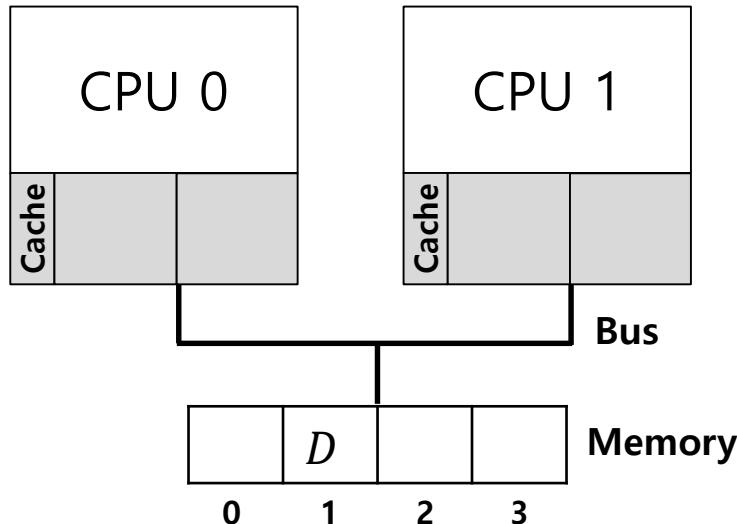


By keeping data in cache, the system can make slow memory appear to be a fast one

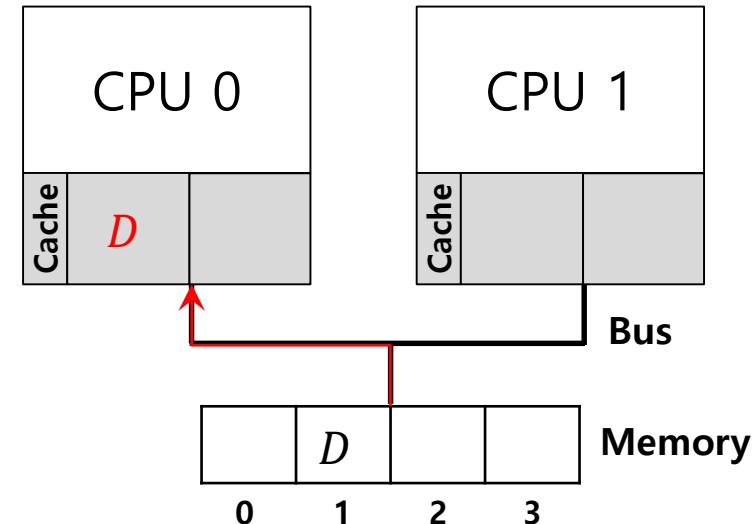
Cache coherence

- Consistency of shared resource data stored in multiple caches.

0. Two CPUs with caches sharing memory

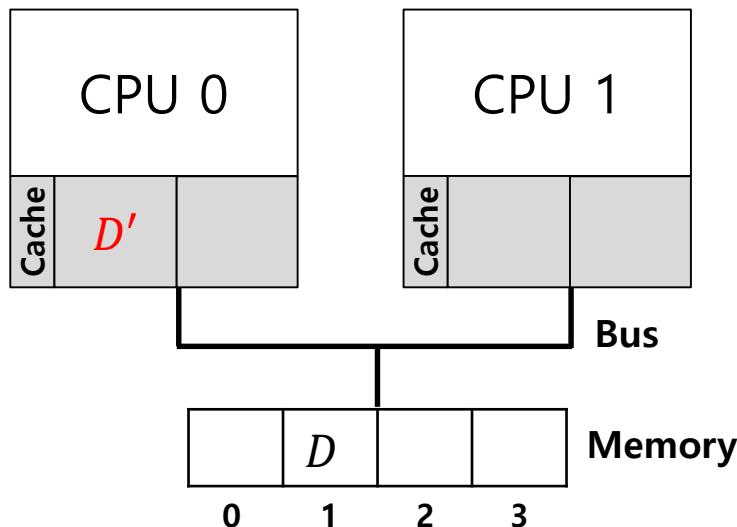


1. CPU0 reads a data at address 1.

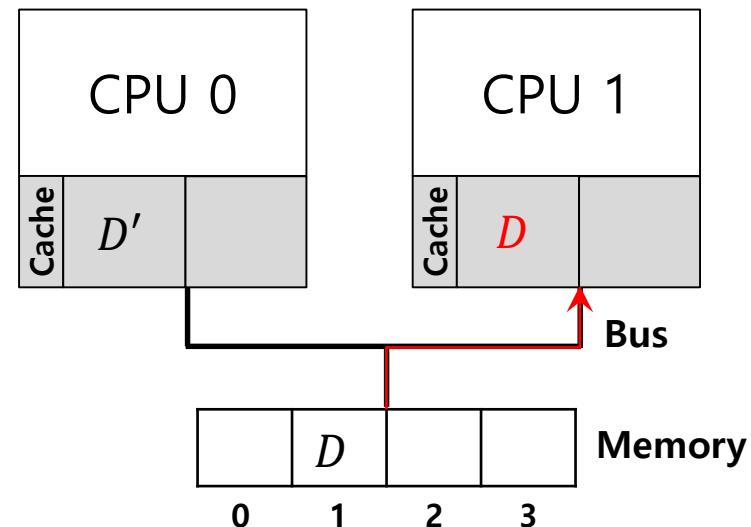


Cache coherence (Cont.)

2. D is updated and CPU1 is scheduled.



3. CPU1 re-reads the value at address A



CPU1 gets the **old value D** instead of the correct value D' .

Cache coherence solution

- ❑ Bus snooping

- ◆ Each cache pays attention to memory updates by **observing the bus**.
- ◆ When a CPU sees an update for a data item it holds in its cache, it will notice the change and either invalidate its copy or update it.

Don't forget synchronization

- When accessing shared data across CPUs, **mutual exclusion** primitives should likely be used to guarantee correctness.

```
1     typedef struct __Node_t {
2         int value;
3         struct __Node_t *next;
4     } Node_t;
5
6     int List_Pop() {
7         Node_t *tmp = head;           // remember old head ...
8         int value = head->value;    // ... and its value
9         head = head->next;          // advance head to next pointer
10        free(tmp);                // free old head
11        return value;              // return value at head
12    }
```

Simple List Delete Code

Don't forget synchronization (Cont.)

▣ Solution

```
1      pthread_mutex_t m;
2      typedef struct __Node_t {
3          int value;
4          struct __Node_t *next;
5      } Node_t;
6
7      int List_Pop() {
8          lock(&m)
9          Node_t *tmp = head;           // remember old head ...
10         int value = head->value;   // ... and its value
11         head = head->next;        // advance head to next pointer
12         free(tmp);                // free old head
13         unlock(&m)
14         return value;             // return value at head
15     }
```

Simple List Delete Code with lock

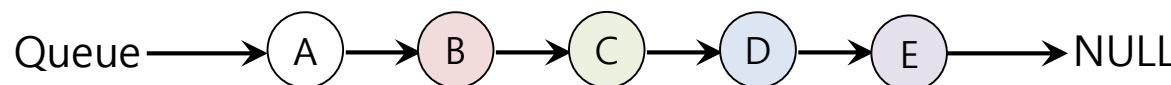
Cache Affinity

- ▣ Keep a process on **the same CPU** if at all possible
 - ◆ A process builds up a fair bit of state in the cache of a CPU.
 - ◆ The next time the process runs, it will run faster if some of its state is *already present* in the cache on that CPU.

A multiprocessor scheduler should consider **cache affinity** when making its scheduling decision.

Single queue Multiprocessor Scheduling (SQMS)

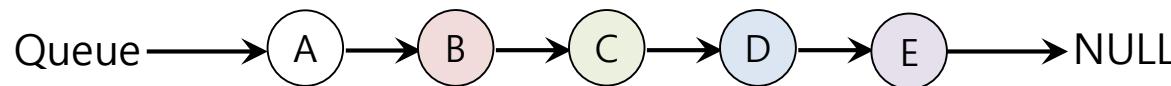
- Put all jobs that need to be scheduled into **a single queue**.
 - Each CPU simply picks the next job from the globally shared queue.
 - Cons:
 - Some form of **locking** have to be inserted → **Lack of scalability**
 - Cache affinity**
 - Example:



- Possible job scheduler across CPUs:

CPU0	A	E	D	C	B	... (repeat) ...
CPU1	B	A	E	D	C	... (repeat) ...
CPU2	C	B	A	E	D	... (repeat) ...
CPU3	D	C	B	A	E	... (repeat) ...

Scheduling Example with Cache affinity



CPU0	A	E	A	A	A	... (repeat) ...
CPU1	B	B	E	B	B	... (repeat) ...
CPU2	C	C	C	E	C	... (repeat) ...
CPU3	D	D	D	D	E	... (repeat) ...

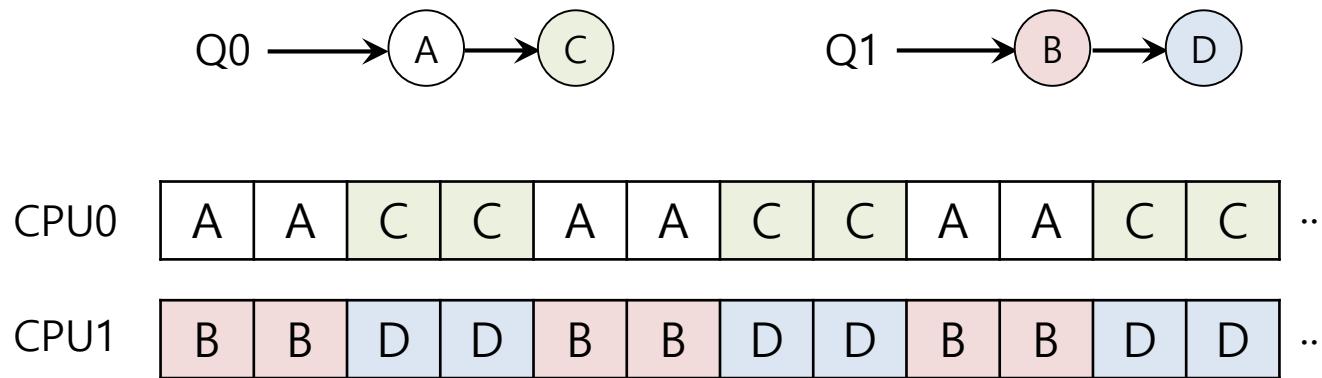
- ◆ Preserving affinity for most
 - Jobs A through D are not moved across processors.
 - Only job e Migrating from CPU to CPU.
- ◆ Implementing such a scheme can be **complex**.

Multi-queue Multiprocessor Scheduling (MQMS)

- ▣ MQMS consists of **multiple scheduling queues**.
 - ◆ Each queue will follow a particular scheduling discipline.
 - ◆ When a job enters the system, it is placed on **exactly one** scheduling queue.
 - ◆ Avoid the problems of information sharing and synchronization.

MQMS Example

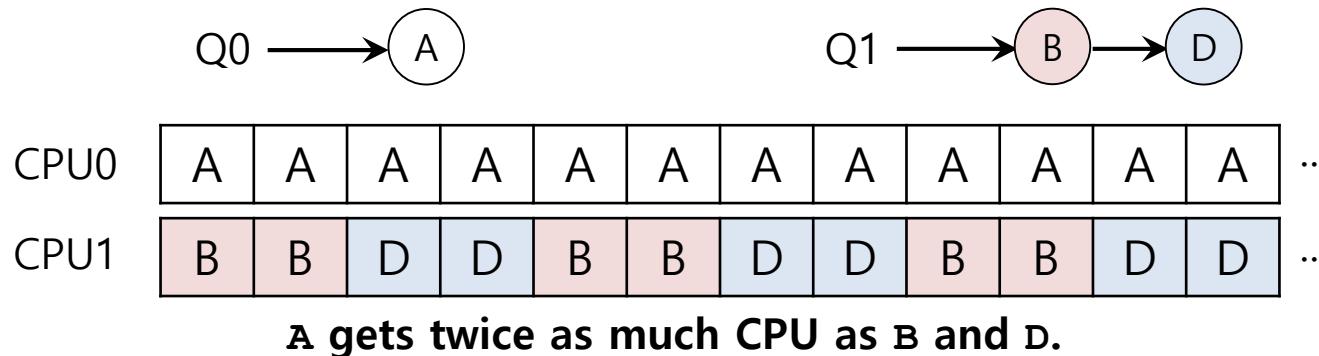
- With **round robin**, the system might produce a schedule that looks like this:



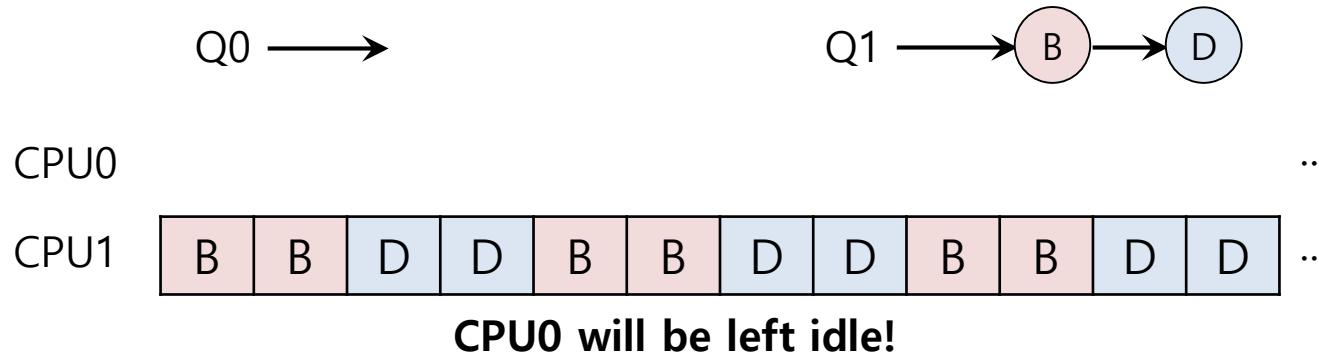
MQMS provides more **scalability** and **cache affinity**.

Load Imbalance issue of MQMS

- After job C in Q0 finishes:

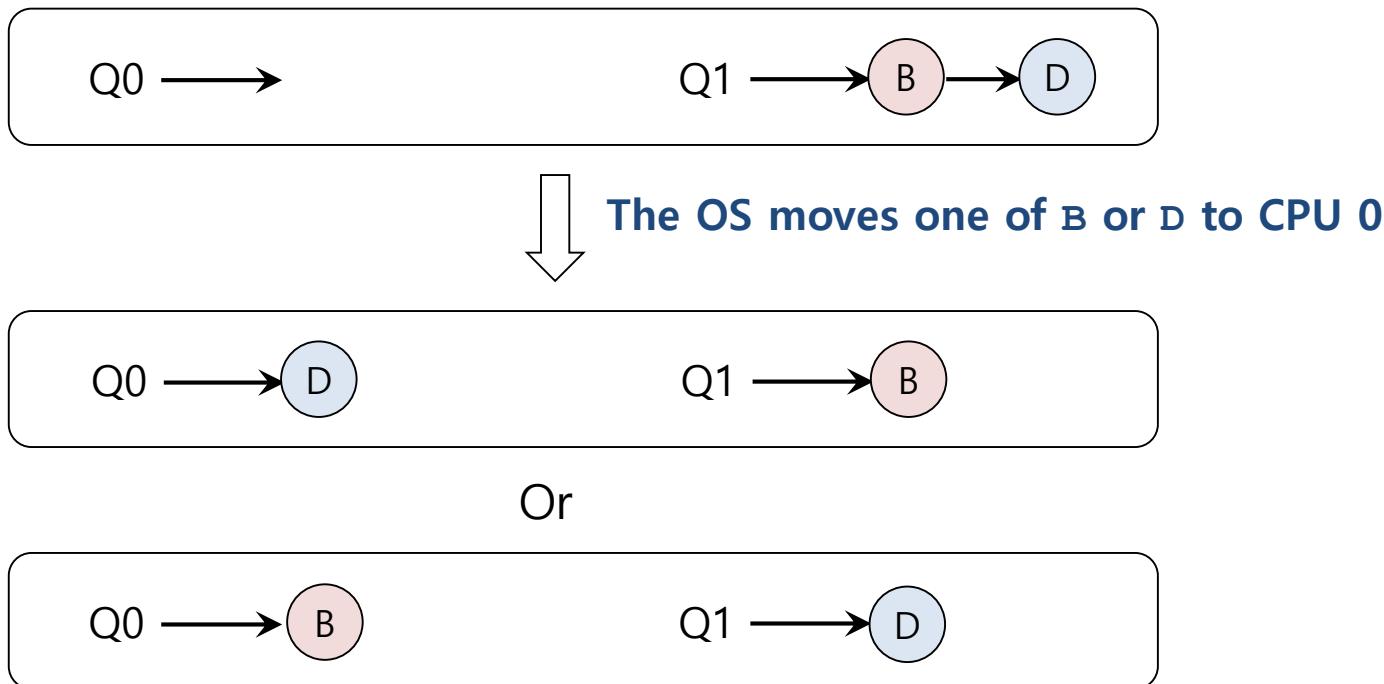


- After job A in Q0 finishes:



How to deal with load imbalance?

- The answer is to move jobs (**Migration**).
 - ◆ Example:



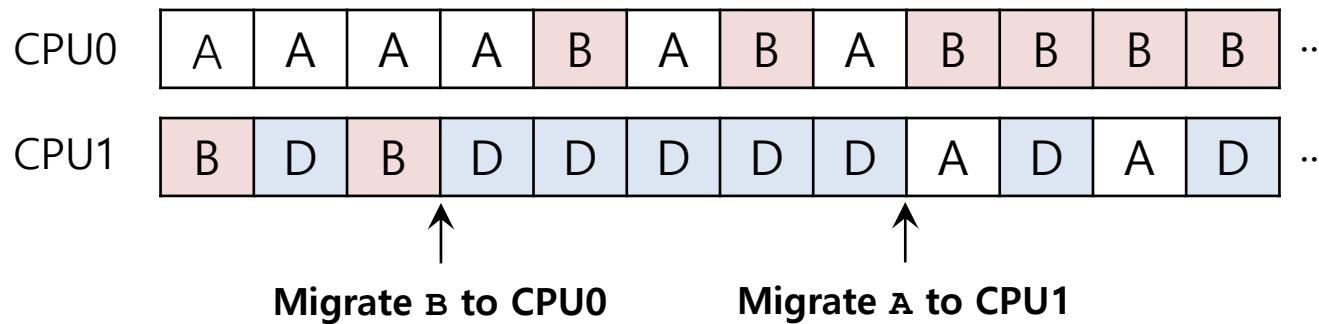
How to deal with load imbalance? (Cont.)

- A more tricky case:



- A possible migration pattern:

- ◆ Keep switching jobs



Work Stealing

- ▣ Move jobs between queues
 - ◆ Implementation:
 - A source queue that is low on jobs is picked.
 - The source queue occasionally peeks at another target queue.
 - If the target queue is more full than the source queue, the source will “**steal**” one or more jobs from the target queue.
 - ◆ Cons:
 - *High overhead* and trouble *scaling*

Linux Multiprocessor Schedulers

- ▣ O(1)
 - ◆ A Priority-based scheduler
 - ◆ Use Multiple queues
 - ◆ Change a process's priority over time
 - ◆ Schedule those with highest priority
 - ◆ Interactivity is a particular focus
- ▣ Completely Fair Scheduler (CFS)
 - ◆ Deterministic proportional-share approach
 - ◆ Multiple queues

Linux Multiprocessor Schedulers (Cont.)

▣ BF Scheduler (BFS)

- ◆ A single queue approach
- ◆ Proportional-share
- ◆ Based on Earliest Eligible Virtual Deadline First(EEVDF)

13. The Abstraction: Address Space

Memory Virtualization

❑ What is **memory virtualization**?

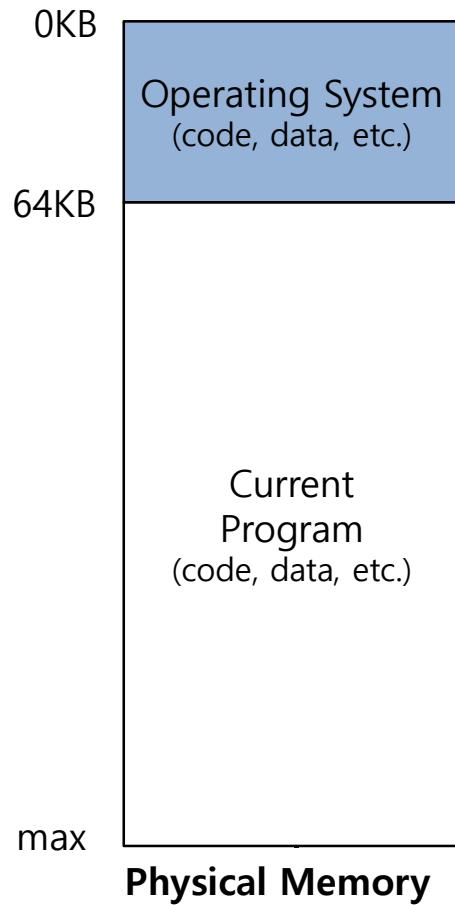
- ◆ OS virtualizes its physical memory.
- ◆ OS provides an **illusion memory space** per each process.
- ◆ It seems to be seen like **each process uses the whole memory** .

Benefit of Memory Virtualization

- ▣ Ease of use in programming
- ▣ Memory efficiency in terms of **times** 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



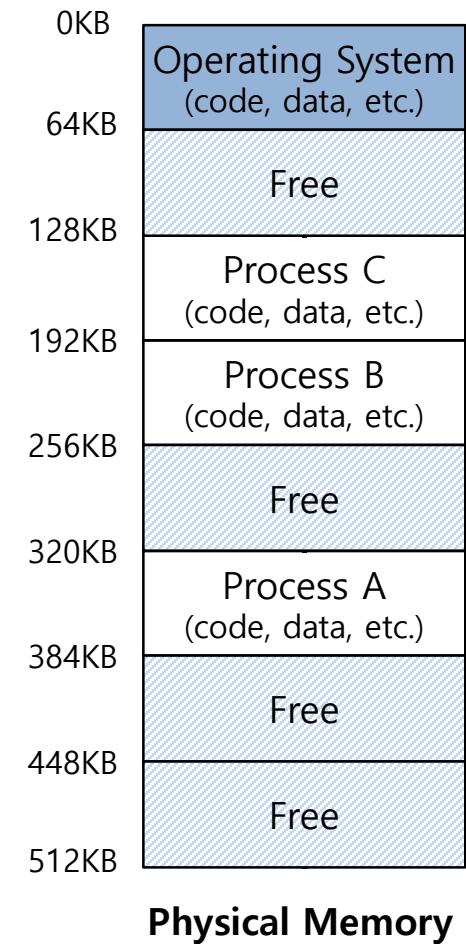
Multiprogramming and Time Sharing

- Load multiple processes in memory.

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

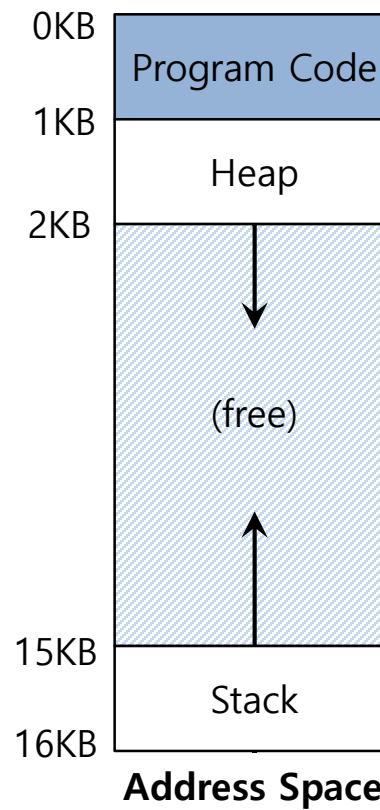
- Cause an important **protection issue**.

- Errant memory accesses from other processes



Address Space

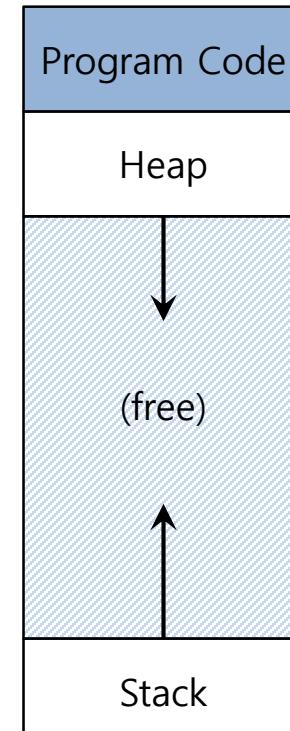
- OS creates an **abstraction** of physical memory.
 - The address space contains all about a running process.
 - That is consist of program code, heap, stack and etc.



▽

Address Space(Cont.)

- Code
 - ◆ Where instructions live
- Heap
 - ◆ Dynamically allocate memory.
 - malloc in C language
 - new in object-oriented language
- Stack
 - ◆ Store return addresses or values.
 - ◆ Contain local variables arguments to routines.



Address Space

Virtual Address

- Every address in a running program is virtual.
 - OS translates the virtual address to physical address

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

int main(int argc, char *argv[]) {

    printf("location of code : %p\n", (void *) main);
    printf("location of heap : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

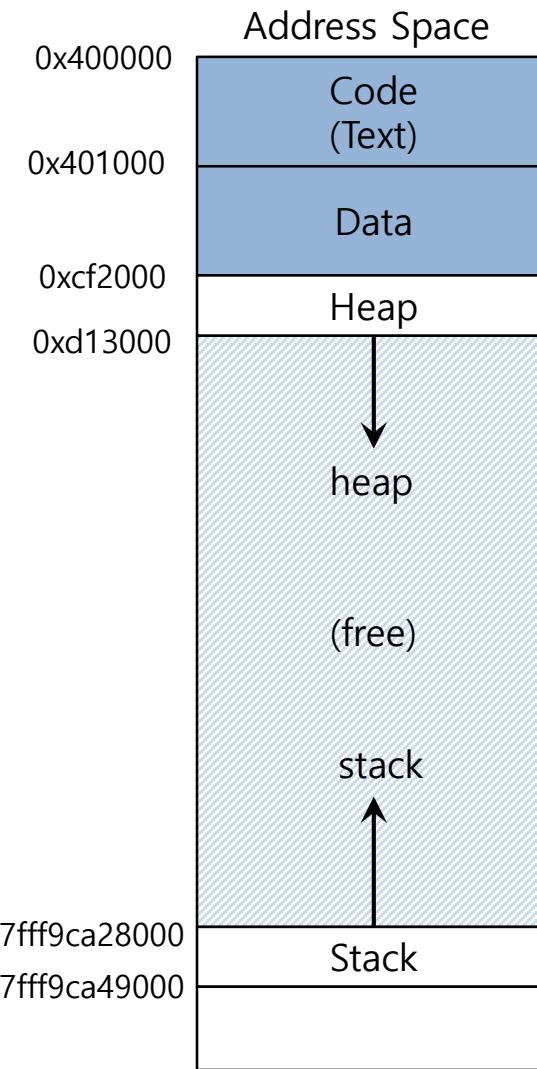
    return x;
}
```

A simple program that prints out addresses

Virtual Address(Cont.)

- The output in 64-bit Linux machine

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



14. Memory API

Memory API: malloc()

```
#include <stdlib.h>

void* malloc(size_t size)
```

- ▣ Allocate a memory region on the heap.

- ◆ Argument

- size_t size : size of the memory block(in bytes)
 - size_t is an unsigned integer type.

- ◆ Return

- Success : a void type pointer to the memory block allocated by malloc
 - Fail : a null pointer

sizeof()

- Routines and macros are utilized for size in malloc instead typing in a number directly.
- Two types of results of sizeof with variables
 - ◆ The actual size of 'x' is known at run-time.

```
int *x = malloc(10 * sizeof(int));  
printf("%d\n", sizeof(x));
```

4

- ◆ The actual size of 'x' is known at compile-time.

```
int x[10];  
printf("%d\n", sizeof(x));
```

40

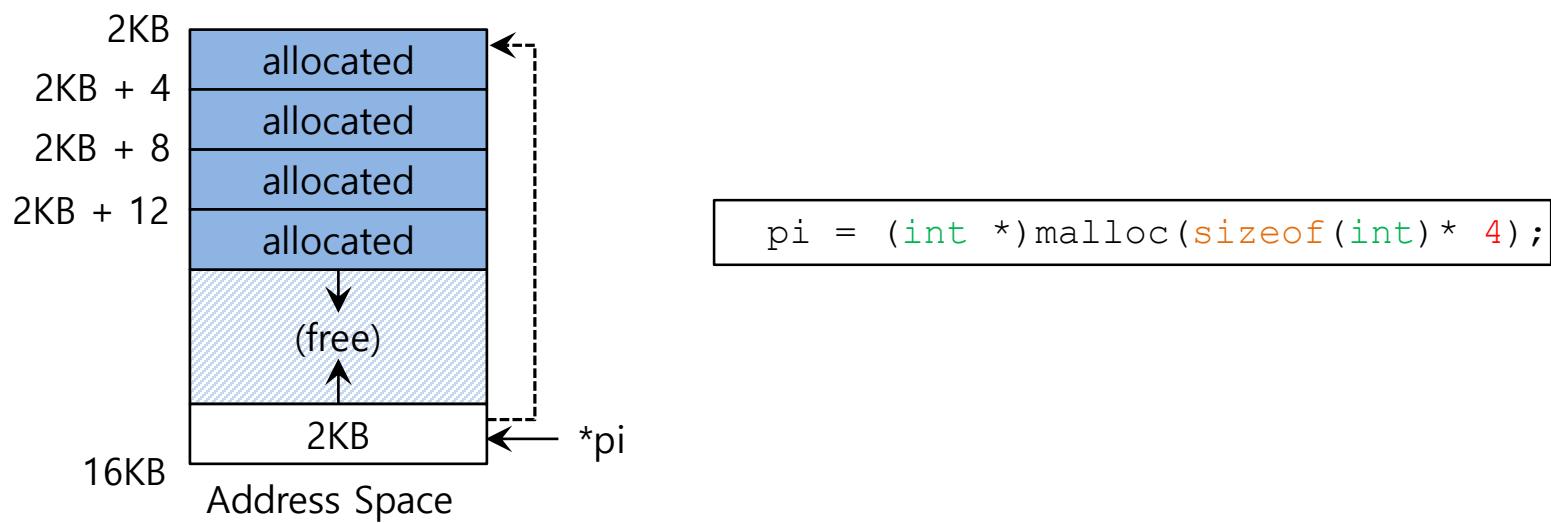
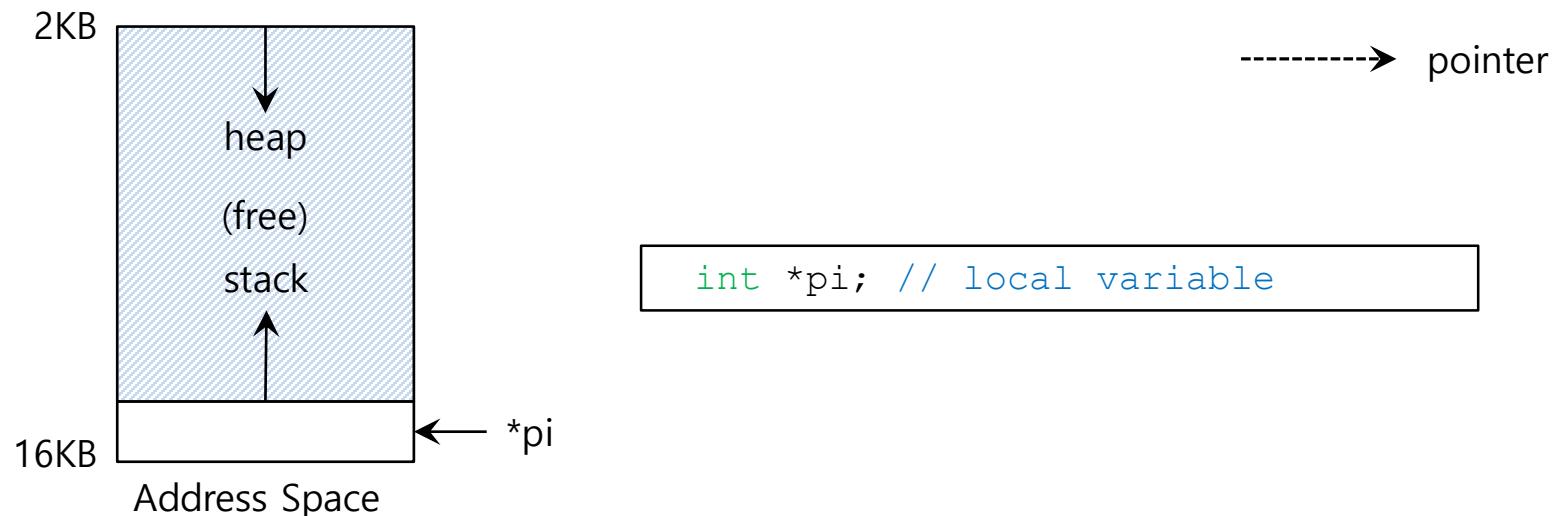
Memory API: free()

```
#include <stdlib.h>

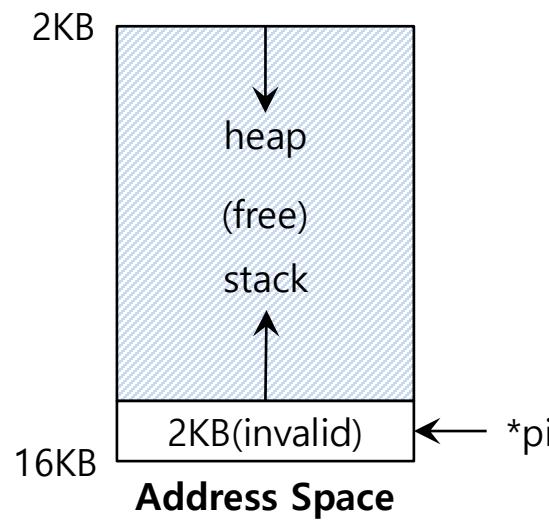
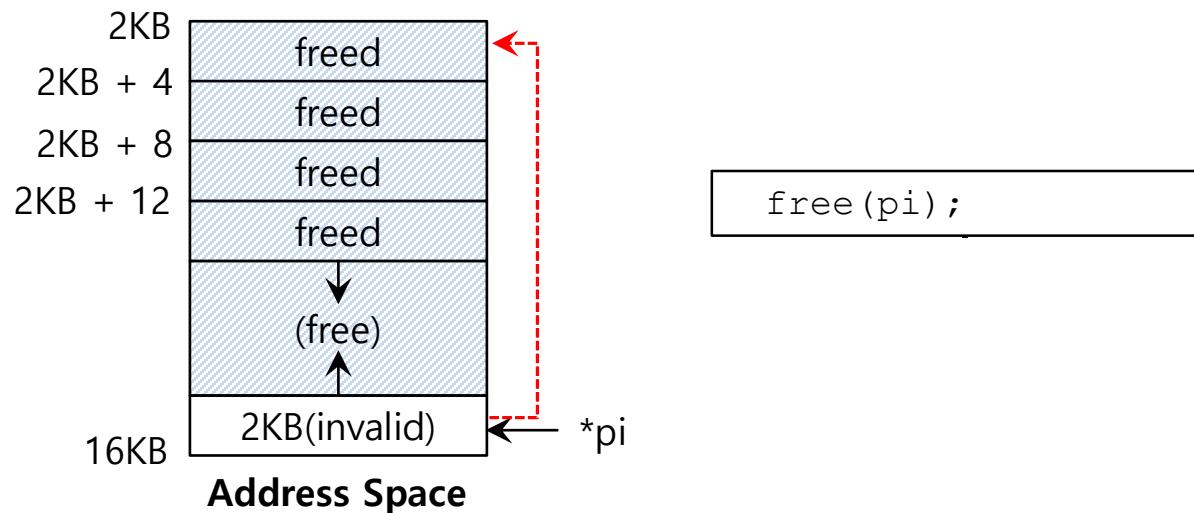
void free(void* ptr)
```

- ▣ Free a memory region allocated by a call to malloc.
 - ◆ Argument
 - void *ptr : a pointer to a memory block allocated with malloc
 - ◆ Return
 - none

Memory Allocating



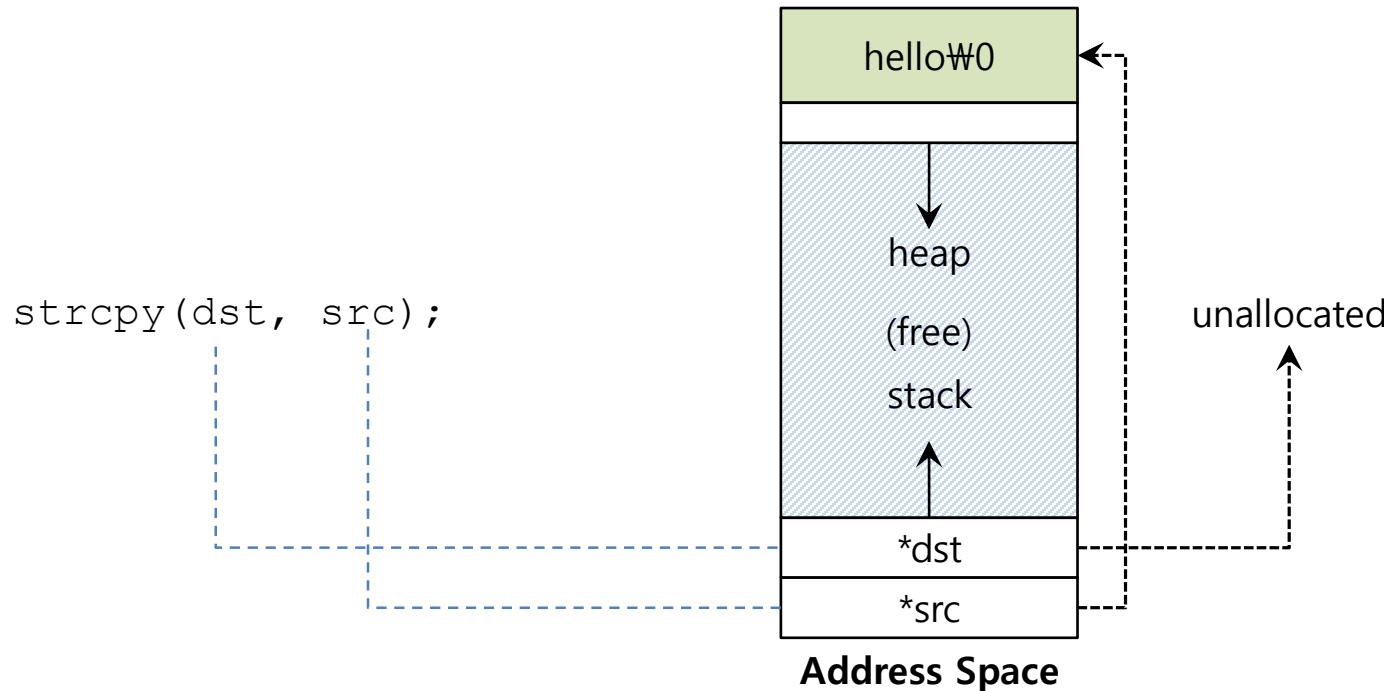
Memory Freeing



Forgetting To Allocate Memory

Incorrect code

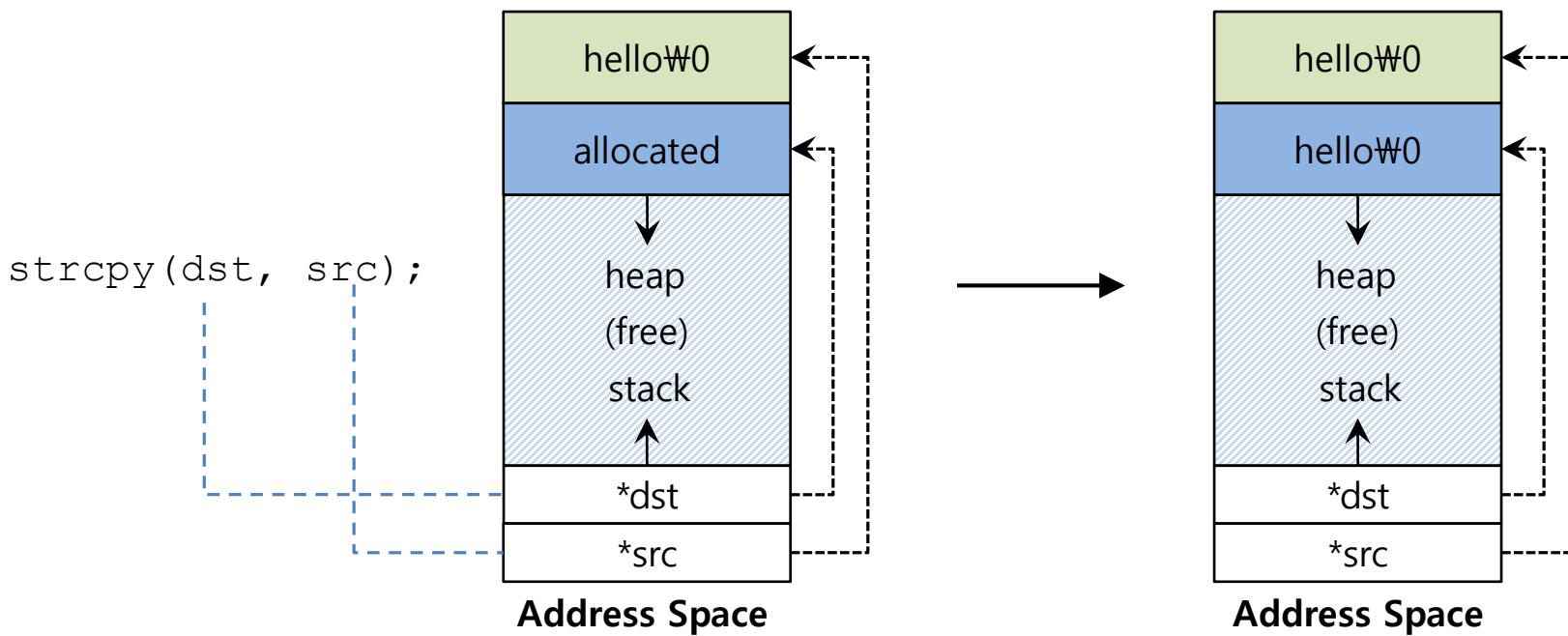
```
char *src = "hello"; //character string constant  
char *dst;           //unallocated  
strcpy(dst, src);   //segfault and die
```



Forgetting To Allocate Memory(Cont.)

❑ Correct code

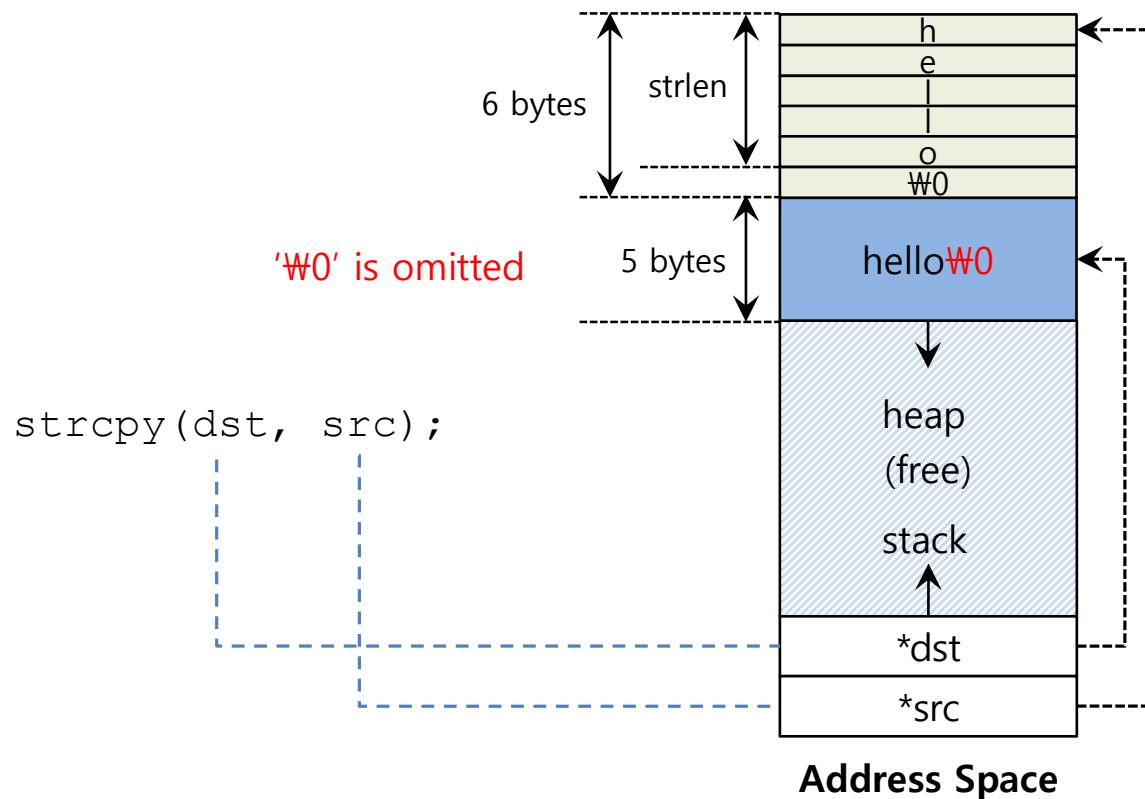
```
char *src = "hello"; //character string constant  
char *dst (char *)malloc(strlen(src) + 1 ); // allocated  
strcpy(dst, src); //work properly
```



Not Allocating Enough Memory

- Incorrect code, but work properly

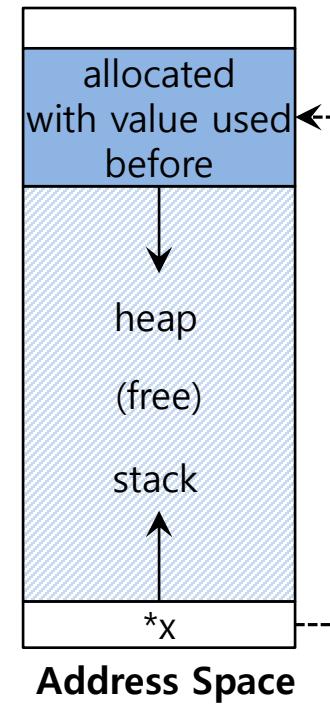
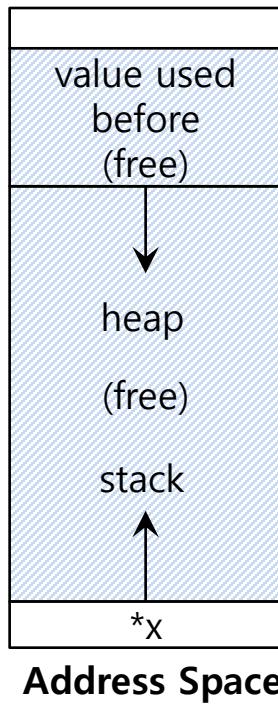
```
char *src = "hello"; //character string constant  
char *dst (char *)malloc(strlen(src)); // too small  
strcpy(dst, src); //work properly
```



Forgetting to Initialize

- Encounter an uninitialized read

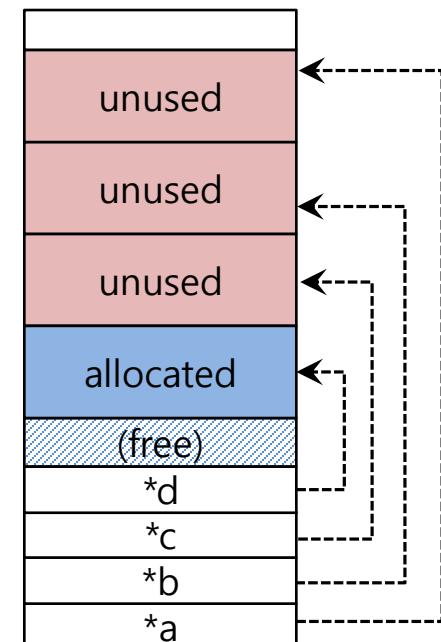
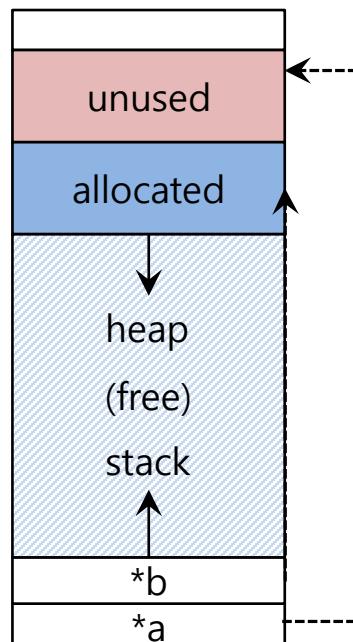
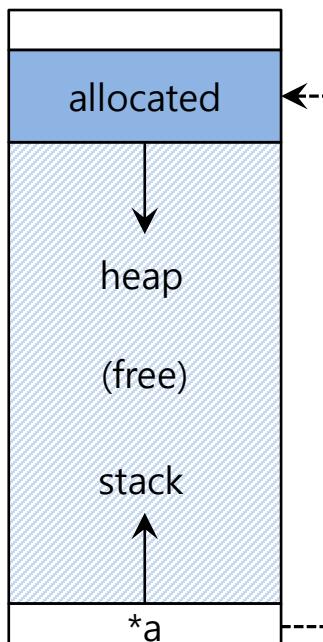
```
int *x = (int *)malloc(sizeof(int)); // allocated  
printf("*x = %d\n", *x); // uninitialized memory access
```



Memory Leak

- A program runs out of memory and eventually dies.

unused : unused, but not freed



Address Space

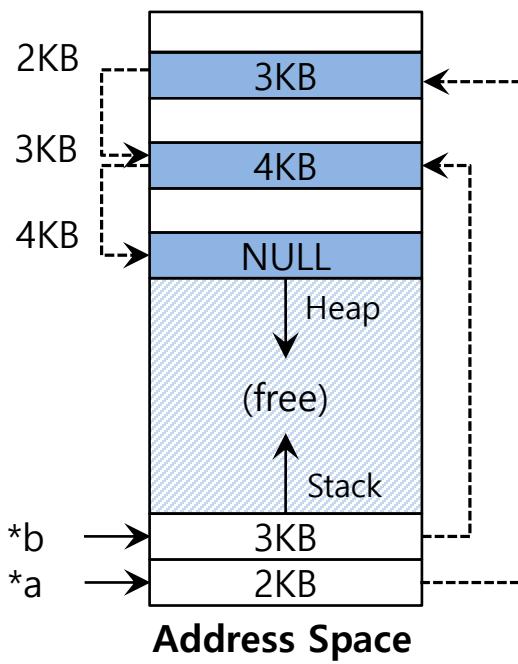
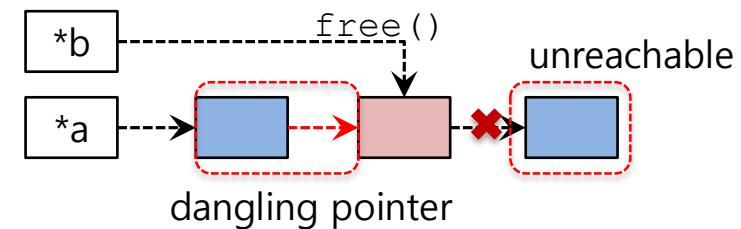
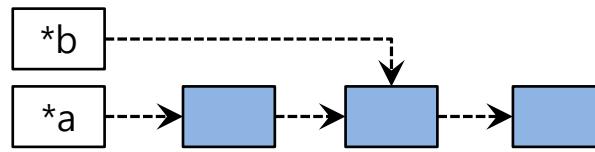
Address Space

Address Space

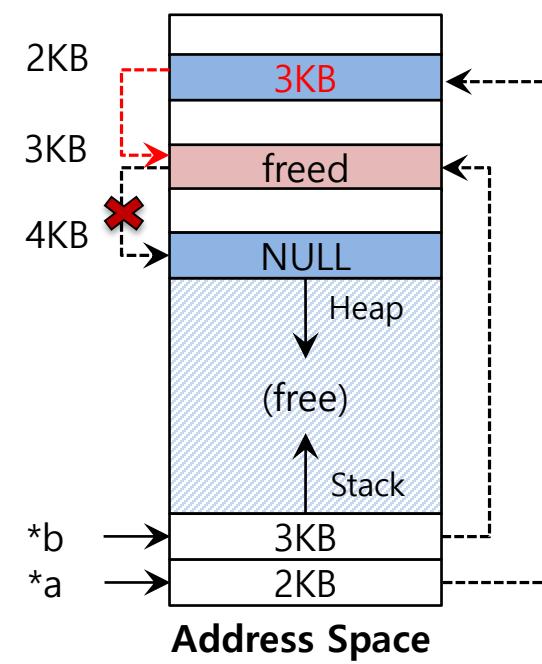
run out of memory

Dangling Pointer

- Freeing memory before it is finished using
 - A program accesses to memory with an invalid pointer



`free(b)`



Other Memory APIs: calloc()

```
#include <stdlib.h>

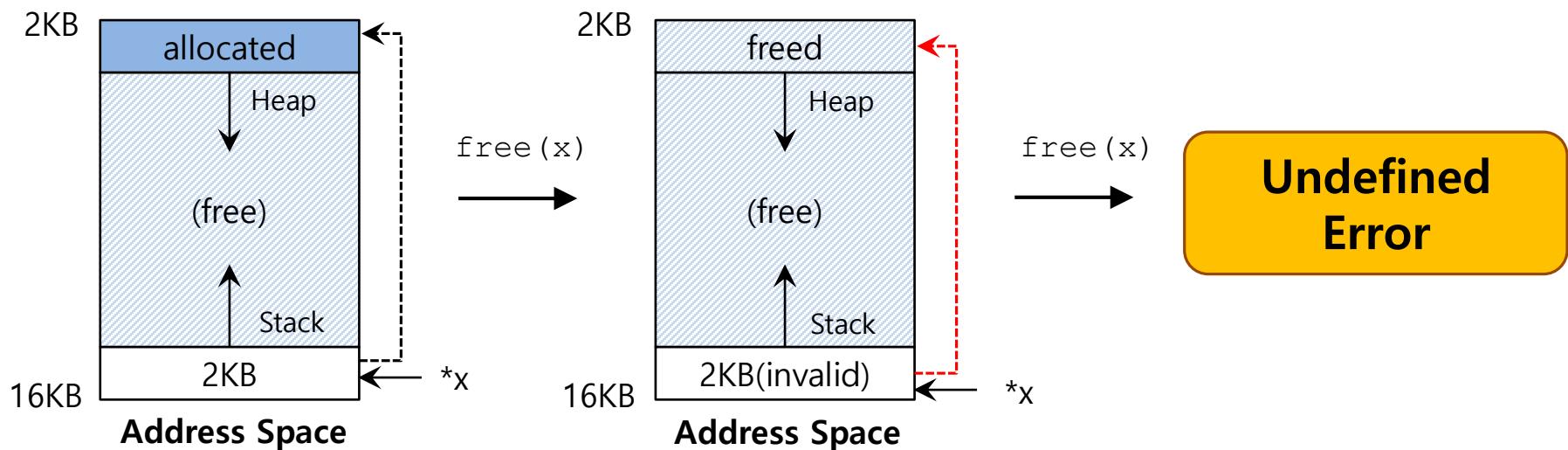
void *calloc(size_t num, size_t size)
```

- ▣ Allocate memory on the heap and zeroes it before returning.
 - ◆ Argument
 - size_t num : number of blocks to allocate
 - size_t size : size of each block(in bytes)
 - ◆ Return
 - Success : a void type pointer to the memory block allocated by calloc
 - Fail : a null pointer

Double Free

- Free memory that was freed already.

```
int *x = (int *)malloc(sizeof(int)); // allocated  
free(x); // free memory  
free(x); // free repeatedly
```



Other Memory APIs: realloc()

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- ▣ Change the size of memory block.

- ◆ A pointer returned by `realloc` may be either the same as `ptr` or a new.
- ◆ Argument
 - `void *ptr`: Pointer to memory block allocated with `malloc`, `calloc` or `realloc`
 - `size_t size`: New size for the memory block(in bytes)
- ◆ Return
 - Success: Void type pointer to the memory block
 - Fail : Null pointer

System Calls

```
#include <unistd.h>

int brk(void *addr)
void *sbrk(intptr_t increment);
```

- ▣ malloc library call use **brk** system call.
 - ◆ brk is called to expand the program's *break*.
 - *break*: The location of **the end of the heap** in address space
 - ◆ sbrk is an additional call similar with brk.
 - ◆ Programmers **should never directly call** either brk or sbrk.

System Calls(Cont.)

```
#include <sys/mman.h>

void *mmap(void *ptr, size_t length, int port, int flags,
int fd, off_t offset)
```

- ◆ mmap system call can create **an anonymous** memory region.

15. Address Translation

Memory Virtualizing with Efficiency and Control

- Memory virtualizing takes a similar strategy known as **limited direct execution(LDE)** for efficiency and control.
- In memory virtualizing, efficiency and control are attained by **hardware support**.
 - ◆ e.g., registers, TLB(Translation Look-aside Buffer)s, page-table

Address Translation

- ▣ Hardware transforms a **virtual address** to a **physical address**.
 - ◆ The desired information is actually stored in a physical address.
- ▣ The OS must get involved at key points to set up the hardware.
 - ◆ The OS must manage memory to judiciously intervene.

Example: Address Translation

- ❑ C - Language code

```
void func()
    int x;
    ...
    x = x + 3; // this is the line of code we are interested in
```

- ◆ **Load** a value from memory
- ◆ **Increment** it by three
- ◆ **Store** the value back into memory

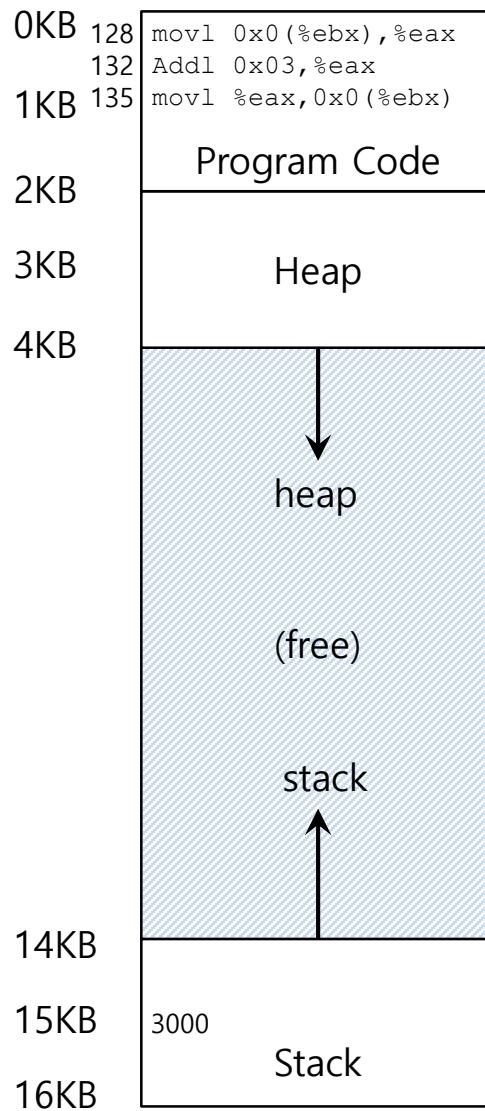
Example: Address Translation(Cont.)

▣ Assembly

```
128 : movl 0x0(%ebx), %eax          ; load 0+ebx into eax  
132 : addl $0x03, %eax             ; add 3 to eax register  
135 : movl %eax, 0x0(%ebx)         ; store eax back to mem
```

- ◆ **Load** the value at that address into `eax` register.
- ◆ **Add** 3 to `eax` register.
- ◆ **Store** the value in `eax` back into memory.

Example: Address Translation(Cont.)

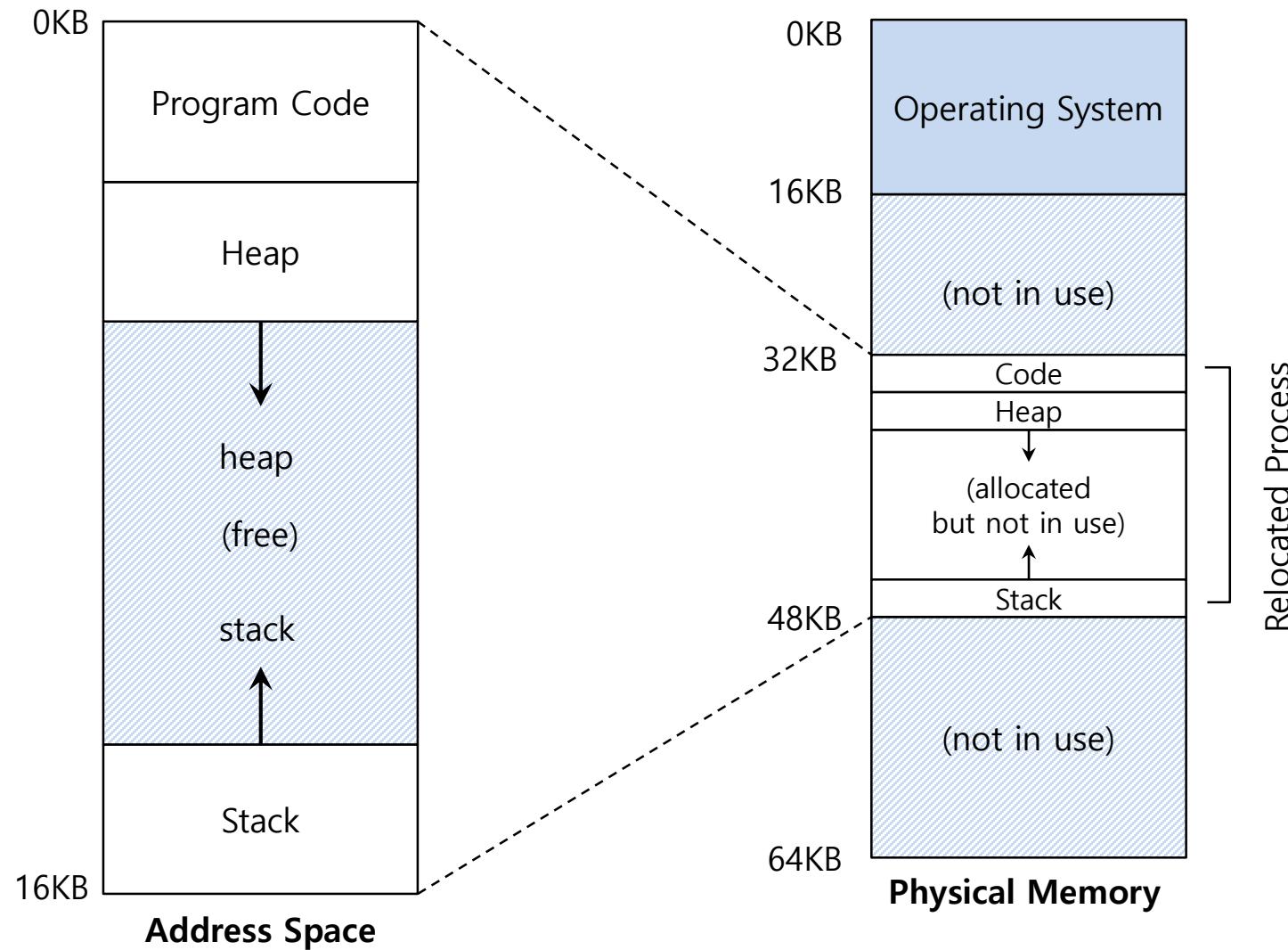


- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

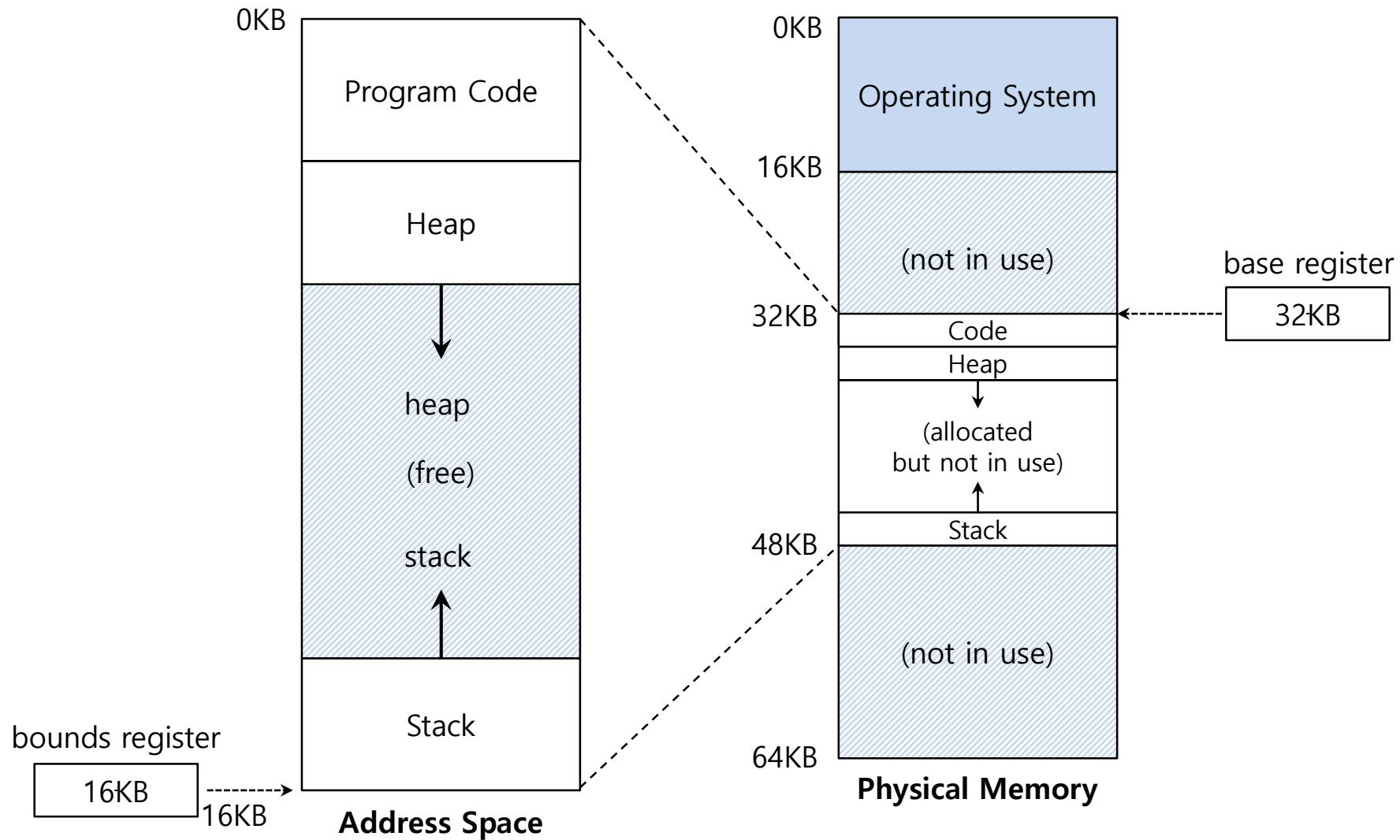
Relocation Address Space

- ▣ The OS wants to place the process **somewhere else** in physical memory, not at address 0.
 - ◆ The address space start at address 0.

A Single Relocated Process



Base and Bounds Register



Dynamic(Hardware base) Relocation

- When a program starts running, the OS decides **where** in physical memory a process should be **loaded**.
 - Set the **base** register a value.

$$\text{physical address} = \text{virtual address} + \text{base}$$

- Every virtual address must **not be greater than bound** and **negative**.

$$0 \leq \text{virtual address} < \text{bounds}$$

Relocation and Address Translation

```
128 : movl 0x0(%ebx), %eax
```

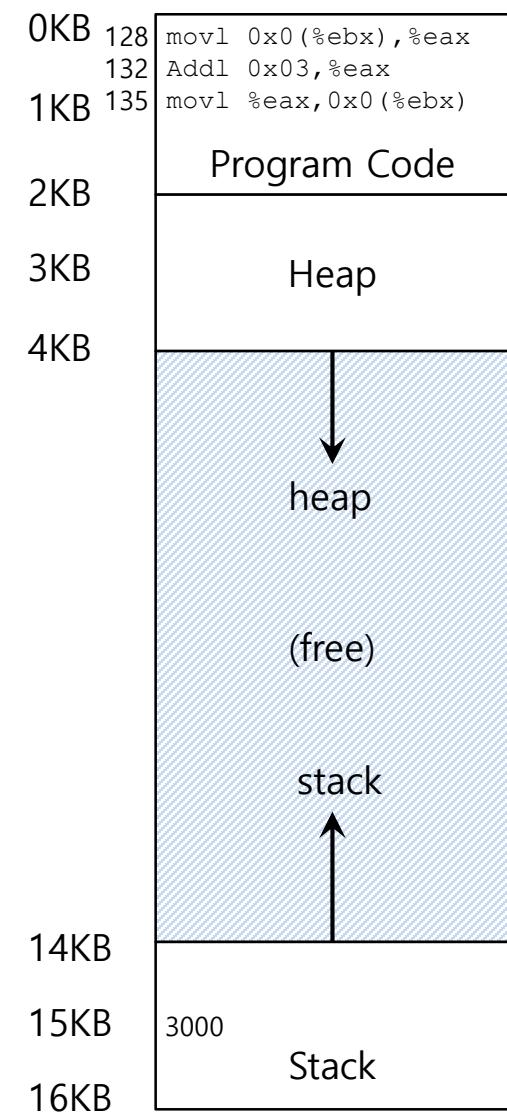
- ◆ Fetch instruction at address 128

$$32896 = 128 + 32KB(base)$$

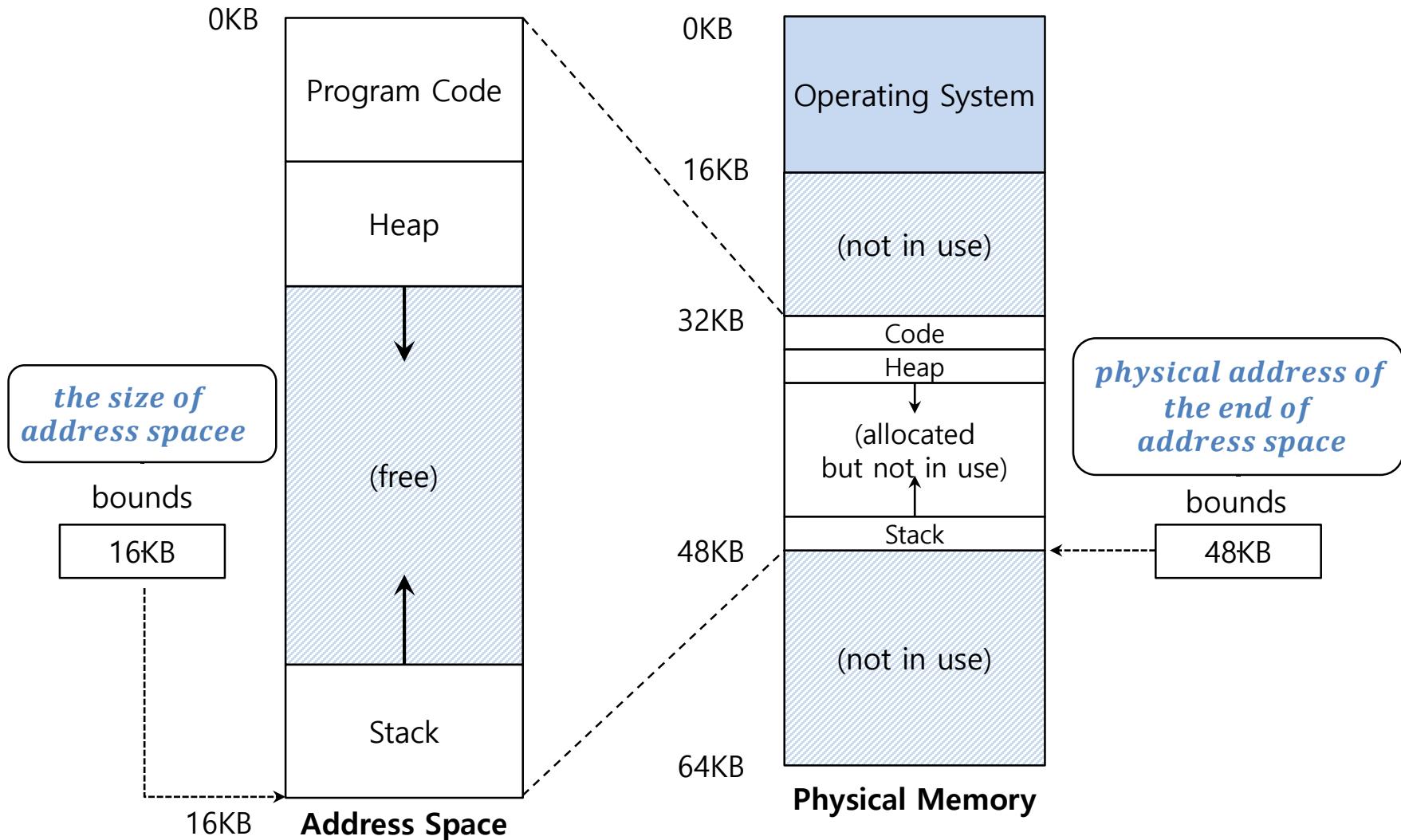
- ◆ Execute this instruction

- Load from address 15KB

$$47KB = 15KB + 32KB(base)$$



Two ways of Bounds Register

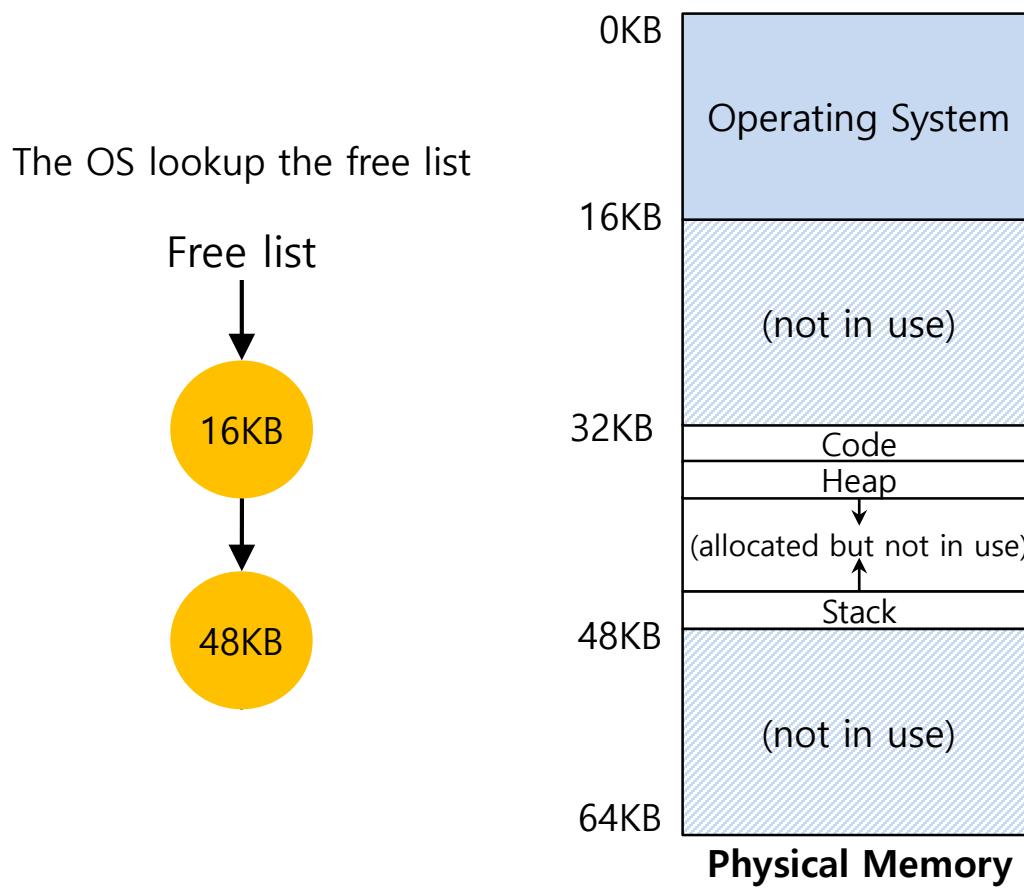


OS Issues for Memory Virtualizing

- ▣ The OS must **take action** to implement **base-and-bounds** approach.
- ▣ Three critical junctures:
 - ◆ When a process **starts running**:
 - Finding space for address space in physical memory
 - ◆ When a process is **terminated**:
 - Reclaiming the memory for use
 - ◆ When context **switch occurs**:
 - Saving and storing the base-and-bounds pair

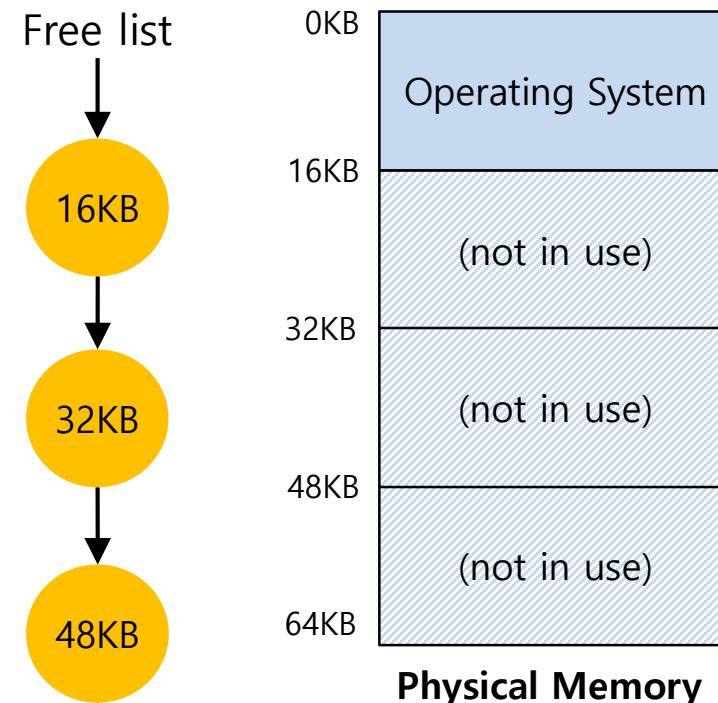
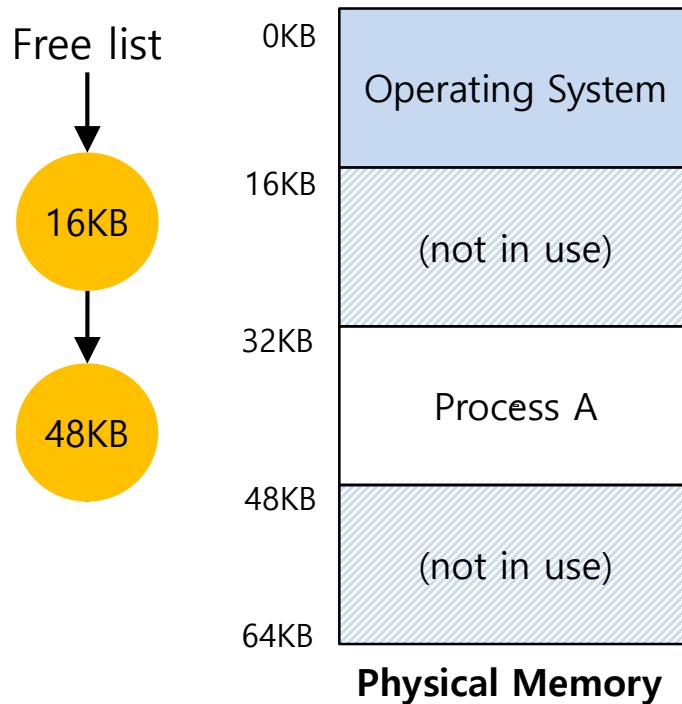
OS Issues: When a Process Starts Running

- The OS must **find a room** for a new address space.
 - ◆ free list : A list of the range of the physical memory which are not in use.



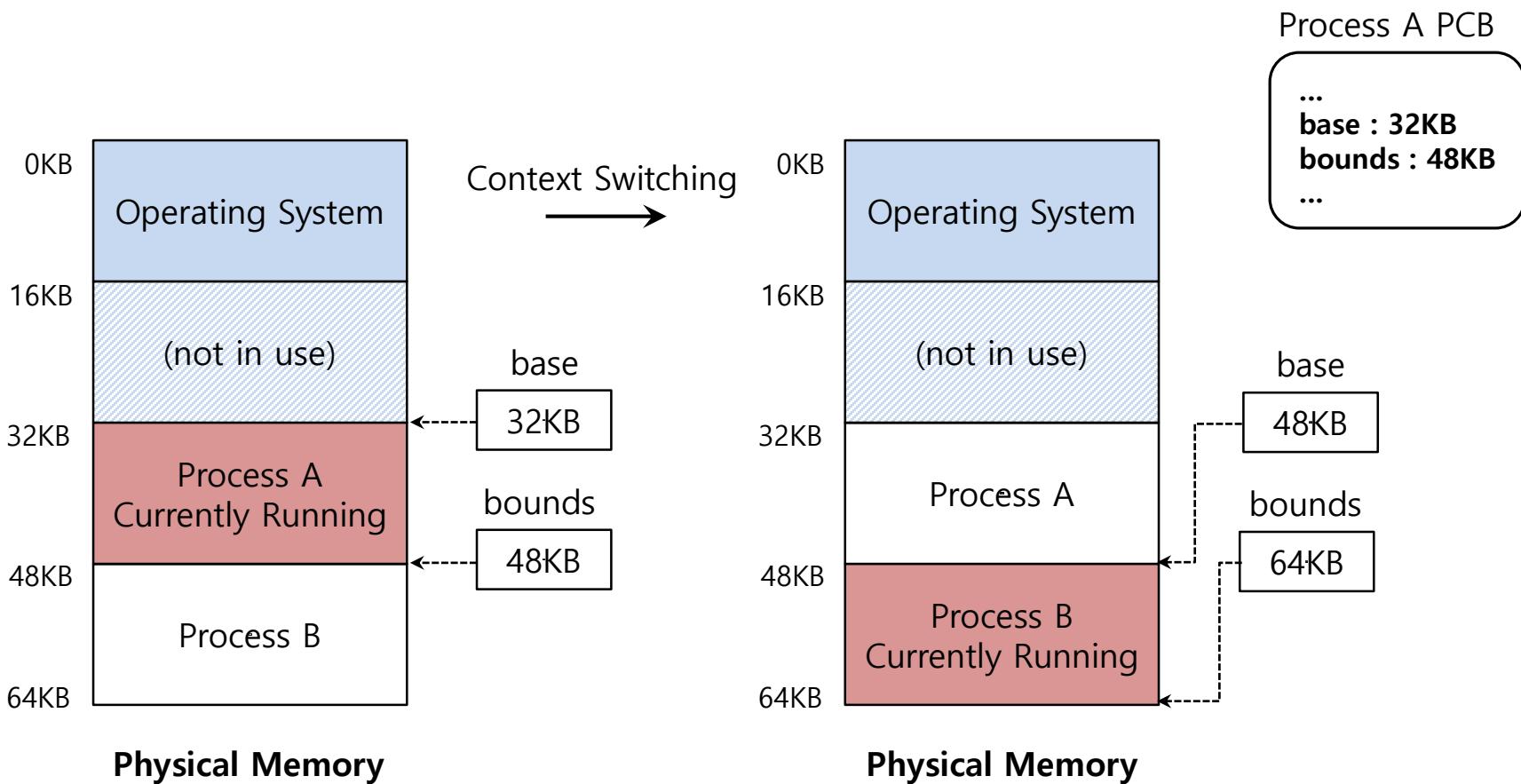
OS Issues: When a Process Is Terminated

- The OS must put the memory back on the free list.



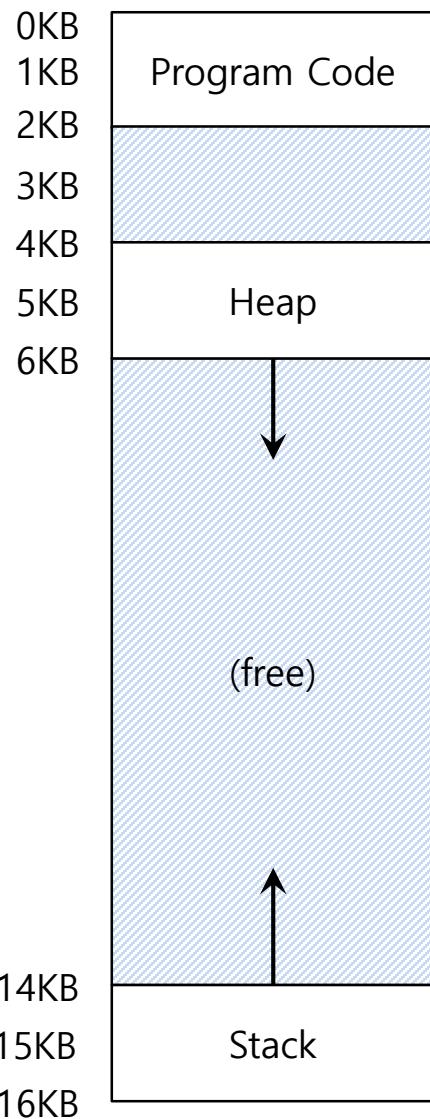
OS Issues: When Context Switch Occurs

- The OS must **save and restore** the base-and-bounds pair.
 - In **process structure** or **process control block(PCB)**



16. Segmentation

Inefficiency of the Base and Bound Approach

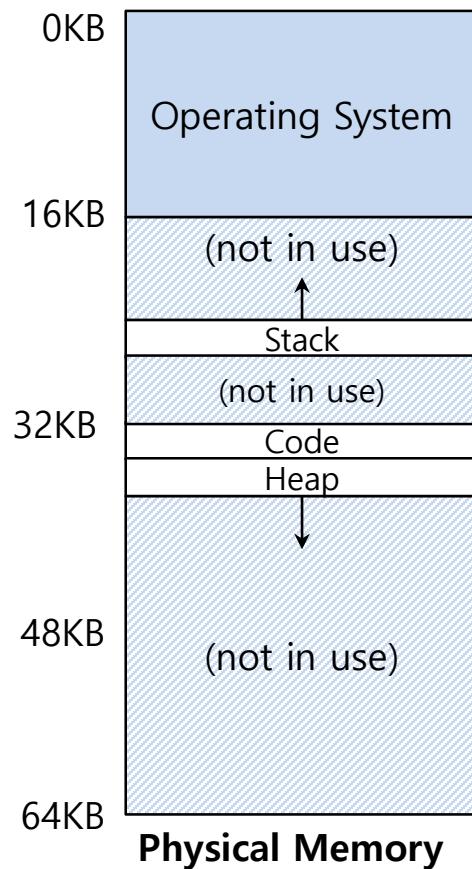


- **Big chunk of “free” space**
- “free” space **takes up** physical memory.
- Hard to run when an address space **does not fit** into physical memory

Segmentation

- ▣ Segment is just **a contiguous portion** of the address space of a particular length.
 - ◆ Logically-different segment: code, stack, heap
- ▣ Each segment can be **placed** in **different part of physical memory**.
 - ◆ **Base** and **bounds** exist **per each segment**.

Placing Segment In Physical Memory

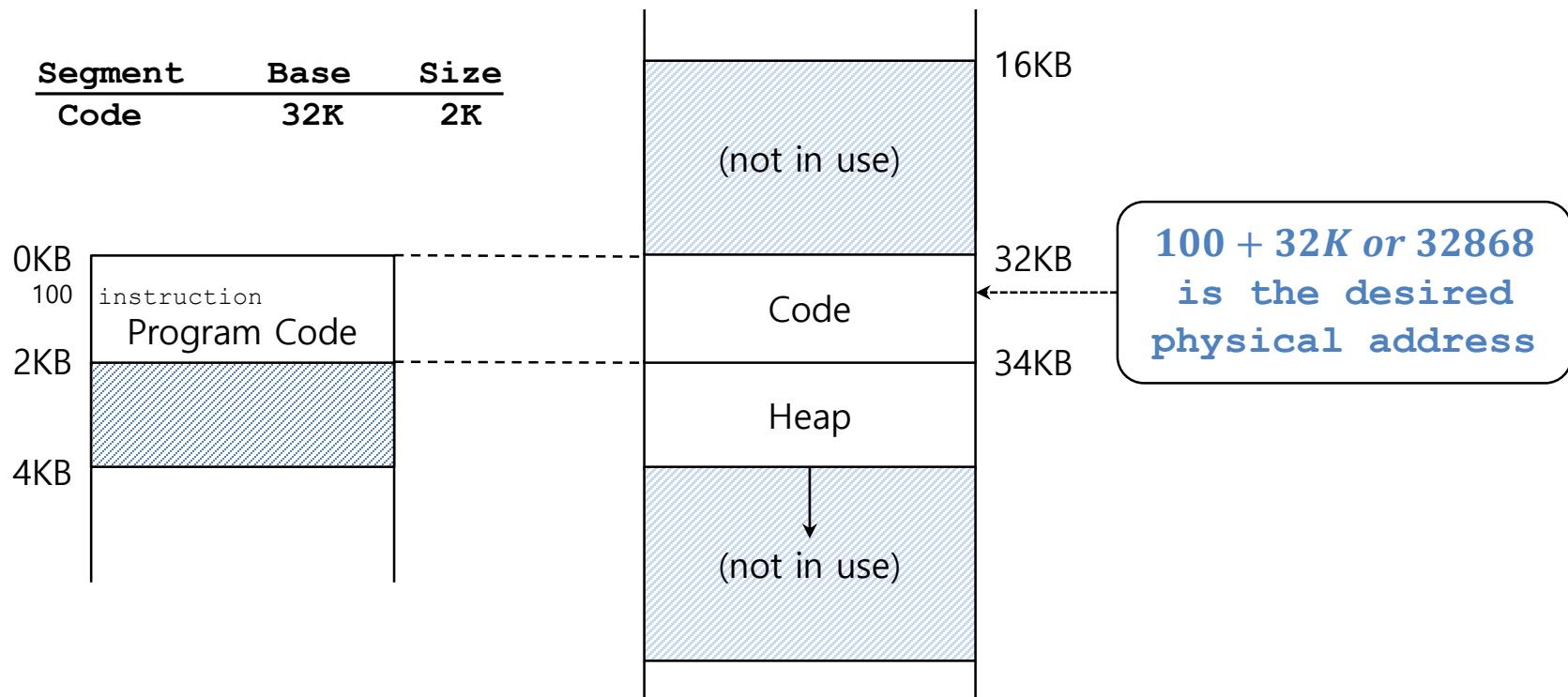


Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Address Translation on Segmentation

$$\text{physical address} = \text{offset} + \text{base}$$

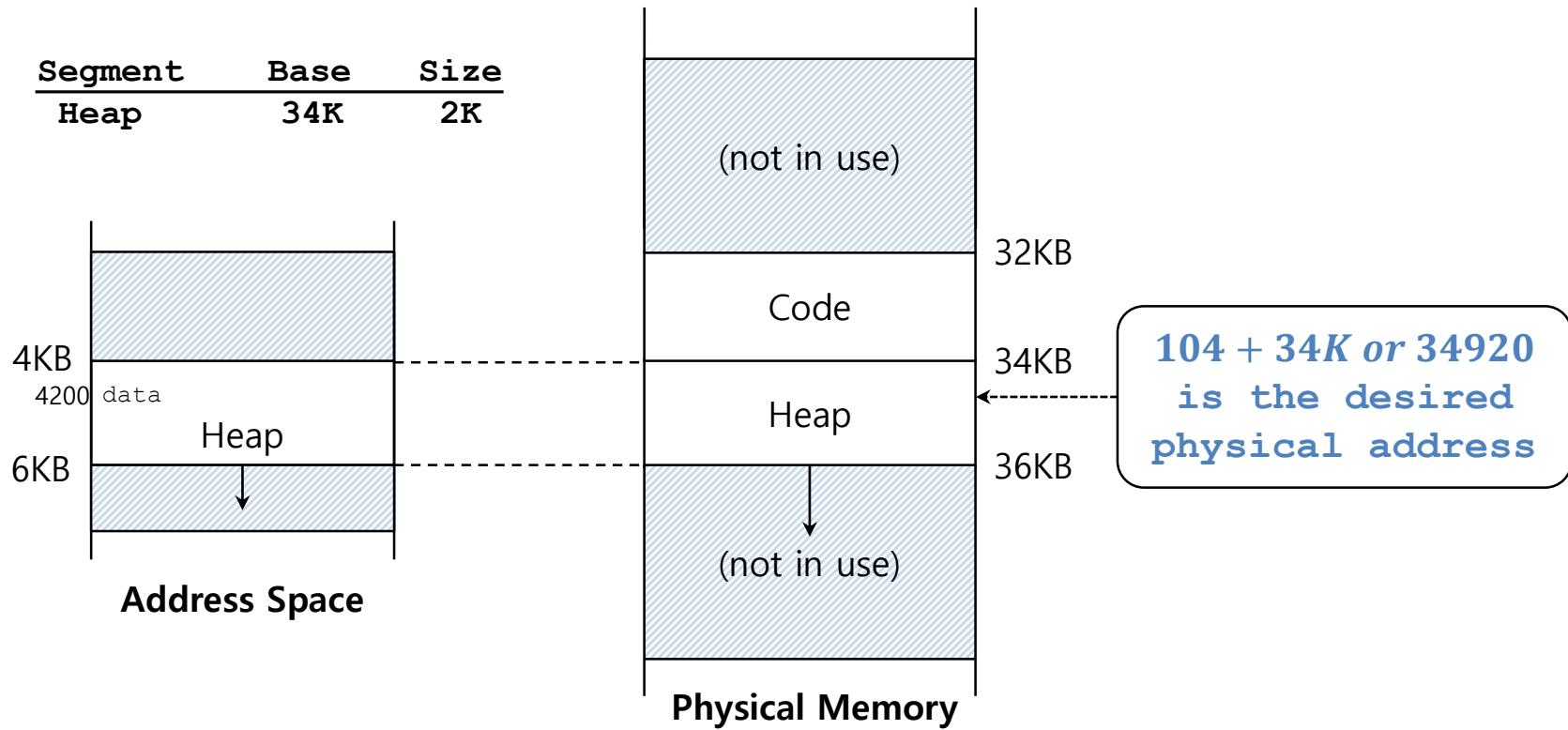
- The offset of virtual address 100 is 100.
 - The code segment **starts at virtual address 0** in address space.



Address Translation on Segmentation(Cont.)

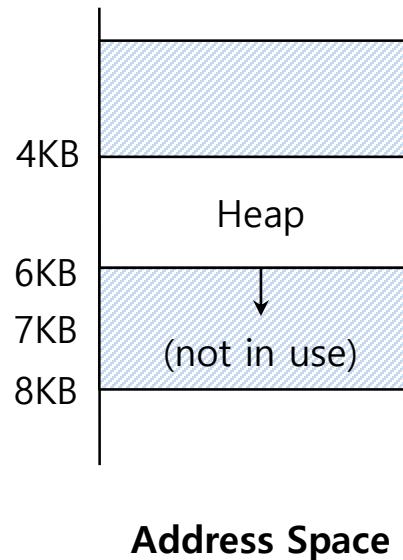
Virtual address + base is not the correct physical address.

- The offset of virtual address 4200 is 104.
 - ◆ The heap segment **starts at virtual address 4096** in address space.



Segmentation Fault or Violation

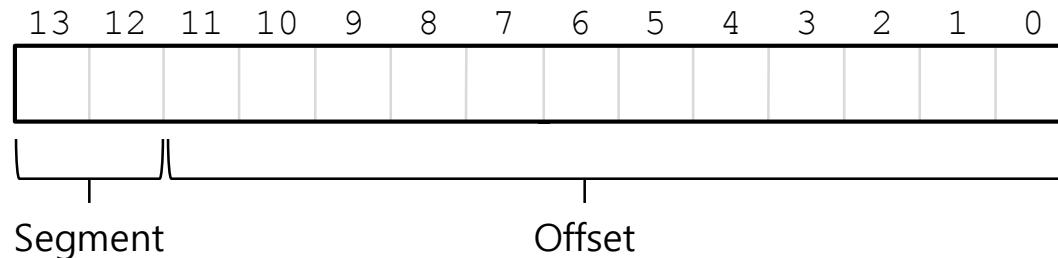
- If an **illegal address** such as 7KB which is beyond the end of heap is referenced, the OS occurs **segmentation fault**.
 - ◆ The hardware detects that address is **out of bounds**.



Referring to Segment

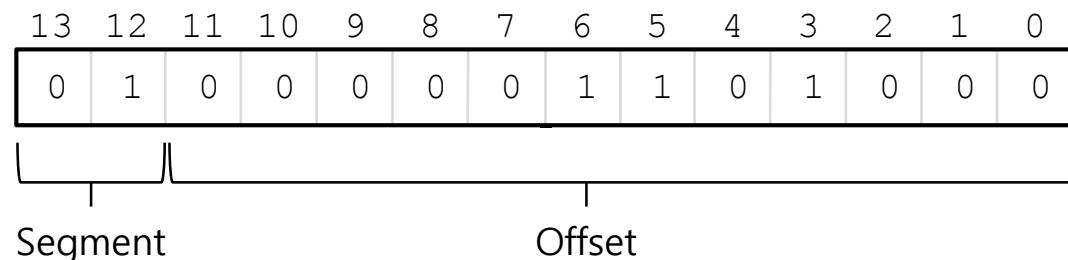
Explicit approach

- Chop up the address space into segments based on the **top few bits** of virtual address.



Example: virtual address 4200 (01000001101000)

Segment	bits
Code	00
Heap	01
Stack	10
-	11



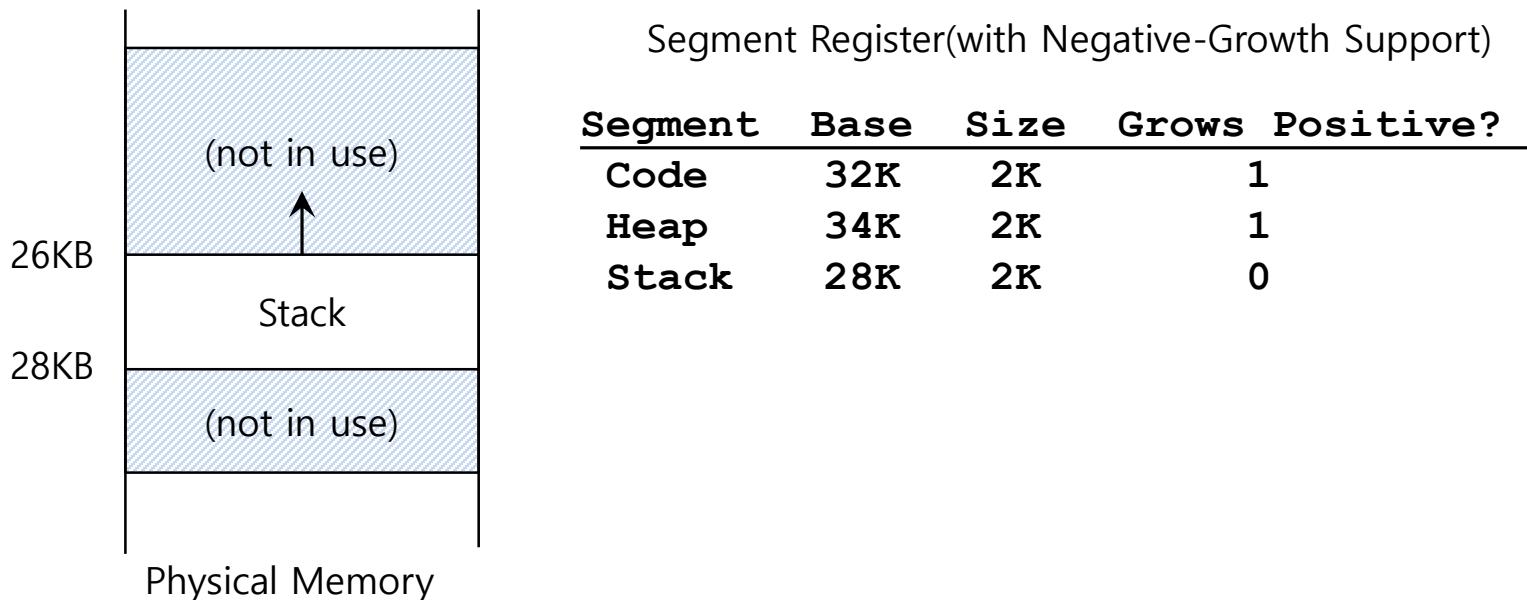
Referring to Segment(Cont.)

```
1 // get top 2 bits of 14-bit VA
2 Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3 // now get offset
4 Offset = VirtualAddress & OFFSET_MASK
5 if (Offset >= Bounds[Segment])
6     RaiseException(PROTECTION_FAULT)
7 else
8     PhysAddr = Base[Segment] + Offset
9     Register = AccessMemory(PhysAddr)
```

- ◆ SEG_MASK = 0x3000 (1100000000000000)
- ◆ SEG_SHIFT = 12
- ◆ OFFSET_MASK = 0xFFFF (00111111111111)

Referring to Stack Segment

- Stack grows **backward**.
- **Extra hardware support** is need.
 - ◆ The hardware checks which way the segment grows.
 - ◆ 1: positive direction, 0: negative direction



Support for Sharing

- ▣ Segment can be **shared between address space**.
 - ◆ **Code sharing** is still in use in systems today.
 - ◆ by extra hardware support.
- ▣ Extra hardware support is need for form of **Protection bits**.
 - ◆ **A few more bits** per segment to indicate **permissions** of **read**, write and **execute**.

Segment Register Values(with Protection)

Segment	Base	Size	Grows	Positive?	Protection
Code	32K	2K		1	Read-Execute
Heap	34K	2K		1	Read-Write
Stack	28K	2K		0	Read-Write

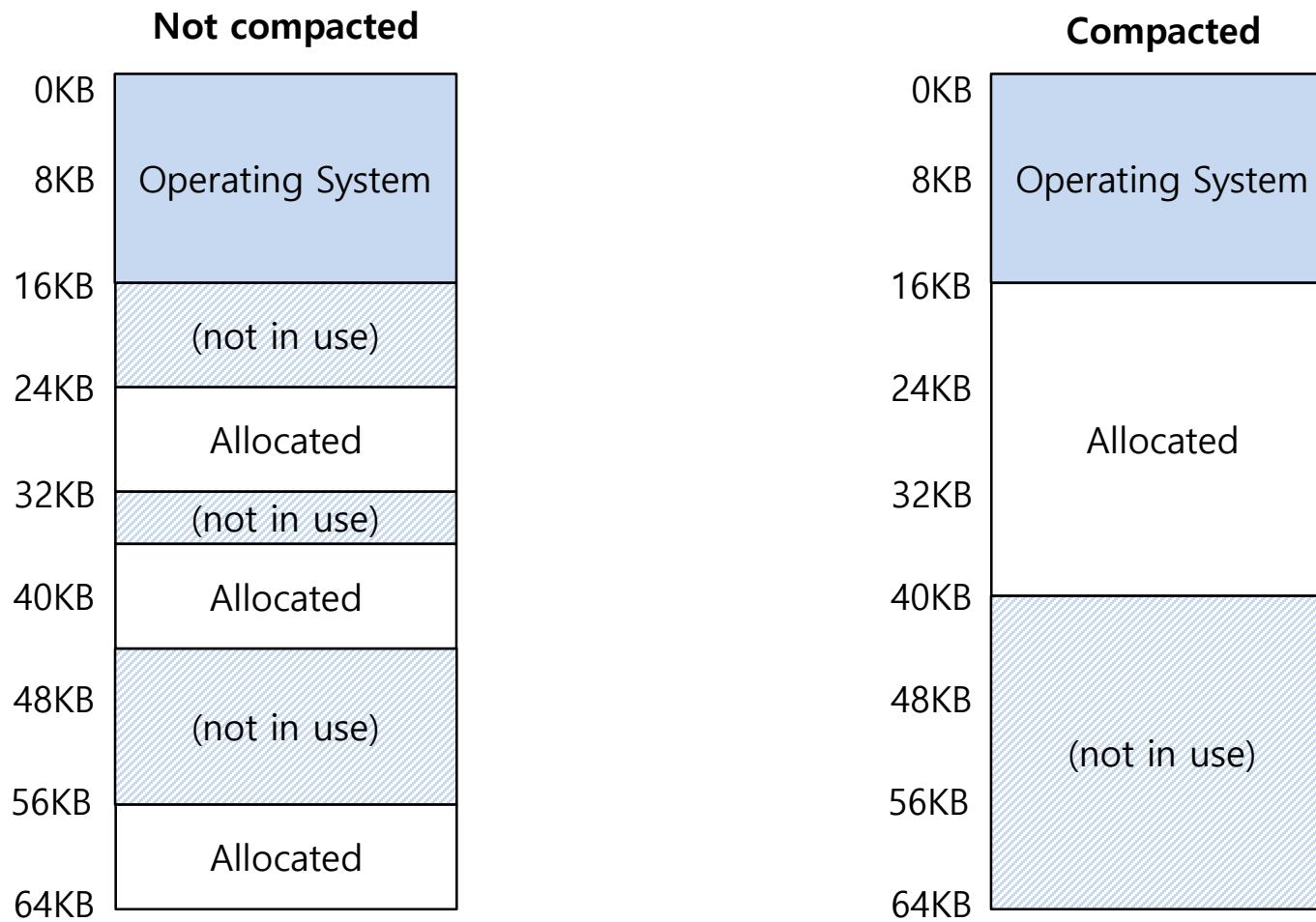
Fine-Grained and Coarse-Grained

- ▣ **Coarse-Grained** means segmentation in a small number.
 - ◆ e.g., code, heap, stack.
- ▣ **Fine-Grained** segmentation allows **more flexibility** for address space in some early system.
 - ◆ To support many segments, Hardware support with a **segment table** is required.

OS support: Fragmentation

- ▣ **External Fragmentation:** little holes of **free space** in physical memory that make difficulty to allocate new segments.
 - ◆ There is **24KB free**, but **not in one contiguous** segment.
 - ◆ The OS **cannot** satisfy the **20KB request**.
- ▣ **Compaction:** rearranging the exiting segments in physical memory.
 - ◆ Compaction is **costly**.
 - **Stop** running process.
 - **Copy** data to somewhere.
 - **Change** segment register value.

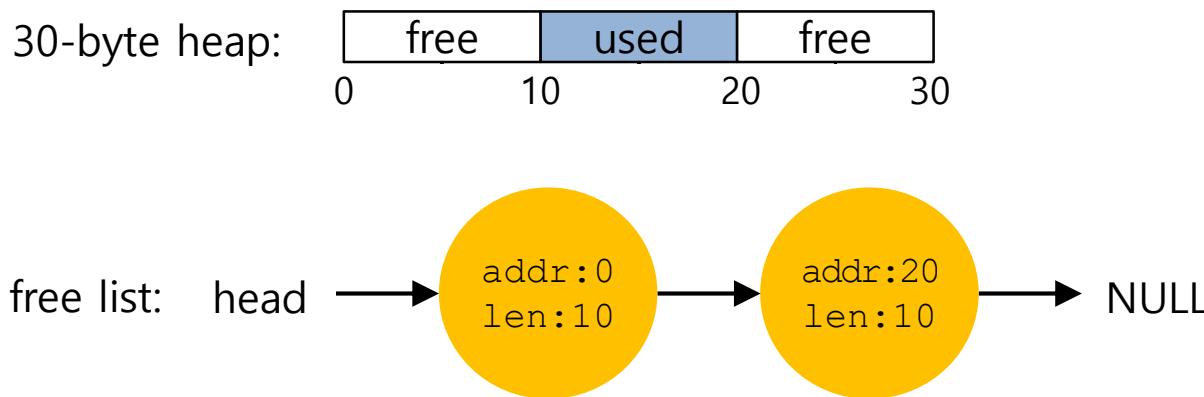
Memory Compaction



17. Free-Space Management

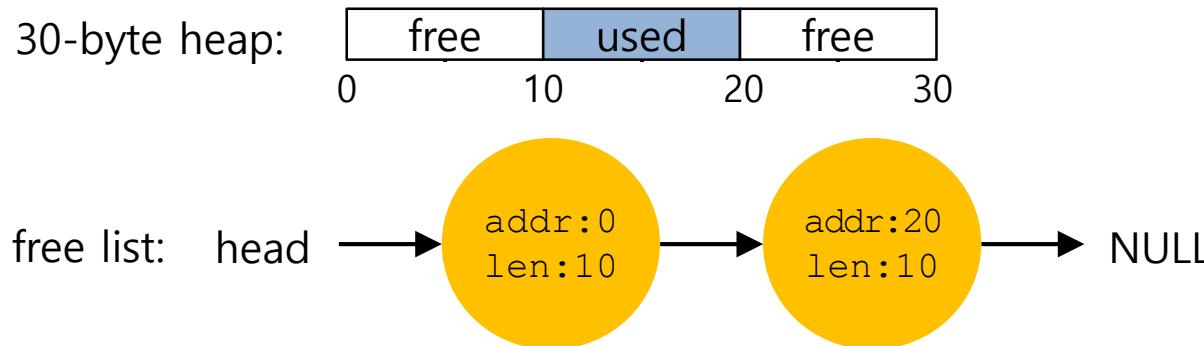
Splitting

- Finding a free chunk of memory that can satisfy the request and splitting it into two.
 - ◆ When request for memory allocation is **smaller** than the size of free chunks.

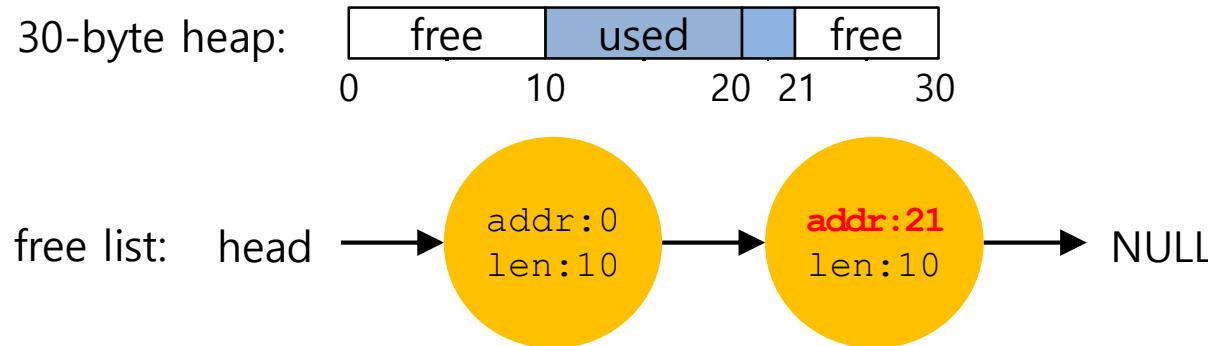


Splitting(Cont.)

- Two 10-bytes free segment with **1-byte request**

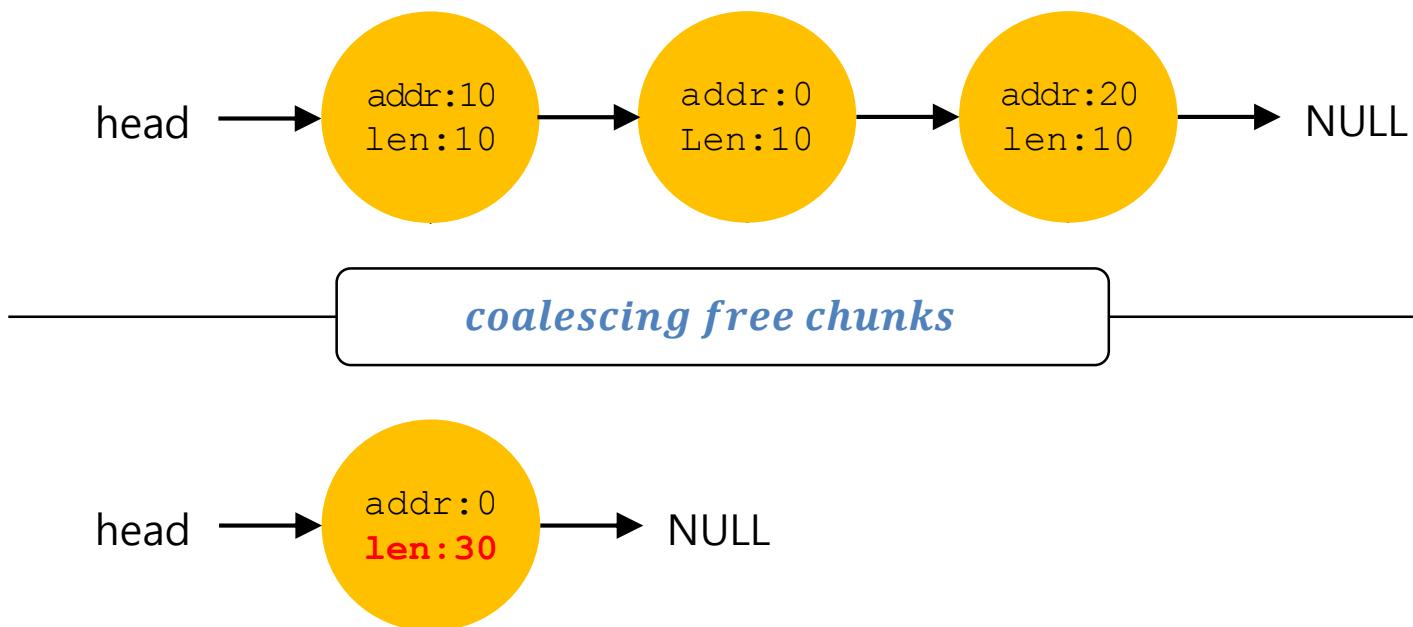


splitting 10 – byte free segment



Coalescing

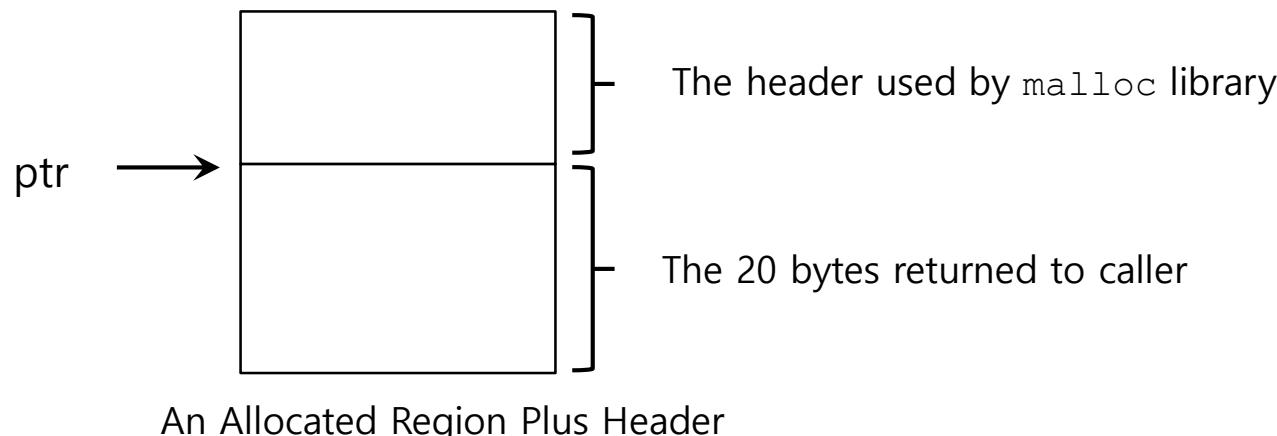
- If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk.
- Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**.



Tracking The Size of Allocated Regions

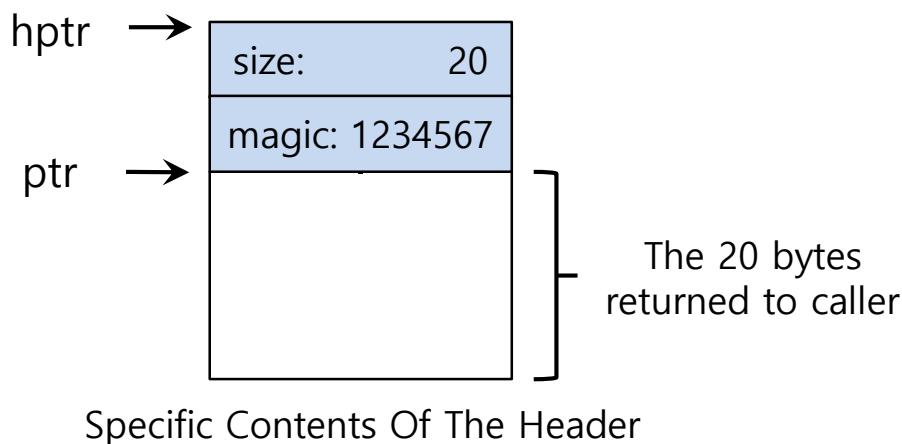
- The interface to `free(void *ptr)` does **not take a size parameter**.
 - ◆ How does the library **know the size** of memory region that will be back **into free list?**
- Most allocators store **extra information** in a **header** block.

```
ptr = malloc(20);
```



The Header of Allocated Memory Chunk

- The header minimally **contains the size** of the allocated memory region.
- The header may also contain
 - ◆ Additional pointers to speed up deallocation
 - ◆ A magic number for integrity checking



```
typedef struct __header_t {  
    int size;  
    int magic;  
} header_t;
```

A Simple Header

The Header of Allocated Memory Chunk(Cont.)

- The **size** for free region is the **size of the header plus the size of the space allocated to the user**.
 - ◆ If a user **request n bytes**, the library searches for a free chunk of **size n plus the size of the header**
- Simple pointer arithmetic to find the header pointer.

```
void free(void *ptr) {  
    header_t *hptr = (void *)ptr - sizeof(header_t);  
}
```

Embedding A Free List

- ▣ The memory-allocation library **initializes** the heap and **puts** the first element of **the free list** in the **free space**.
 - ◆ The library **can't use** `malloc()` to build a list **within itself**.

Embedding A Free List(Cont.)

- Description of a node of the list

```
typedef struct __node_t {  
    int size;  
    struct __node_t *next;  
} nodet_t;
```

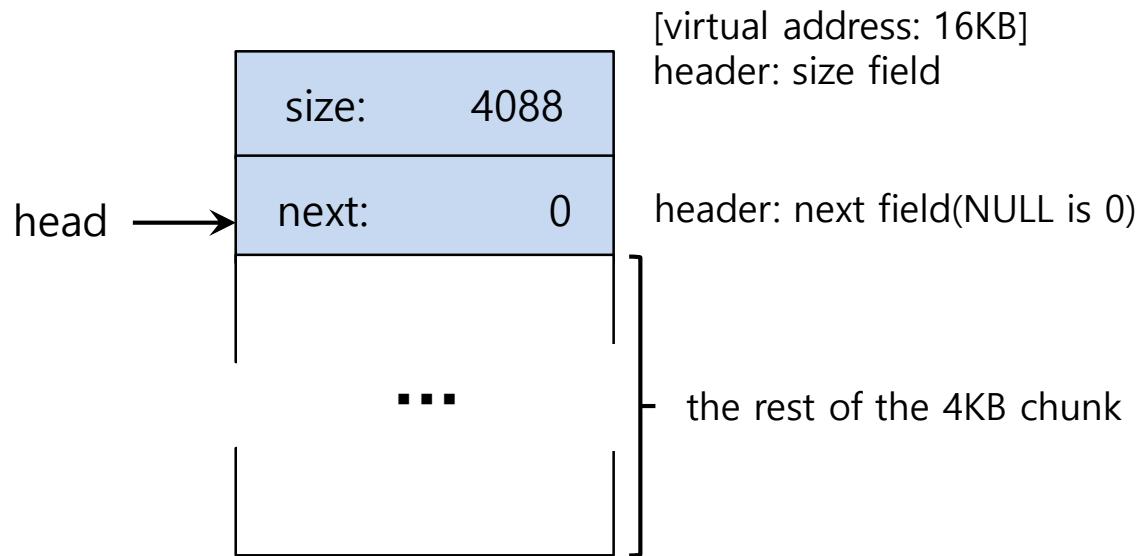
- Building heap and putting a free list

- ◆ Assume that the heap is built via `mmap()` system call.

```
// mmap() returns a pointer to a chunk of free space  
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,  
                    MAP_ANON|MAP_PRIVATE, -1, 0);  
head->size = 4096 - sizeof(node_t);  
head->next = NULL;
```

A Heap With One Free Chunk

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

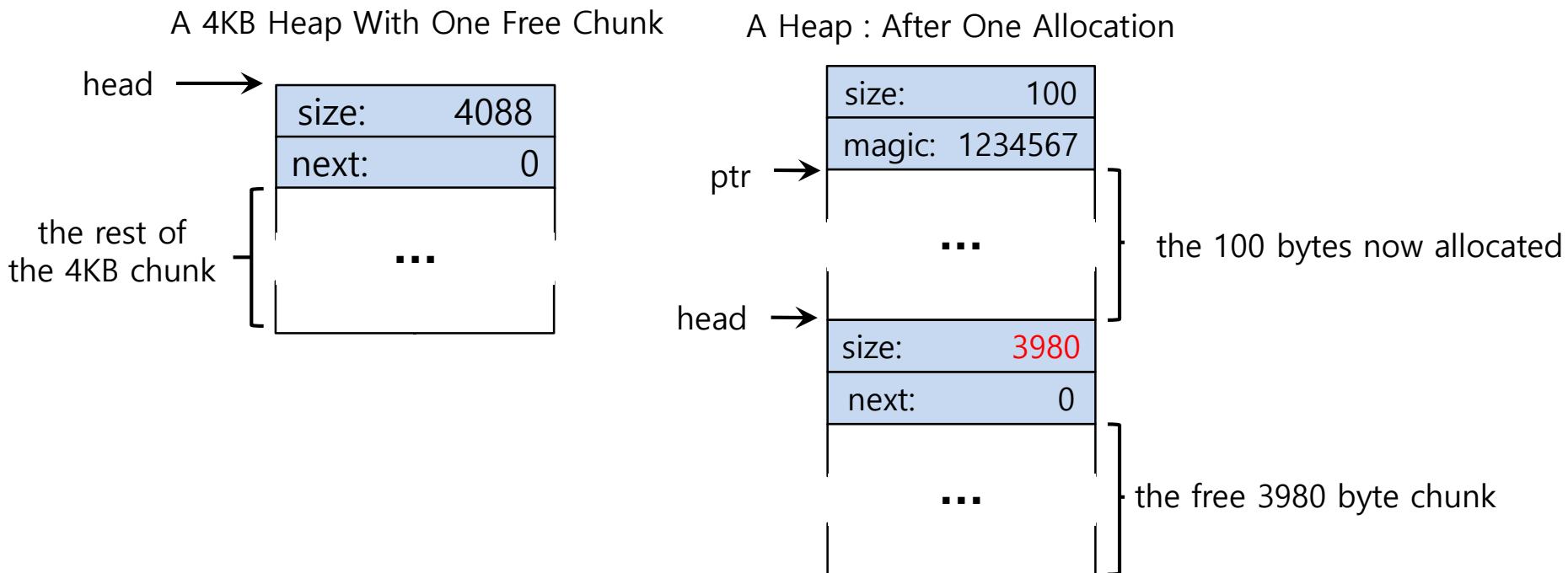


Embedding A Free List: Allocation

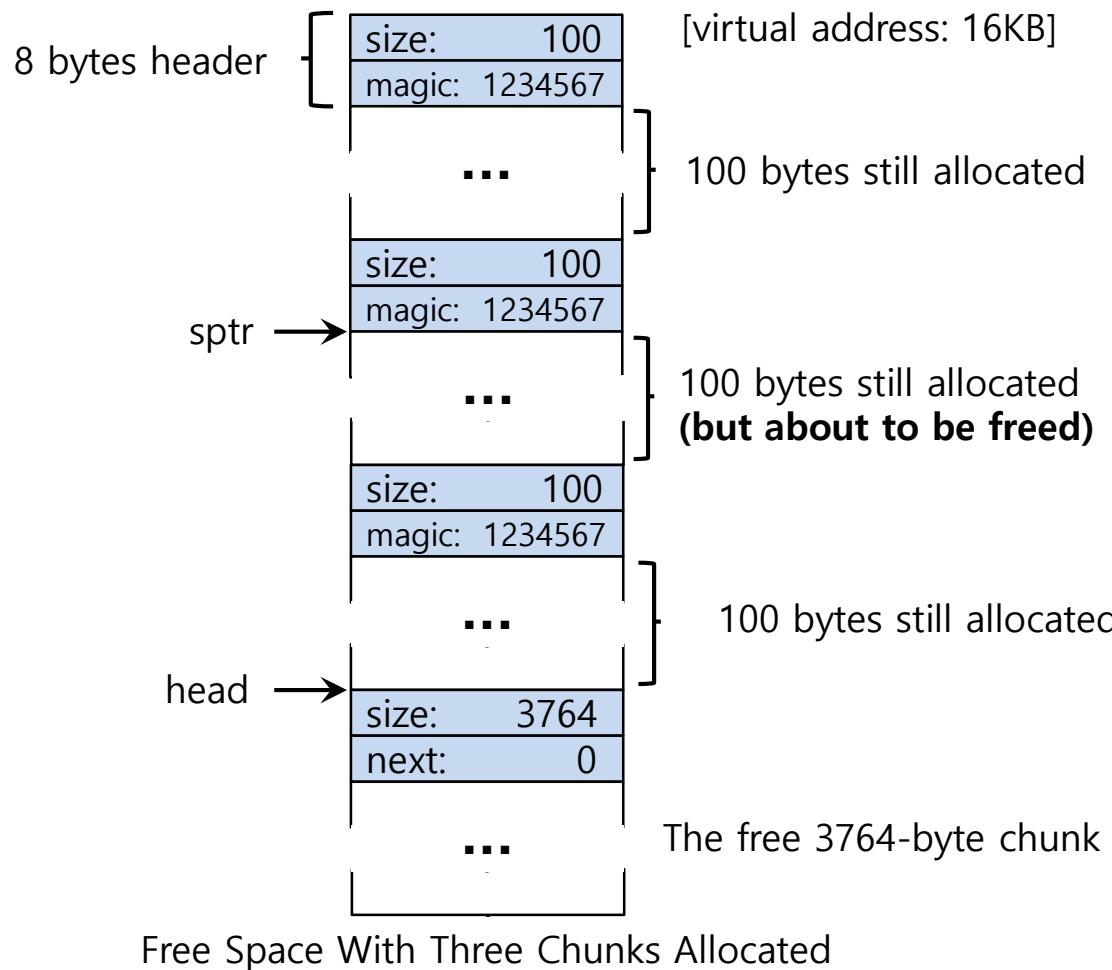
- ▣ If a chunk of memory is requested, the library **will first find** a chunk that is **large enough** to accommodate the request.
- ▣ The library will
 - ◆ **Split** the large free chunk into two.
 - **One** for the **request** and the **remaining** free chunk
 - ◆ **Shrink** the size of free chunk in the list.

Embedding A Free List: Allocation(Cont.)

- Example: a request for 100 bytes by `ptr = malloc(100)`
 - Allocating 108 bytes out of the existing one free chunk.
 - shrinking the one free chunk to 3980(4088 minus 108).



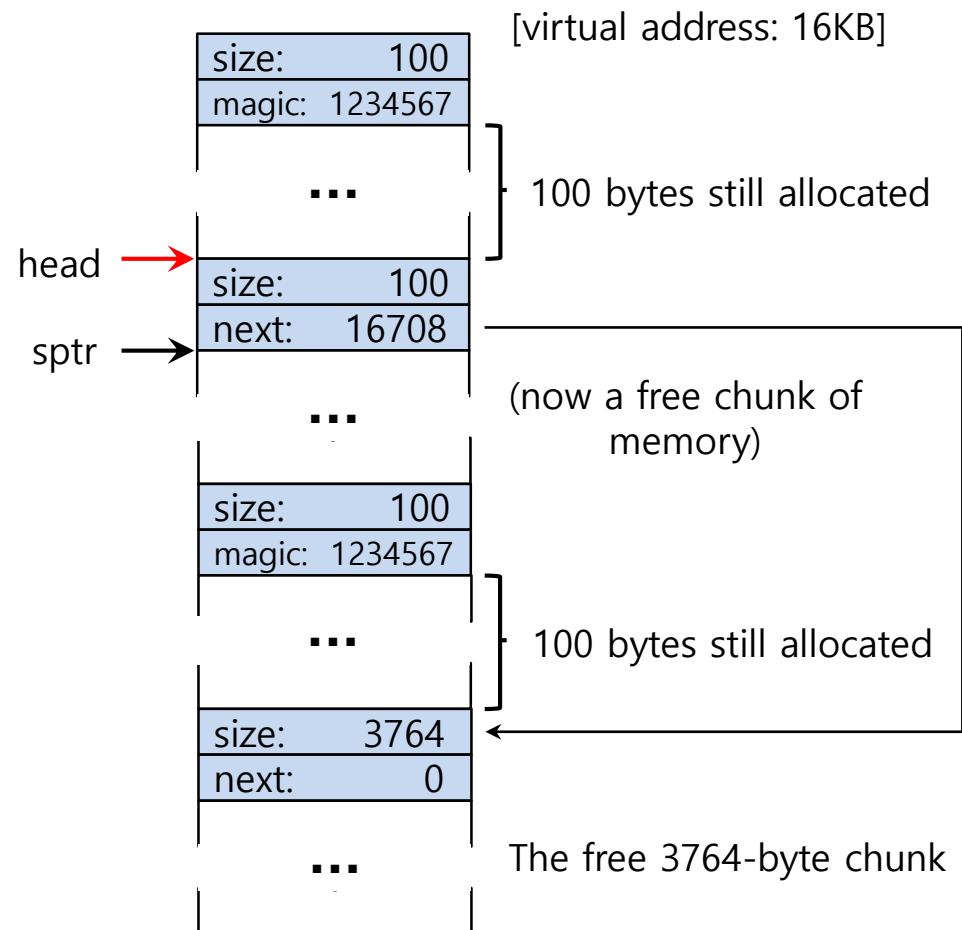
Free Space With Chunks Allocated



Free Space With `free()`

Example: `free(sptr)`

- The 100 bytes chunks is **back into** the free list.
- The free list will **start** with a **small chunk**.
 - The list header will point the small chunk

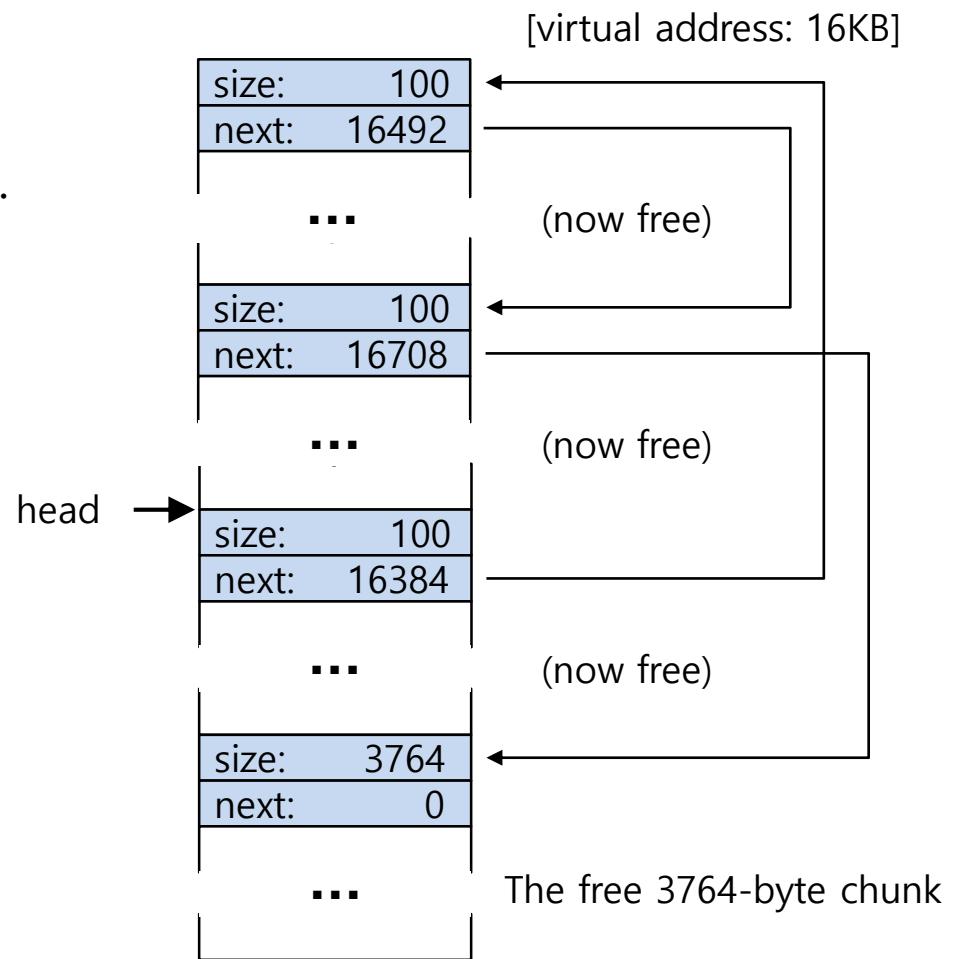


Free Space With Freed Chunks

- Let's assume that the last two in-use chunks are freed.

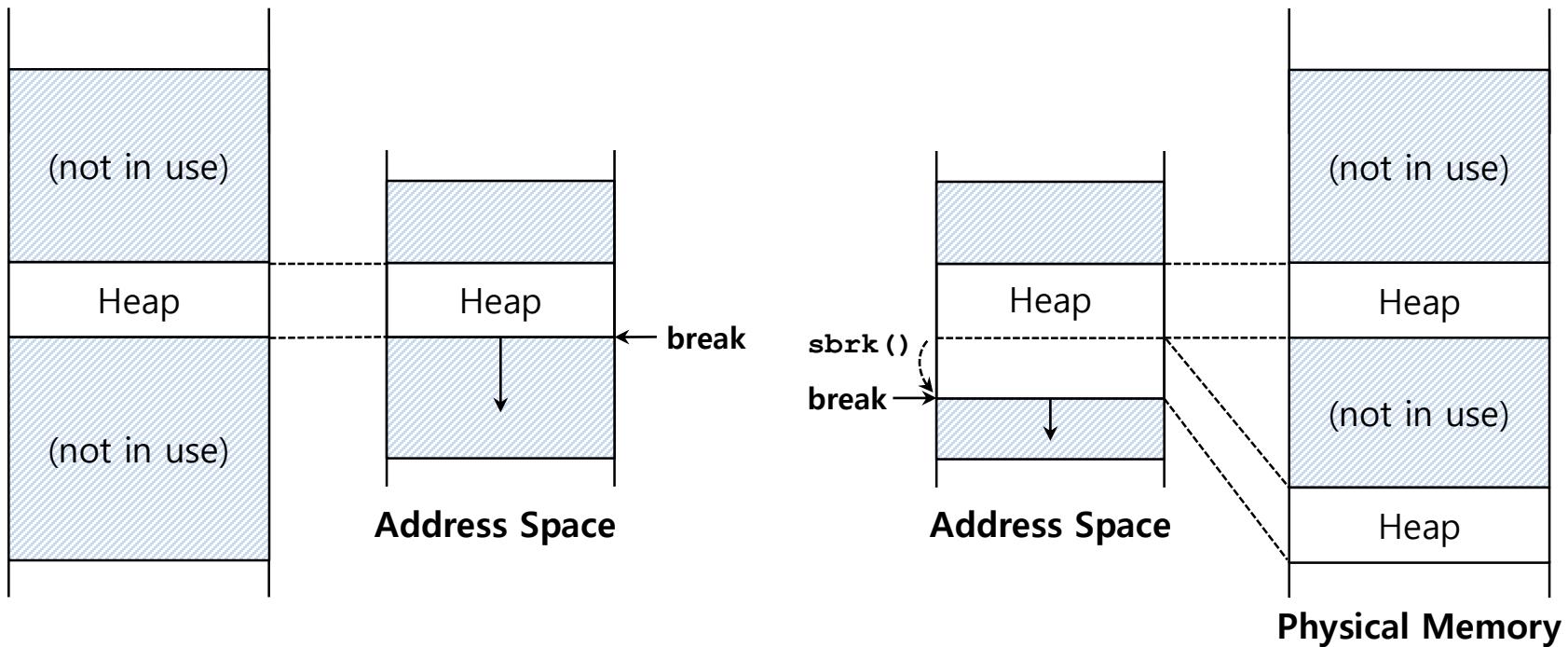
- External Fragmentation** occurs.

- Coalescing is needed in the list.



Growing The Heap

- Most allocators **start** with a **small-sized heap** and then **request more** memory from the OS when they run out.
 - e.g., `sbrk()`, `brk()` in most UNIX systems.



Managing Free Space: Basic Strategies

- Best Fit:

- ◆ Finding free chunks that are **big or bigger than the request**
- ◆ Returning the **one of smallest** in the chunks **in the group** of candidates

- Worst Fit:

- ◆ Finding the **largest free chunks** and allocation the amount of the request
- ◆ **Keeping the remaining chunk** on the free list.

Managing Free Space: Basic Strategies(Cont.)

- ▣ First Fit:

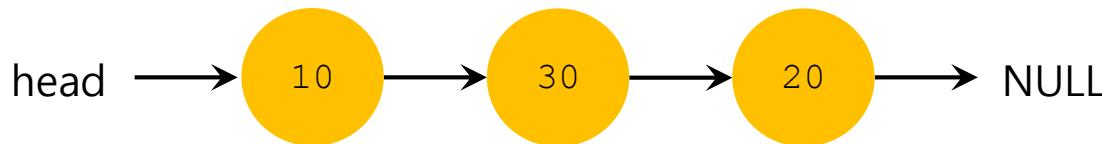
- ◆ Finding the **first chunk** that is **big enough** for the request
- ◆ Returning the requested amount and remaining the rest of the chunk.

- ▣ Next Fit:

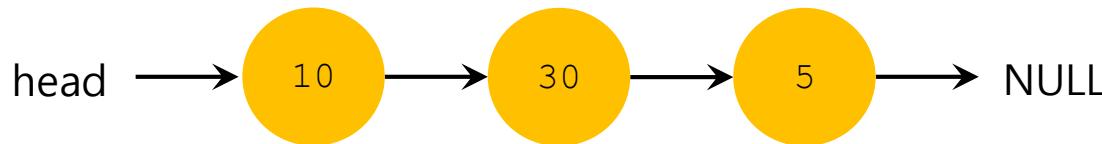
- ◆ Finding the first chunk that is big enough for the request.
- ◆ Searching at **where one was looking** at instead of the beginning of the list.

Examples of Basic Strategies

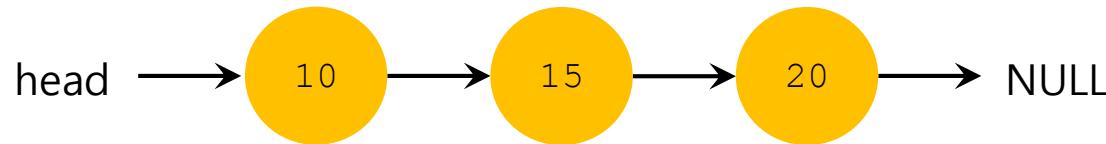
- Allocation Request Size 15



- Result of Best-fit



- Result of Worst-fit



Other Approaches: Segregated List

- ▣ Segregated List:

- ◆ Keeping free chunks in different size in a separate list for the size of popular request.
- ◆ New Complication:
 - **How much** memory should dedicate to **the pool of memory** that serves **specialized requests** of a given size?
- ◆ **Slab allocator** handles this issue.

Other Approaches: Segregated List(Cont.)

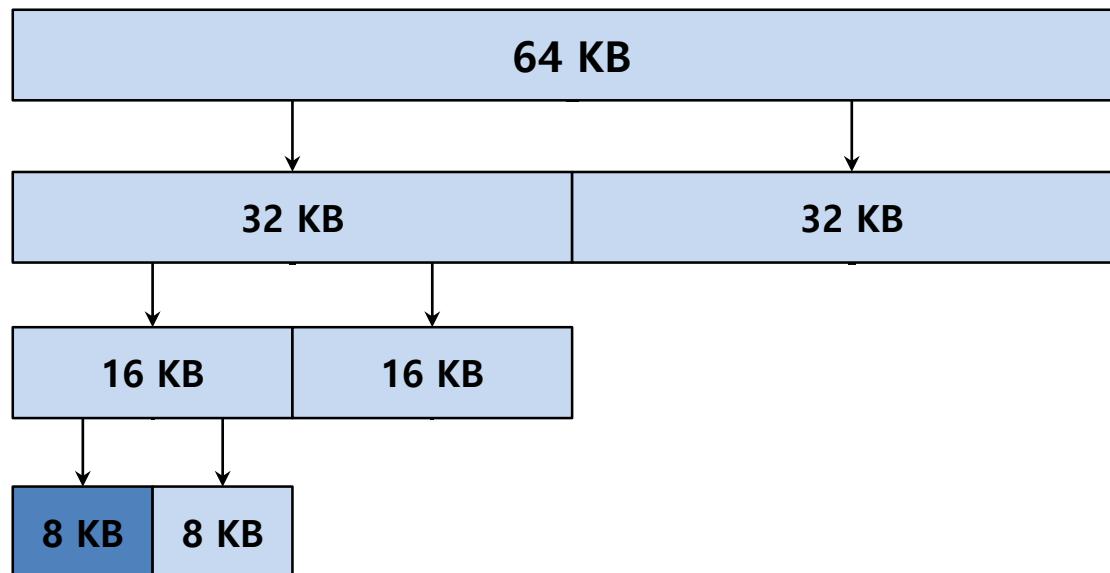
▫ Slab Allocator

- ◆ Allocate a number of object caches.
 - The objects are likely to be requested frequently.
 - e.g., locks, file-system inodes, etc.
- ◆ **Request some memory** from a more general memory allocator when a **given cache is running low** on free space.

Other Approaches: Buddy Allocation

Binary Buddy Allocation

- The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**.



64KB free space for 7KB request

Other Approaches: Buddy Allocation(Cont.)

- ❑ Buddy allocation can suffer from **internal fragmentation**.
- ❑ Buddy system makes **coalescing** simple.
 - ◆ **Coalescing** two blocks into the next level of block.

18. Paging: Introduction

Concept of Paging

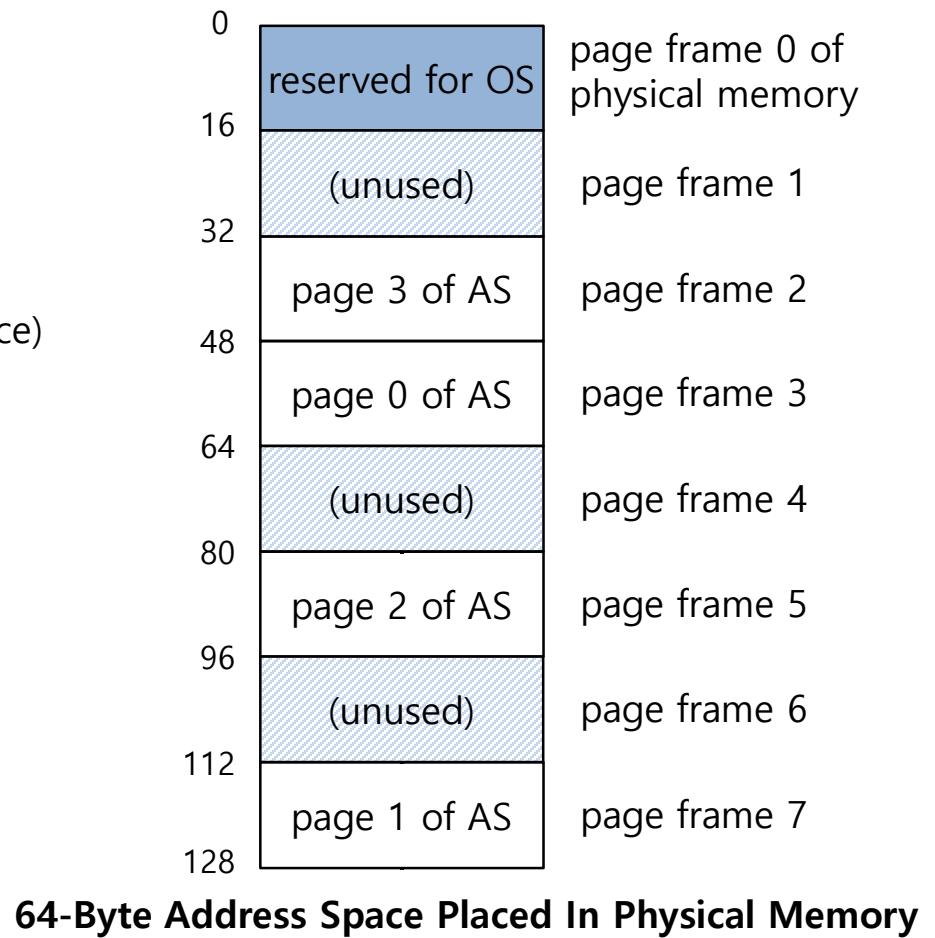
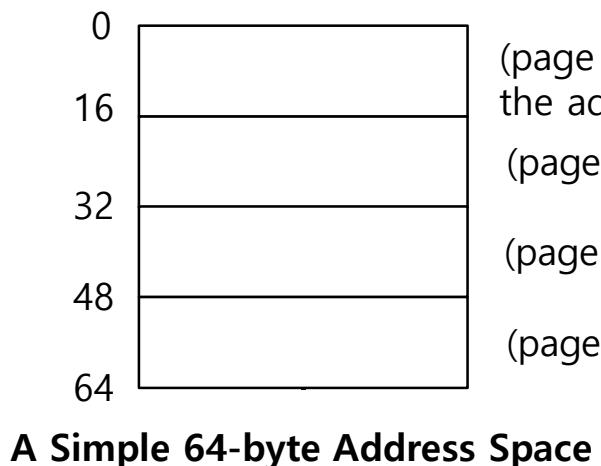
- ▣ Paging **splits up** address space into **fixed-sized** unit called a **page**.
 - ◆ Segmentation: variable size of logical segments(code, stack, heap, etc.)
- ▣ With paging, **physical memory** is also **split** into some number of pages called a **page frame**.
- ▣ **Page table** per process is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

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

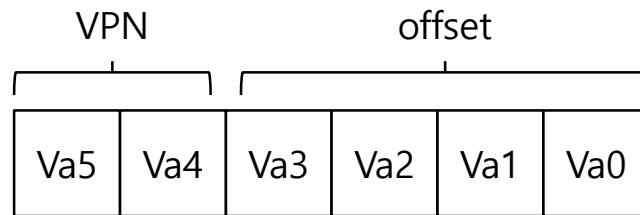
Example: A Simple Paging

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

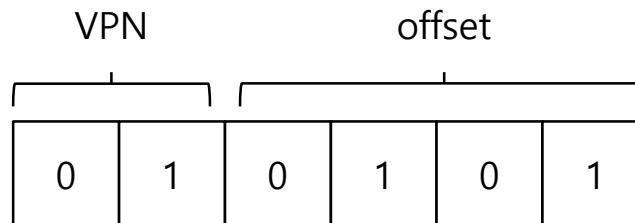


Address Translation

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

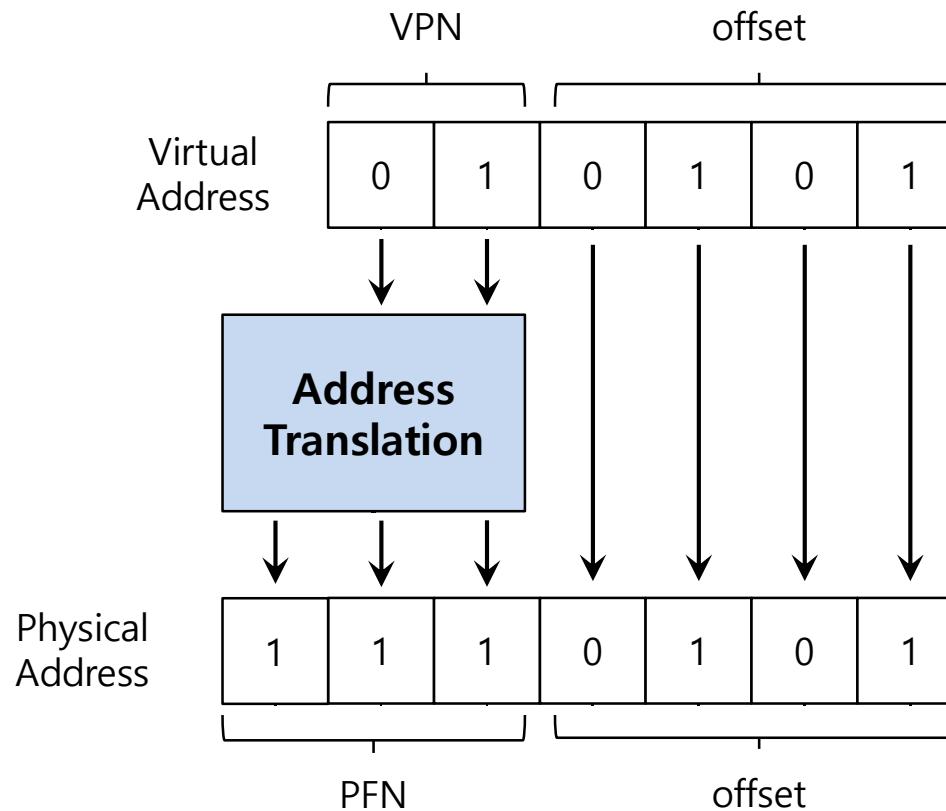


- Example: virtual address 21 in 64-byte address space



Example: Address Translation

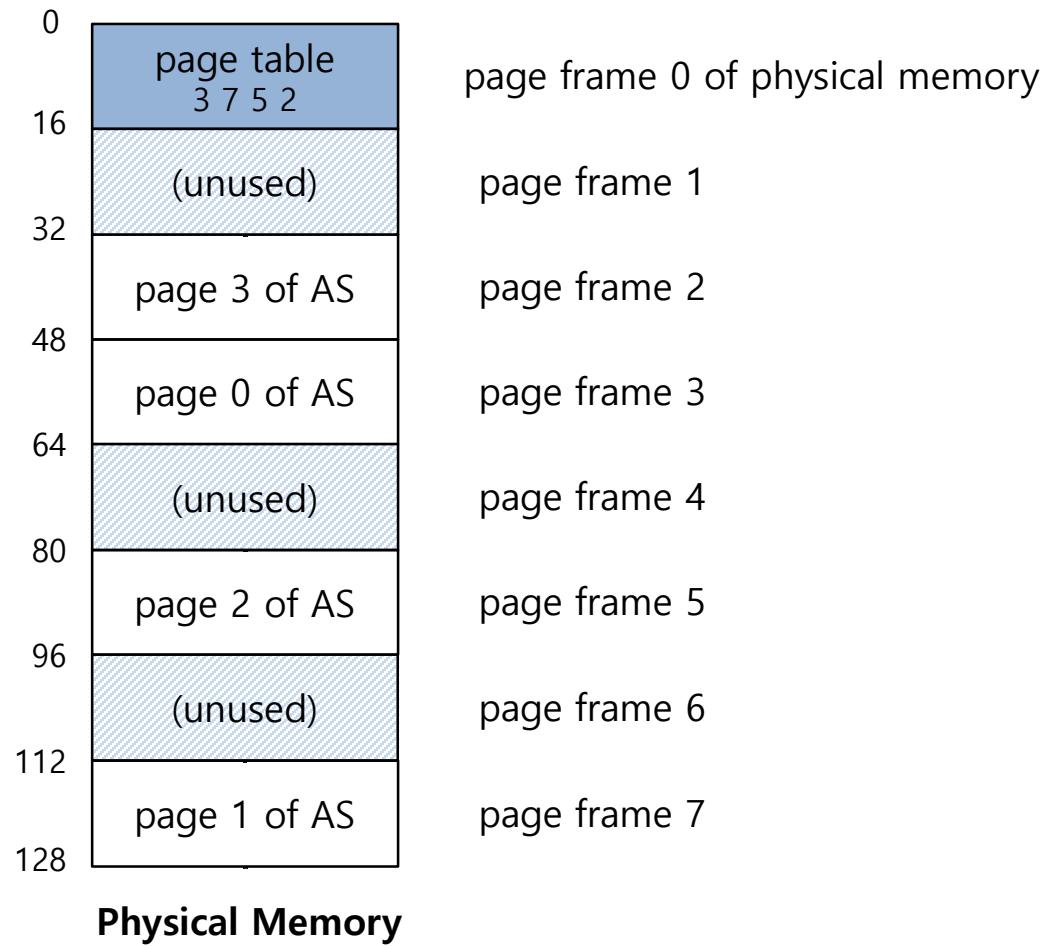
- The virtual address 21 in 64-byte address space



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



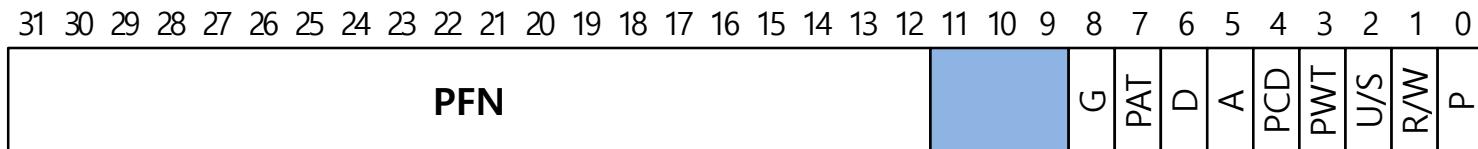
What Is In The Page Table?

- ▣ 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

- **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



An x86 Page Table Entry(PTE)

- ▣ P: present
- ▣ R/W: read/write bit
- ▣ U/S: supervisor
- ▣ A: accessed bit
- ▣ D: dirty bit
- ▣ PFN: the page frame number

Paging: Too Slow

- ❑ To find a location of the desired PTE, the **starting location** of the page table is **needed**.
- ❑ 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
2 VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4 // Form the address of the page-table entry (PTE)
5 PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7 // Fetch the PTE
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;
```

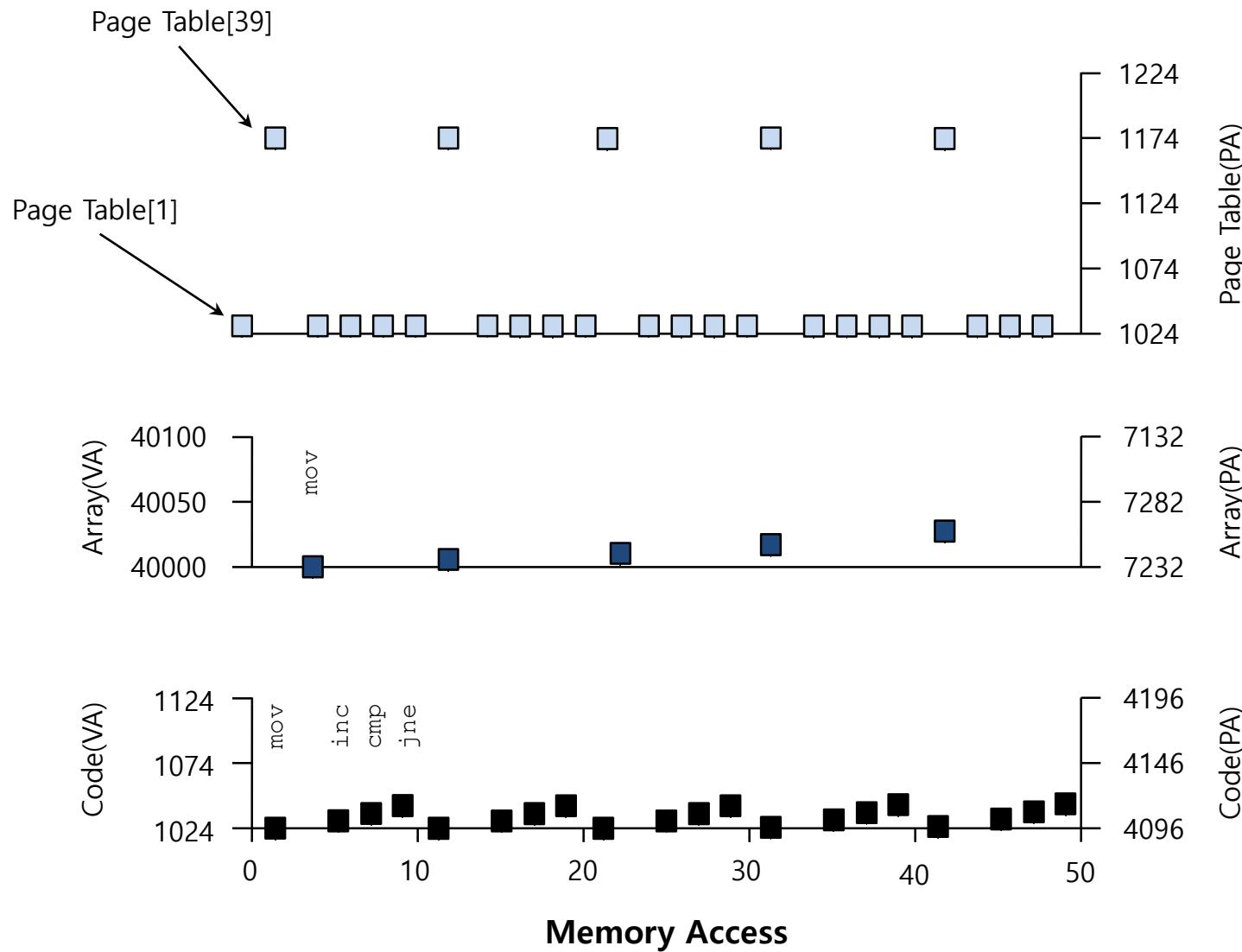
- Compile and execute

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

- Resulting Assembly code

```
0x1024 movl $0x0, (%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

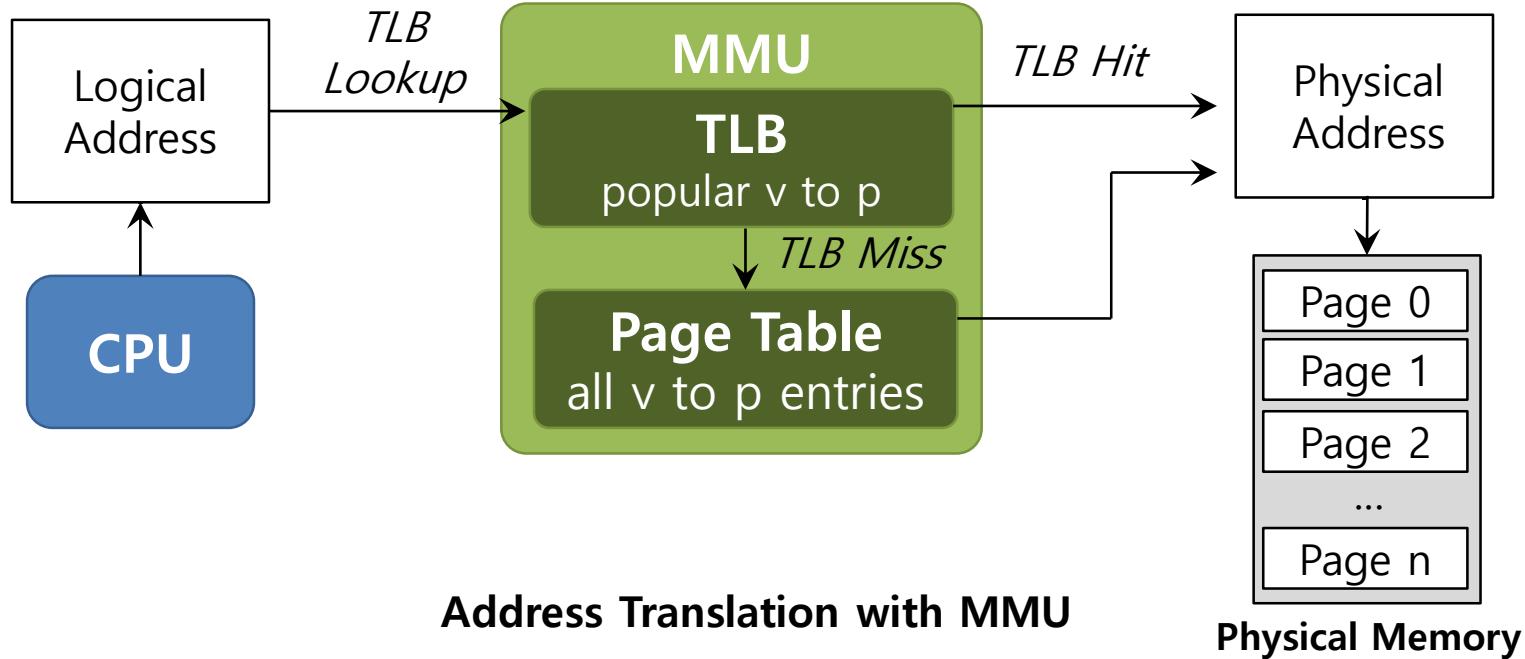
A Virtual(And Physical) Memory Trace



19. Translation Lookaside Buffers

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.

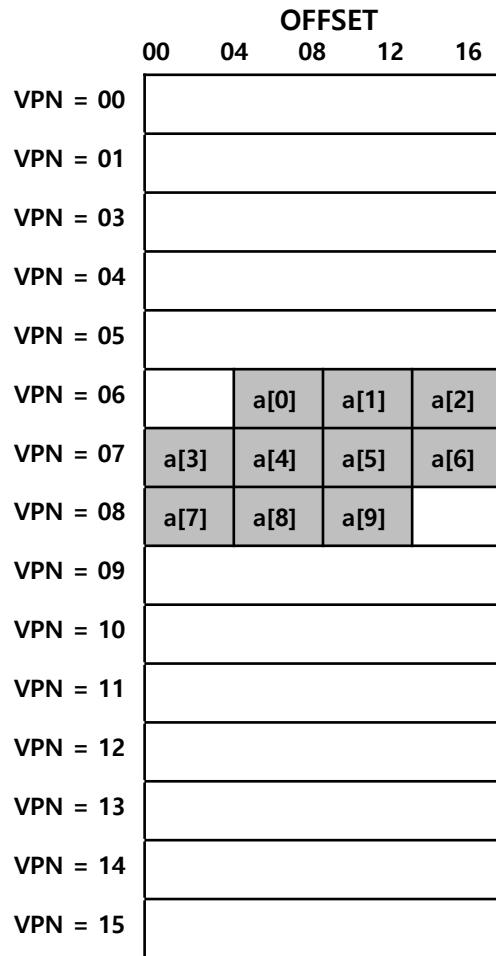
TLB Basic Algorithms (Cont.)

```
11:     }else{ //TLB Miss  
12:         PTEAddr = PTBR + (VPN * sizeof(PTE))  
13:         PTE = AccessMemory(PTEAddr)  
14:         (...)  
15:     }else{  
16:         TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)  
17:         RetryInstruction()  
18:     }  
19: }
```

- ◆ (11-12 lines) The hardware accesses the page table to find the translation.
- ◆ (16 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:     }
```

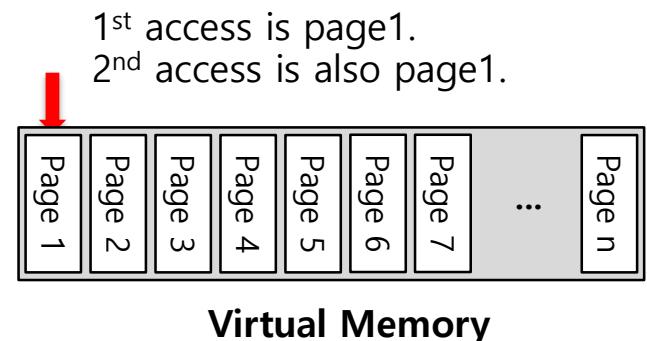
The TLB improves performance
due to spatial locality

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

Locality

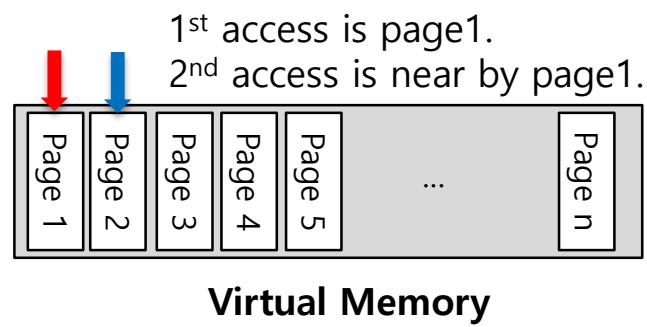
Temporal Locality

- 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, find the correct page-table entry and **extract** the desired translation, **update** and **retry** instruction.
 - ◆ **hardware-managed TLB**.

Who Handles The TLB Miss? (Cont.)

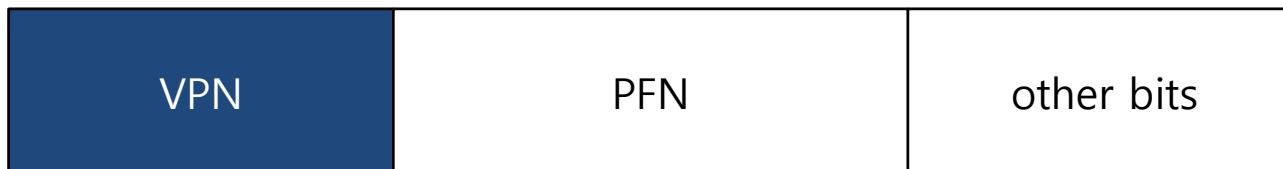
- ▣ RISC have what is known as a software-managed TLB.
 - ◆ On a TLB miss, the hardware raises exception(trap handler).
 - Trap handler is code within the OS that is written with the express purpose of handling TLB miss.

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)
```

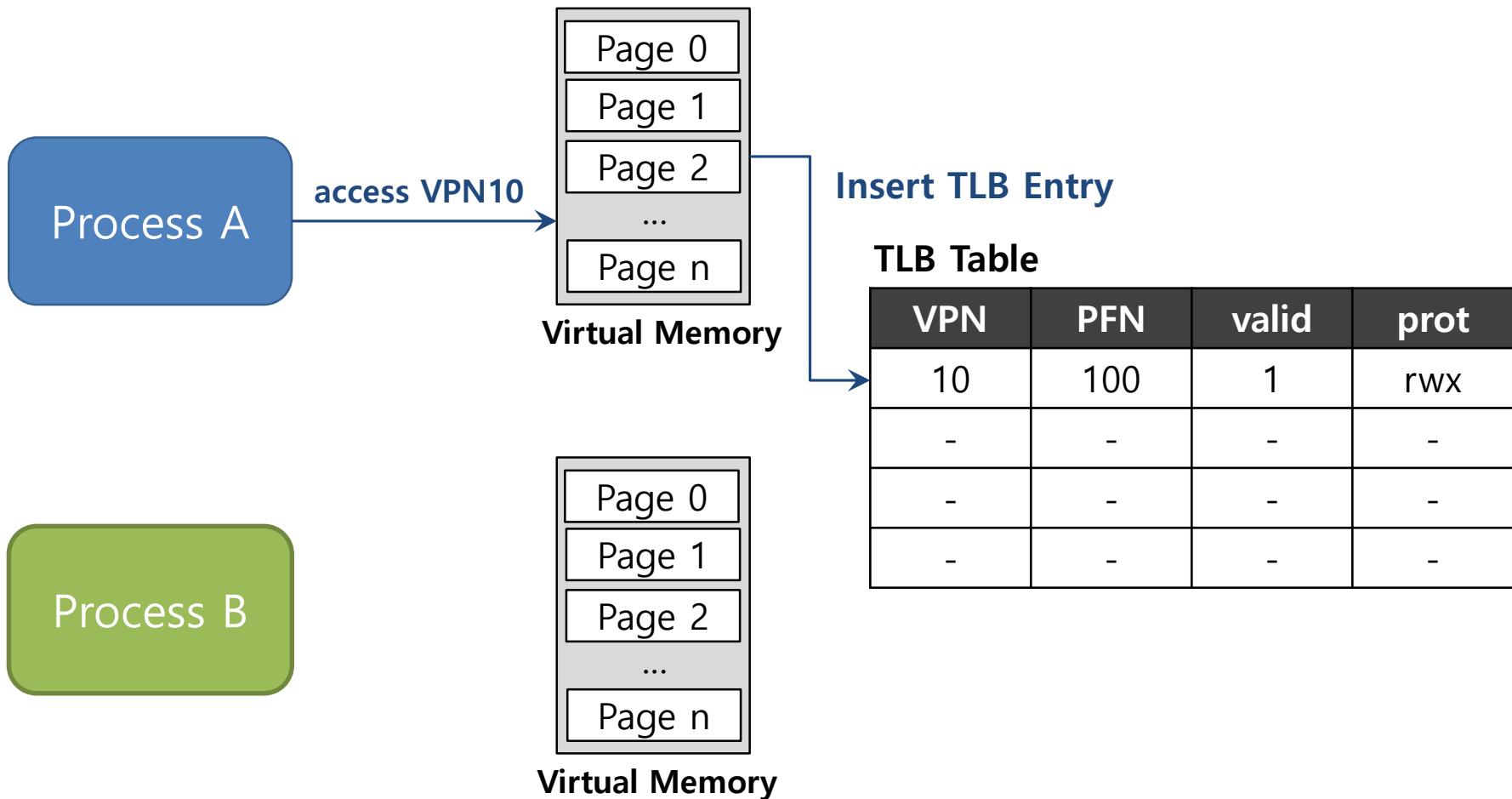
TLB entry

- TLB is managed by **Full Associative** method.
 - ◆ A typical TLB might have 32,64, or 128 entries.
 - ◆ Hardware search the entire TLB in parallel to find the desired translation.
 - ◆ other bits: valid bits , protection bits, address-space identifier, dirty bit

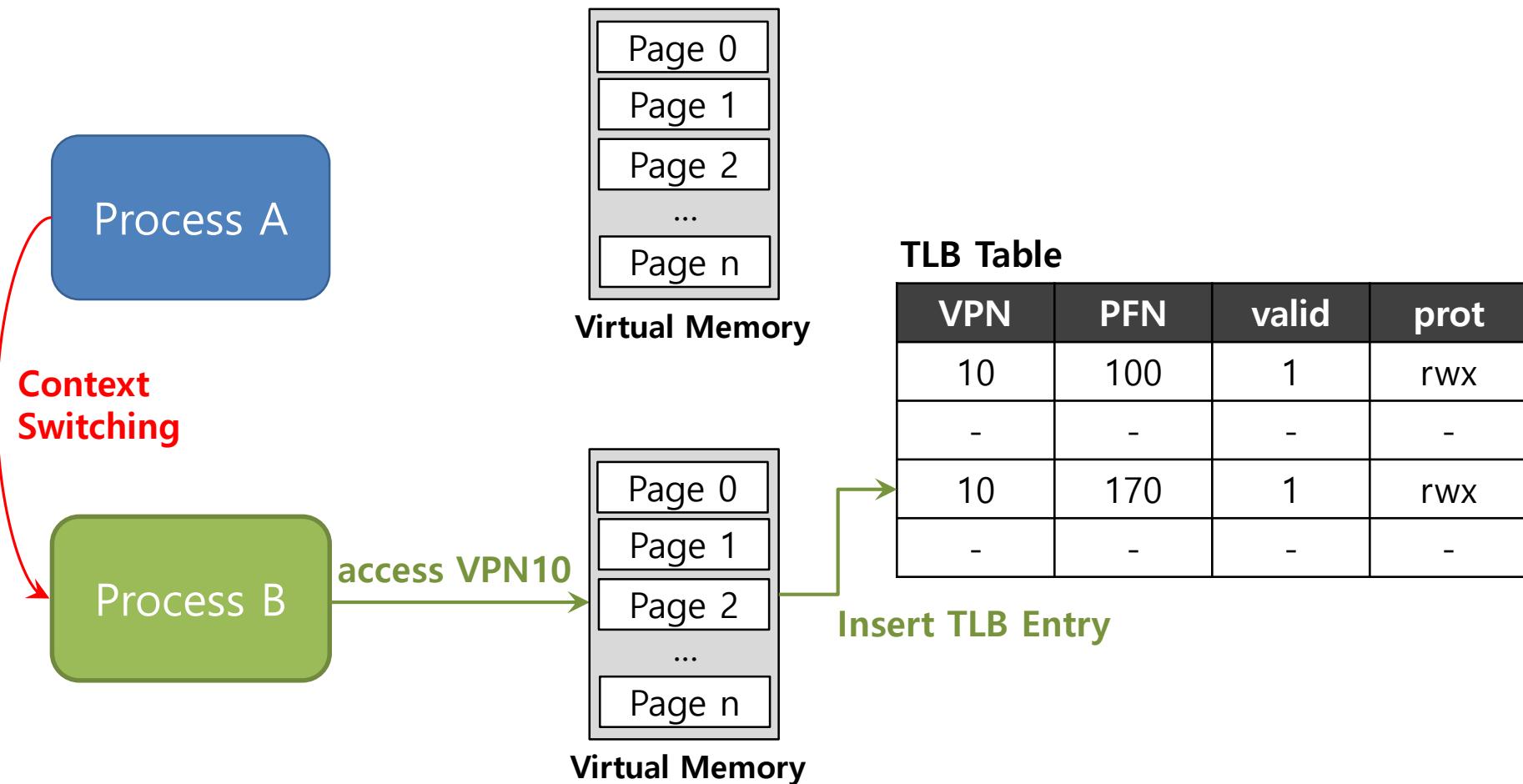


Typical TLB entry look like this

TLB Issue: Context Switching

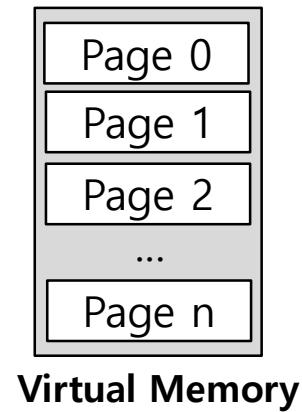


TLB Issue: Context Switching

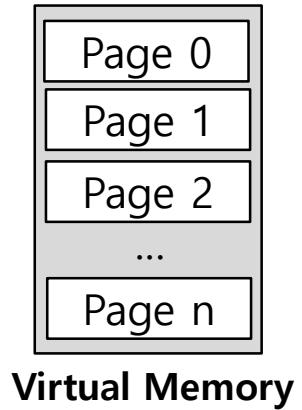


TLB Issue: Context Switching

Process A



Process B



TLB Table

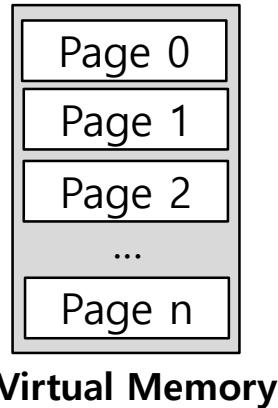
VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

To Solve Problem

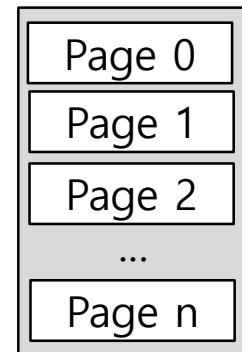
- Provide an address space identifier(ASID) field in the TLB.

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.

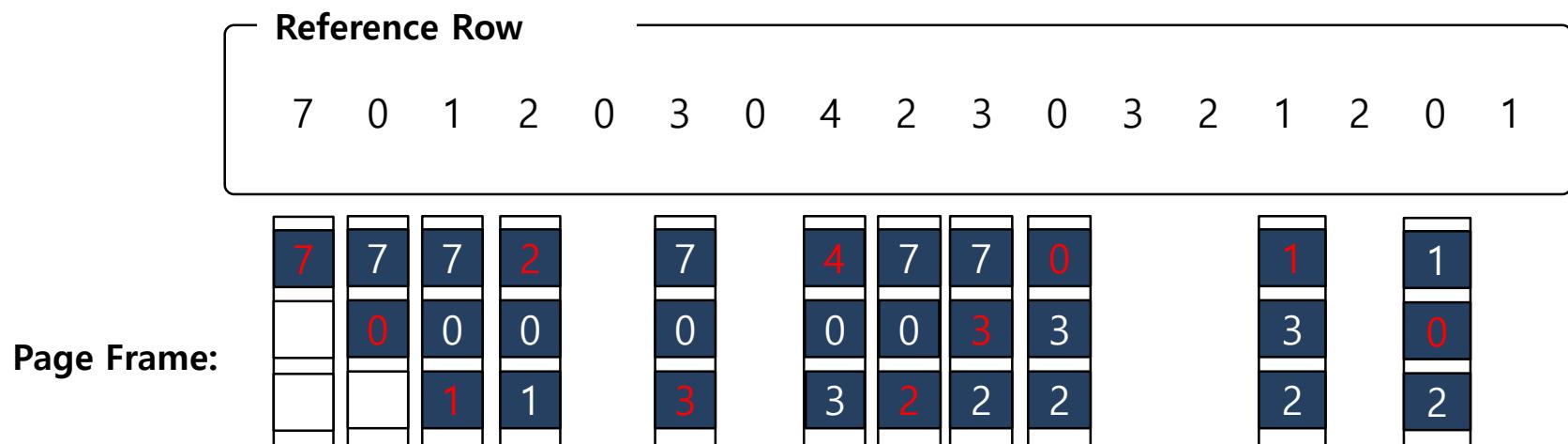
VPN	PFN	valid	prot	ASID
10	101	1	rwx	1
-	-	-	-	-
50	101	1	rwx	2
-	-	-	-	-

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.

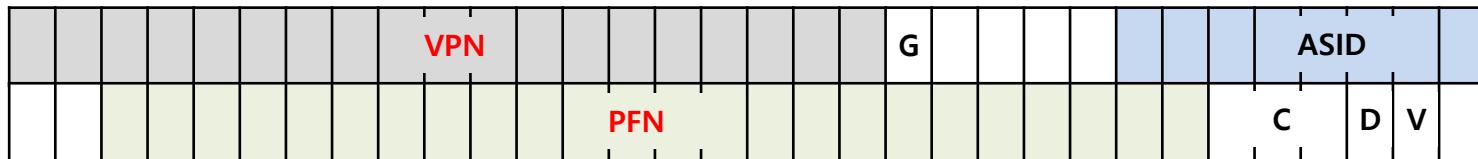


Total 11 TLB miss

A Real TLB Entry

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

0 1 2 3 4 5 6 7 8 9 10 11 ... 19 ... 31

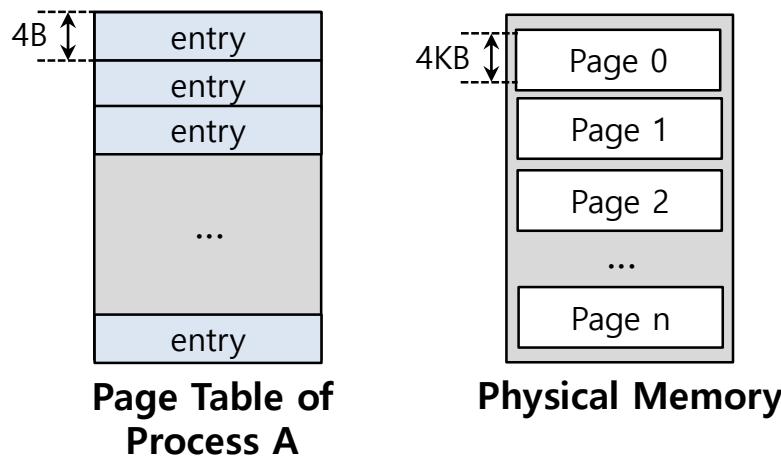


Flag	Content
19-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support up to 64GB of main memory($2^{24} * 4KB$ pages).
Global bit(G)	Used for pages that are globally-shared among processes.
ASID	OS can use to distinguish between address spaces.
Coherence bit(C)	determine how a page 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.

20. Advanced Page Tables

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.

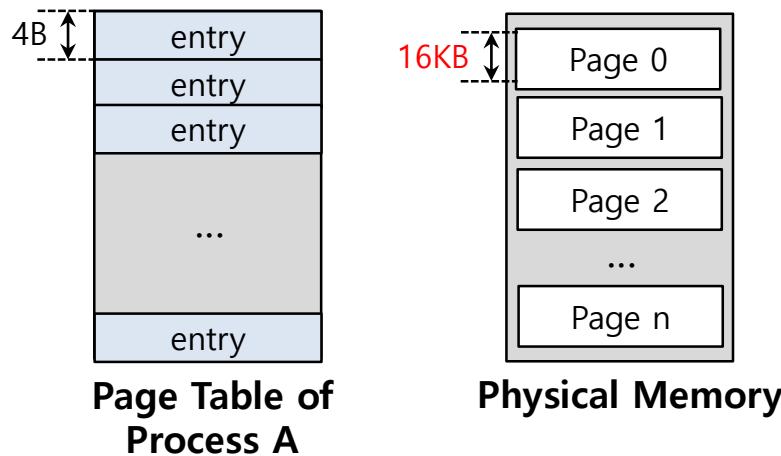


$$\text{Page table size} = \frac{2^{32}}{2^{12}} * 4\text{Byte} = 4\text{MByte}$$

Page table 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.

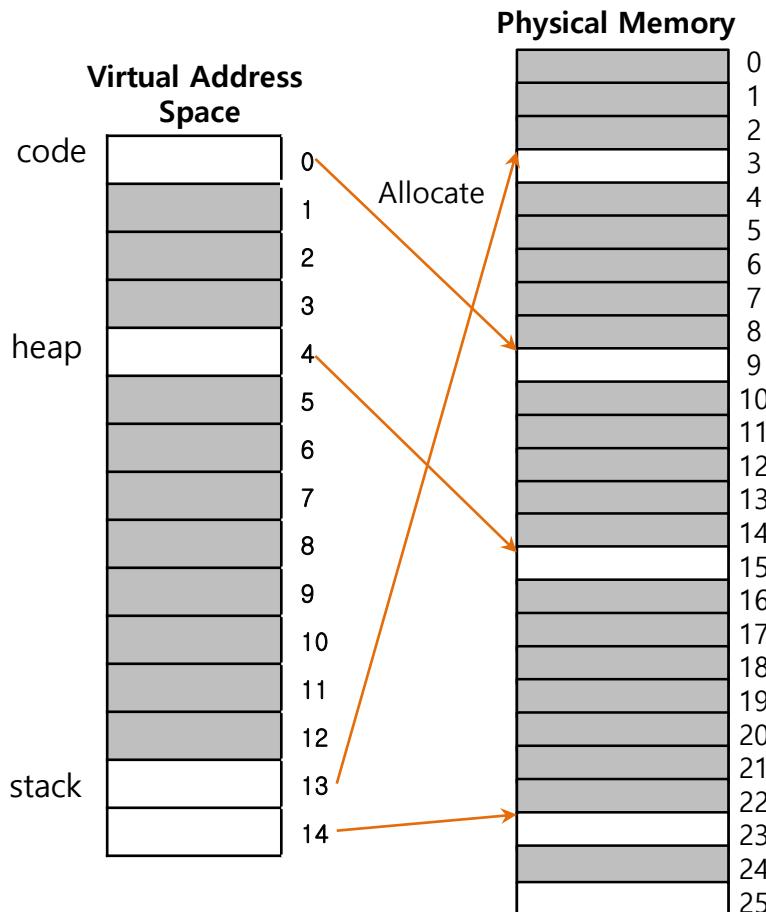


$$\frac{2^{32}}{2^{16}} * 4 = 1MB \text{ per page table}$$

Big pages lead to **internal fragmentation**.

Problem

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



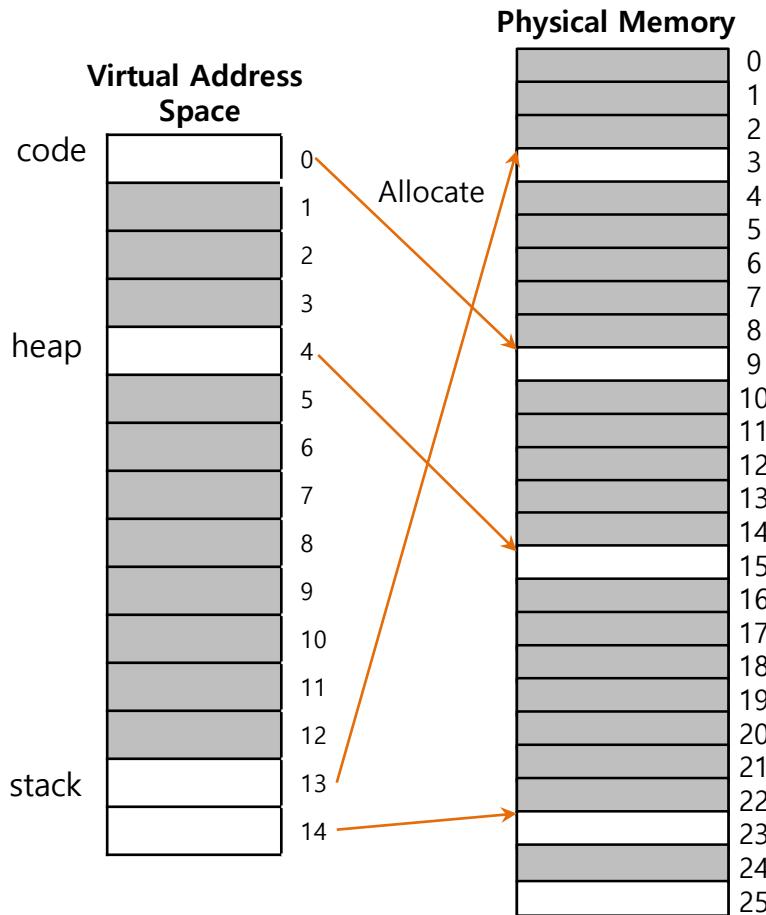
A 16KB Address Space with 1KB Pages

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

Problem

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



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

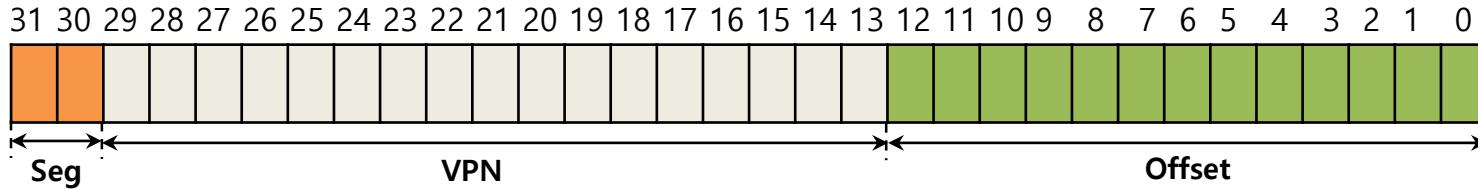
A Page Table For 16KB Address Space

Hybrid Approach: 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))
```

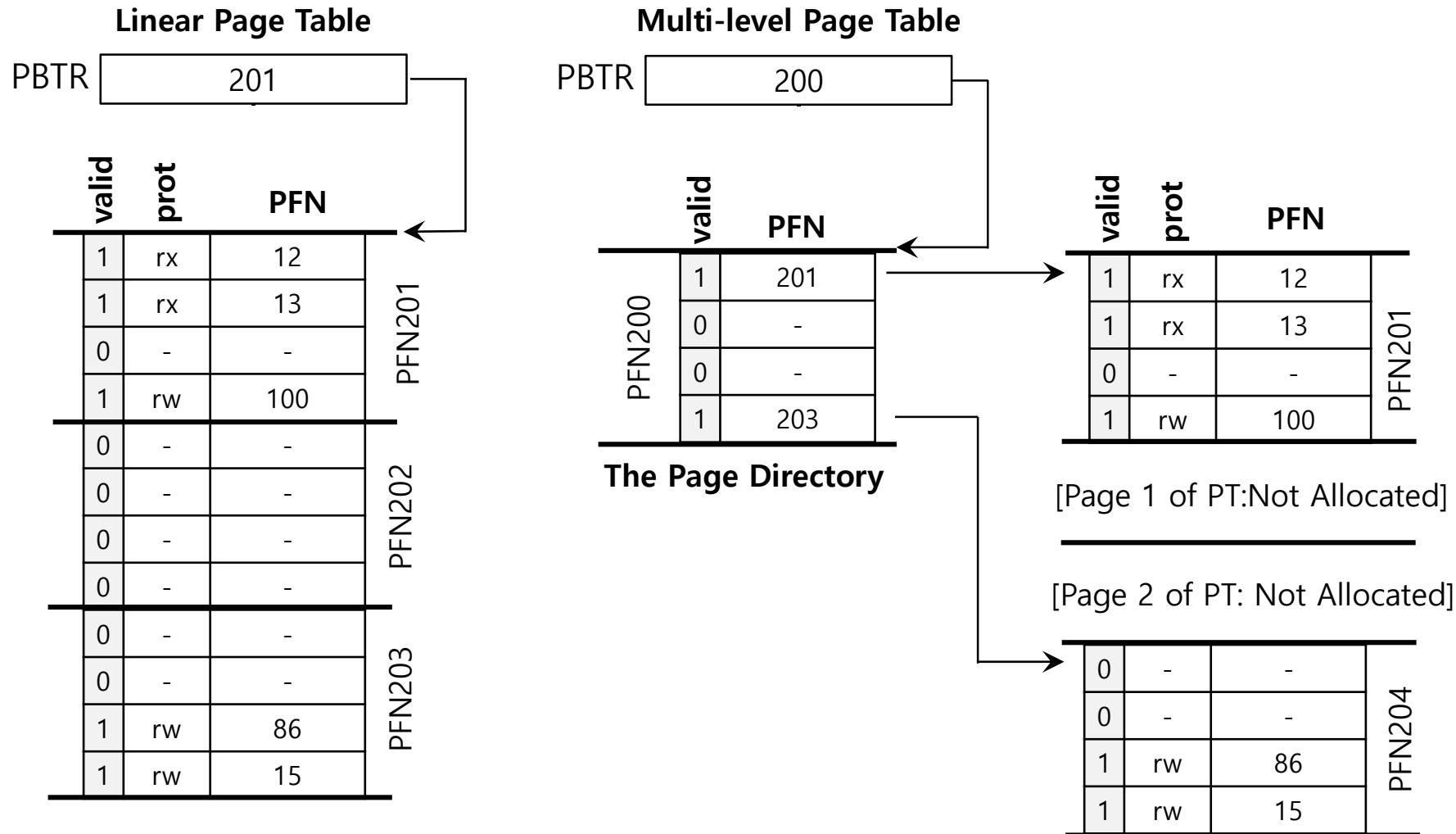
Problem of Hybrid Approach

- ▣ Hybrid Approach is not without problems.
 - ◆ If we have a large but sparsely-used heap, we can still end up with a lot of page table waste.
 - ◆ Causing external fragmentation to arise again.

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 (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 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

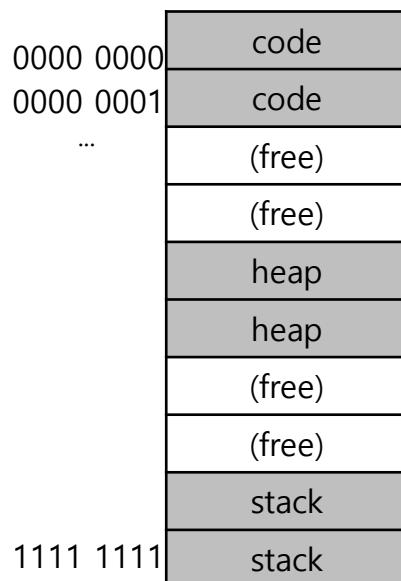
- ◆ Multi-level table is a small example of a **time-space trade-off**.
- ◆ **Complexity**.

Multi-level Page Table: Level of indirection

- A multi-level structure can adjust **level of indirection** through use of the page directory.
 - ◆ Indirection place page-table pages wherever we would like in physical memory.

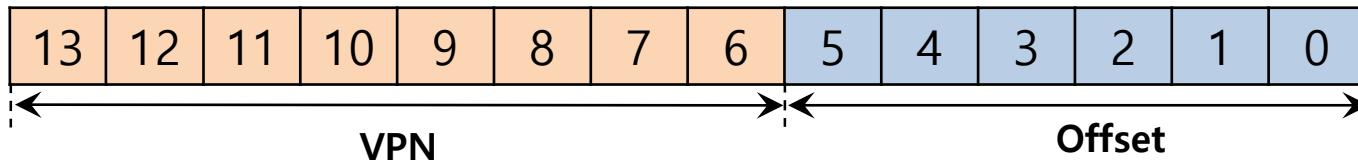
A Detailed Multi-Level Example

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



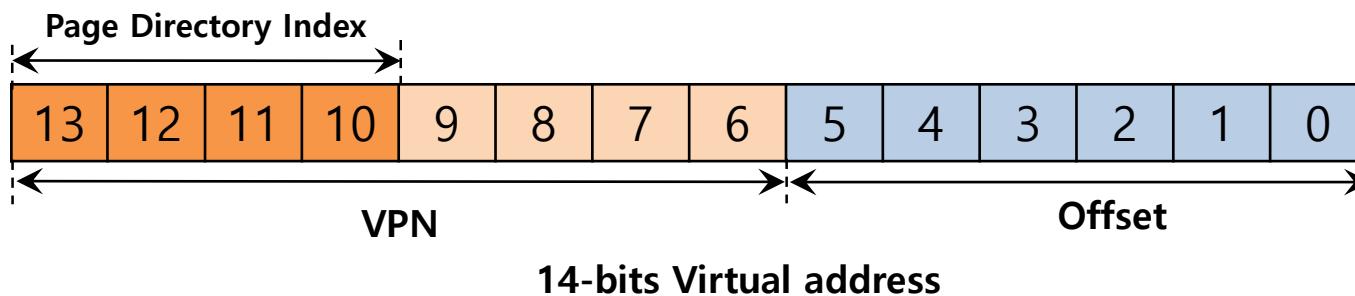
Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
Page table entry	$2^8(256)$

A 16-KB Address Space With 64-byte Pages



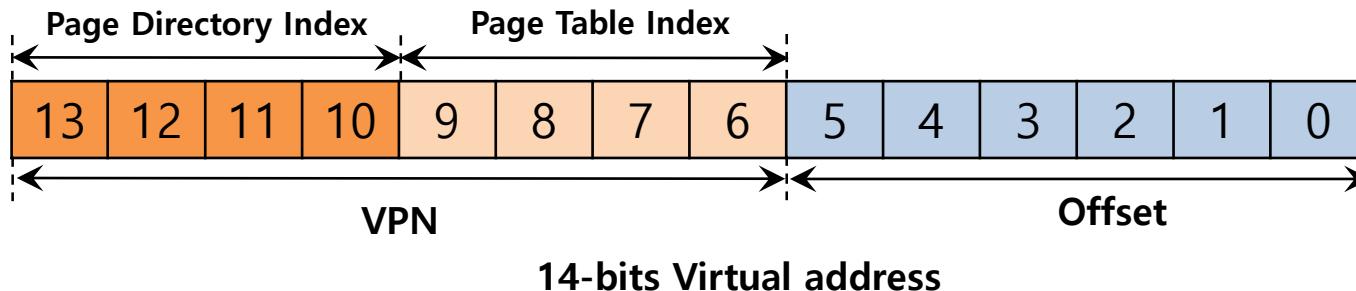
A Detailed Multi-Level Example: Page Directory idx

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



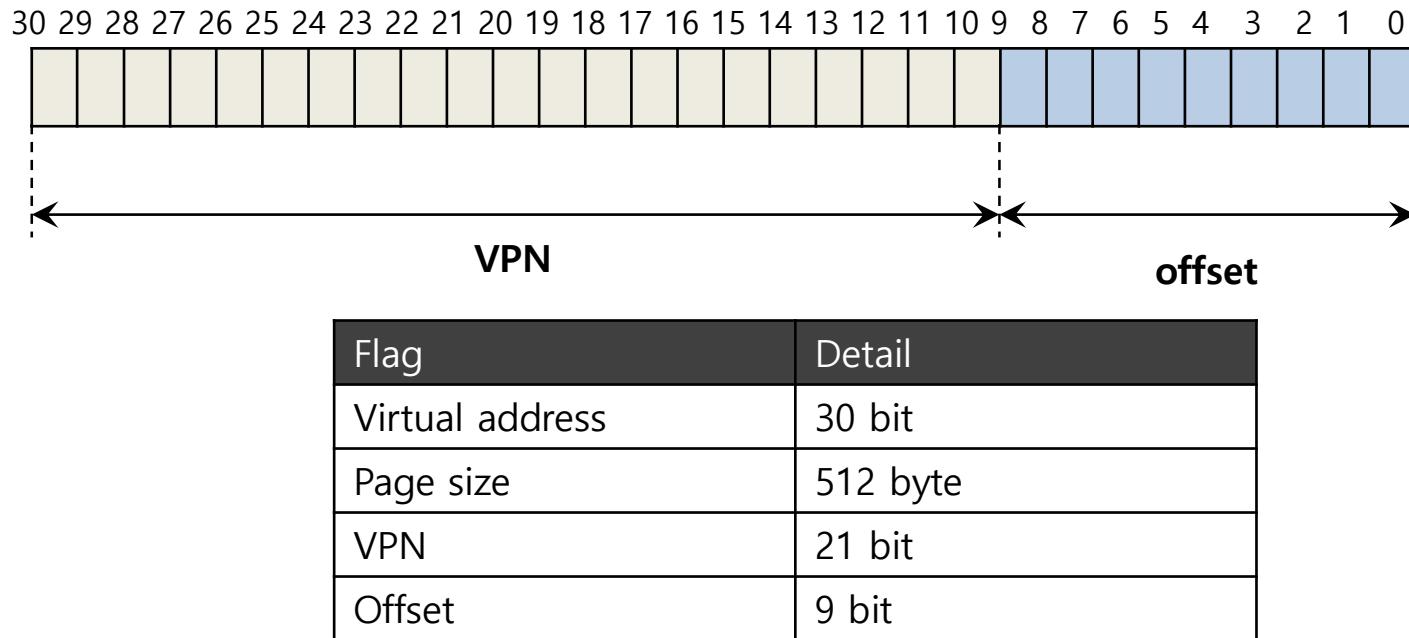
A Detailed Multi-Level Example: Page Table Idx

- 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.



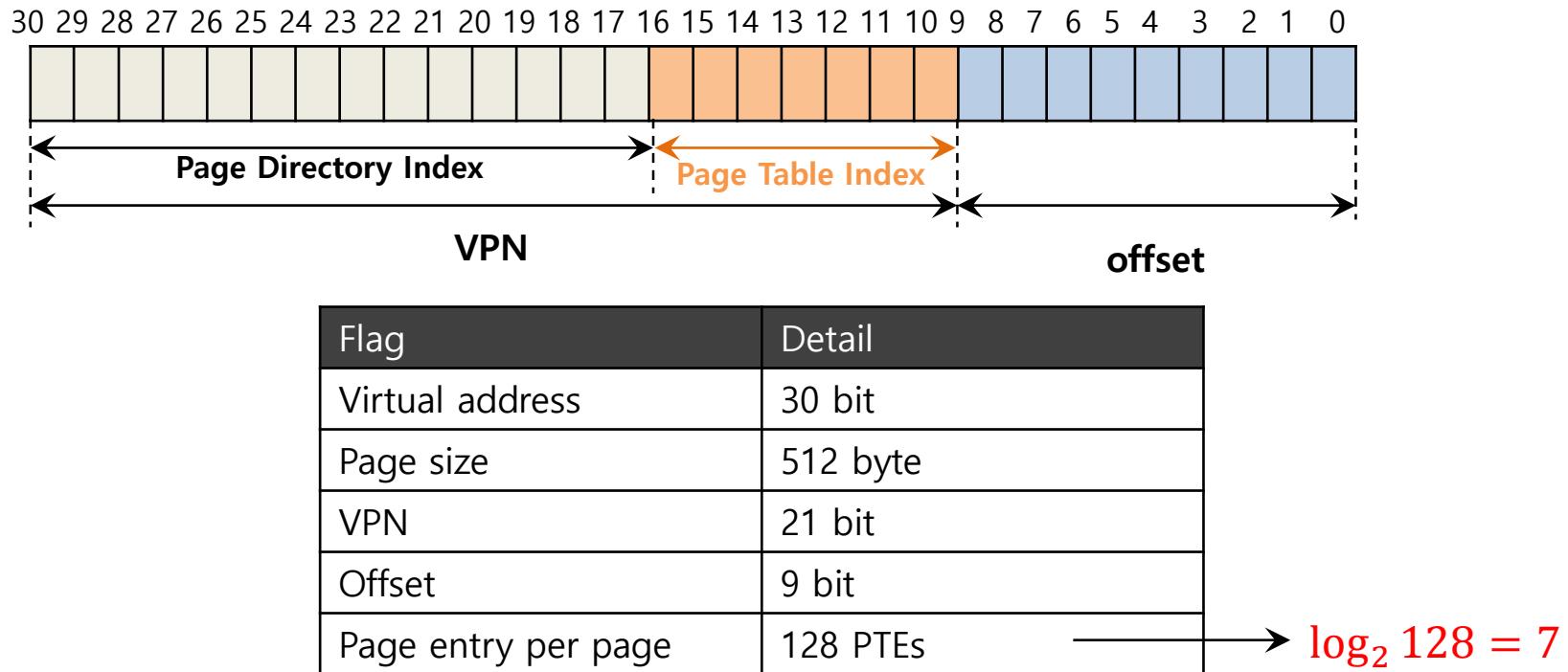
More than Two Level

- In some cases, a deeper tree is possible.



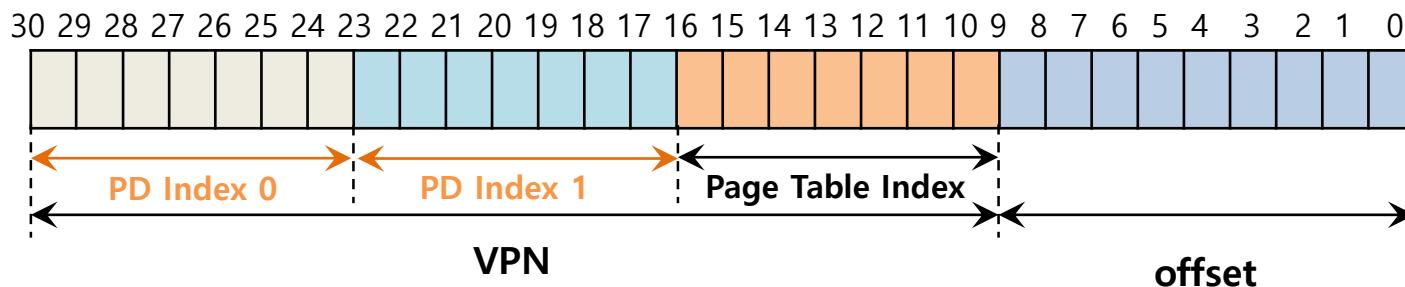
More than Two Level : Page Table Index

- In some cases, a deeper tree is possible.



More than Two Level : Page Directory

- If our page directory has 2^{14} entries, it spans not one page but 128.
- 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()
```

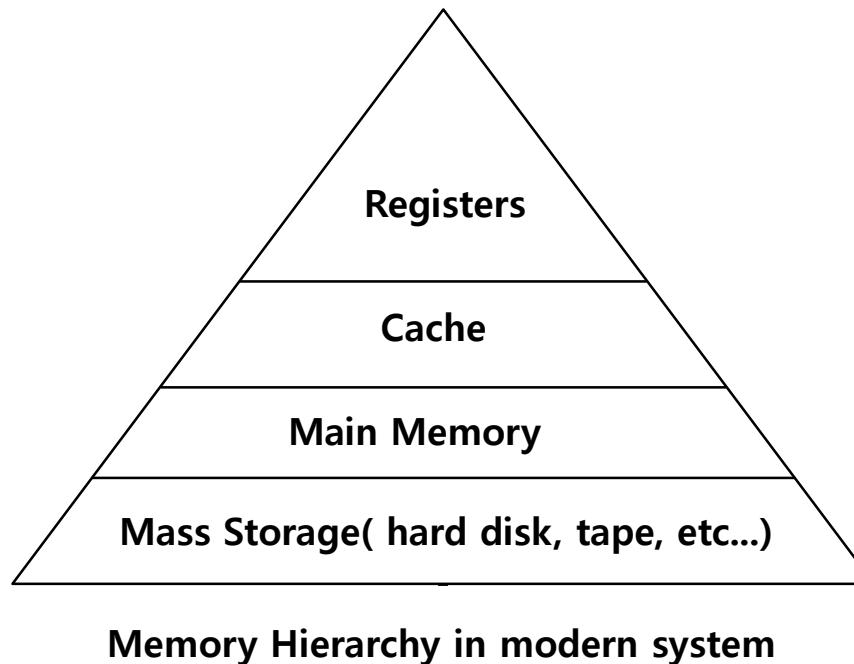
Inverted Page Tables

- ▣ 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.

21. Swapping: Mechanisms

Beyond Physical Memory: Mechanisms

- Require an additional level in the **memory hierarchy**.
 - OS need a place to stash away portions of address space that currently aren't in great demand.
 - In modern systems, this role is usually served by a **hard disk drive**

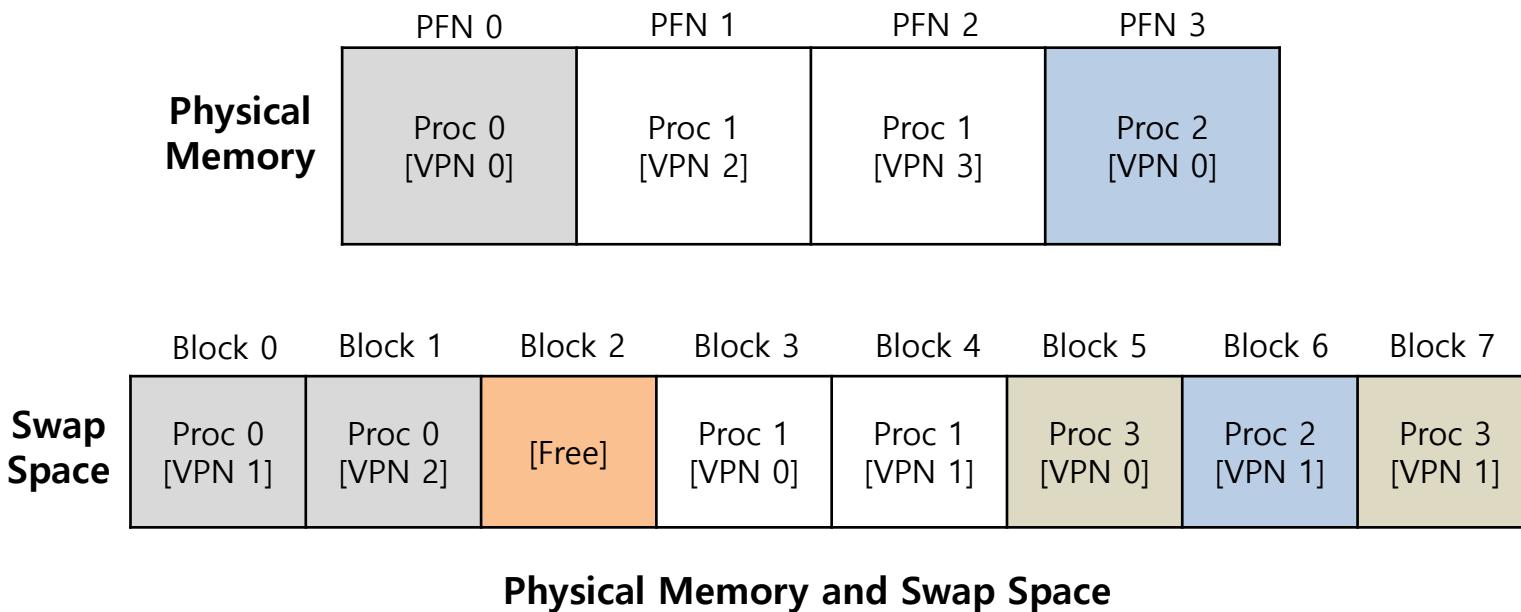


Single large address for a process

- ▣ Always need to first arrange for the code or data to be in memory when before calling a function or accessing data.
- ▣ To Beyond just a **single process**.
 - ◆ The addition of **swap space** allows the OS to support the illusion of a large virtual memory for multiple concurrently-running process

Swap Space

- Reserve some space on the disk for moving pages back and forth.
- OS need to remember to the swap space, in **page-sized unit**



Present Bit

- ▣ Add some machinery higher up in the system in order to support swapping pages to and from the disk.
 - ◆ When the hardware looks in the PTE, it may find that the page is not present in physical memory.

Value	Meaning
1	page is present in physical memory
0	The page is not in memory but rather on disk.

What If Memory Is Full ?

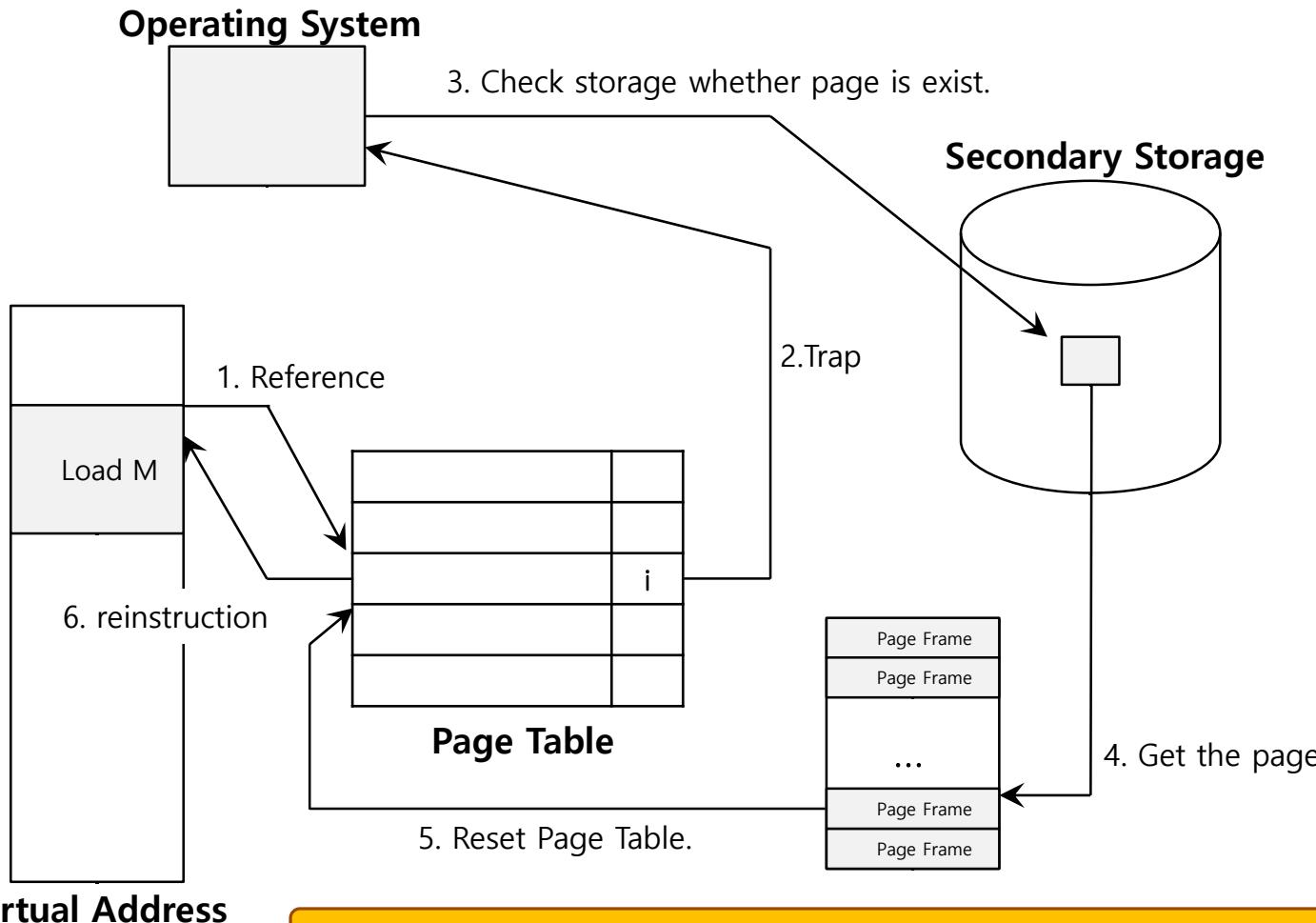
- ▣ The OS like to page out pages to make room for the new pages the OS is about to bring in.
 - ◆ The process of picking a page to kick out, or replace is known as **page-replacement** policy

The Page Fault

- ▣ Accessing page that is **not in physical memory**.
 - ◆ If a page is not present and has been swapped disk, the OS need to swap the page into memory in order to service the page fault.

Page Fault Control Flow

- PTE used for data such as the PFN of the page for a disk address.



Page Fault Control Flow – Hardware

```
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 RaiseException(PROTECTION_FAULT)
```

Page Fault Control Flow – Hardware

```
9:      else // TLB Miss
10:     PTEAddr = PTBR + (VPN * sizeof(PTE) )
11:     PTE = AccessMemory(PTEAddr)
12:     if (PTE.Valid == False)
13:         RaiseException(SEGMENTATION_FAULT)
14:     else
15:       if (CanAccess(PTE.ProtectBits) == False)
16:           RaiseException(PROTECTION_FAULT)
17:       else if (PTE.Present == True)
18:           // assuming hardware-managed TLB
19:           TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
20:           RetryInstruction()
21:       else if (PTE.Present == False)
22:           RaiseException(PAGE_FAULT)
```

Page Fault Control Flow – Software

```
1:      PFN = FindFreePhysicalPage()
2:      if (PFN == -1) // no free page found
3:          PFN = EvictPage() // run replacement algorithm
4:          DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5:          PTE.present = True // update page table with present
6:          PTE.PFN = PFN // bit and translation (PFN)
7:          RetryInstruction() // retry instruction
```

- ◆ The OS must find a physical frame for the **soon-be-faulted-in page** to reside within.
- ◆ If there is no such page, waiting for the **replacement algorithm** to run and kick some pages out of memory.

When Replacements Really Occur

- ▣ OS waits until memory is entirely full, and only then replaces a page to make room for some other page
 - ◆ This is a little bit unrealistic, and there are many reason for the OS to keep a small portion of memory free more proactively.
- ▣ Swap Daemon, Page Daemon
 - ◆ There are fewer than **LW pages** available, a background thread that is responsible for freeing memory runs.
 - ◆ The thread evicts pages until there are **HW pages** available.

22. Swapping: Policies

Beyond Physical Memory: Policies

- Memory pressure forces the OS to start **paging out** pages to make room for actively-used pages.
- Deciding which page to evict is encapsulated within the replacement policy of the OS.

Cache Management

- Goal in picking a replacement policy for this cache is to minimize the number of cache misses.
- The number of cache hits and misses let us calculate the *average memory access time(AMAT)*.

$$AMAT = (P_{Hit} * T_M) + (P_{Miss} * T_D)$$

Argument	Meaning
T_M	The cost of accessing memory
T_D	The cost of accessing disk
P_{Hit}	The probability of finding the data item in the cache(a hit)
P_{Miss}	The probability of not finding the data in the cache(a miss)

The Optimal Replacement Policy

- ▣ Leads to the fewest number of misses overall
 - ◆ Replaces the page that will be accessed furthest in the future
 - ◆ Resulting in the **fewest-possible** cache misses
- ▣ Serve only as a comparison point, to know how close we are to **perfect**

Tracing the Optimal Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = 54.6\%$

Future is not known.

A Simple Policy: FIFO

- ▣ Pages were placed in a queue when they enter the system.
- ▣ When a replacement occurs, the page on the tail of the queue(the "**First-in**" pages) is evicted.
 - ◆ It is simple to implement, but can't determine the importance of blocks.

Tracing the FIFO Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

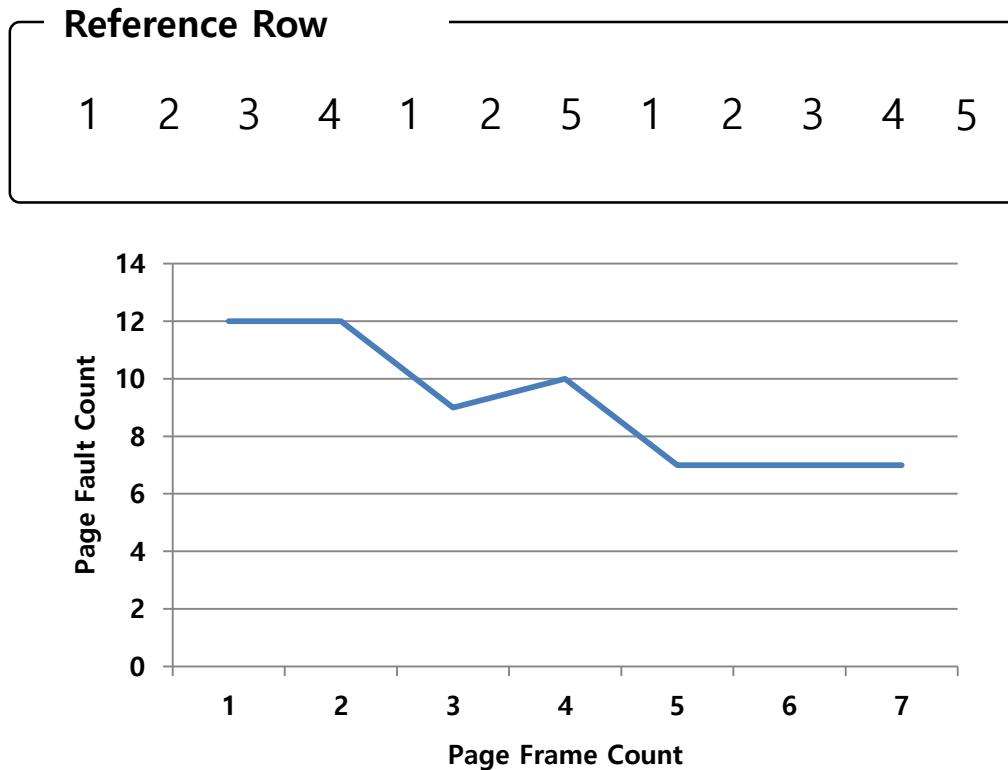
Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = 36.4\%$

Even though page 0 had been accessed a number of times, FIFO still kicks it out.

BELADY' S ANOMALY

- We would expect the cache hit rate to **increase** when the cache gets larger. But in this case, with FIFO, it gets worse.



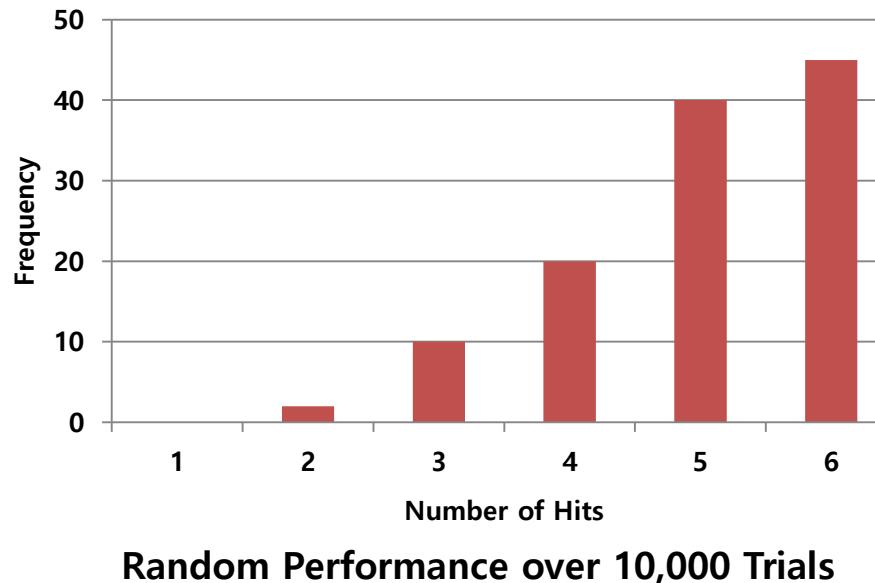
Another Simple Policy: Random

- ❑ Picks a random page to replace under memory pressure.
 - ◆ It doesn't really try to be too intelligent in picking which blocks to evict.
 - ◆ Random does depends entirely upon how lucky Random gets in its choice.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss	3	2,0,1
2	Hit		2,0,1
1	Hit		2,0,1

Random Performance

- ❑ Sometimes, Random is as good as optimal, achieving 6 hits on the example trace.



Using History

- Lean on the past and use history.
 - Two type of historical information.

Historical Information	Meaning	Algorithms
recency	The more recently a page has been accessed, the more likely it will be accessed again	LRU
frequency	If a page has been accessed many times, It should not be replaced as it clearly has some value	LFU

Using History : LRU

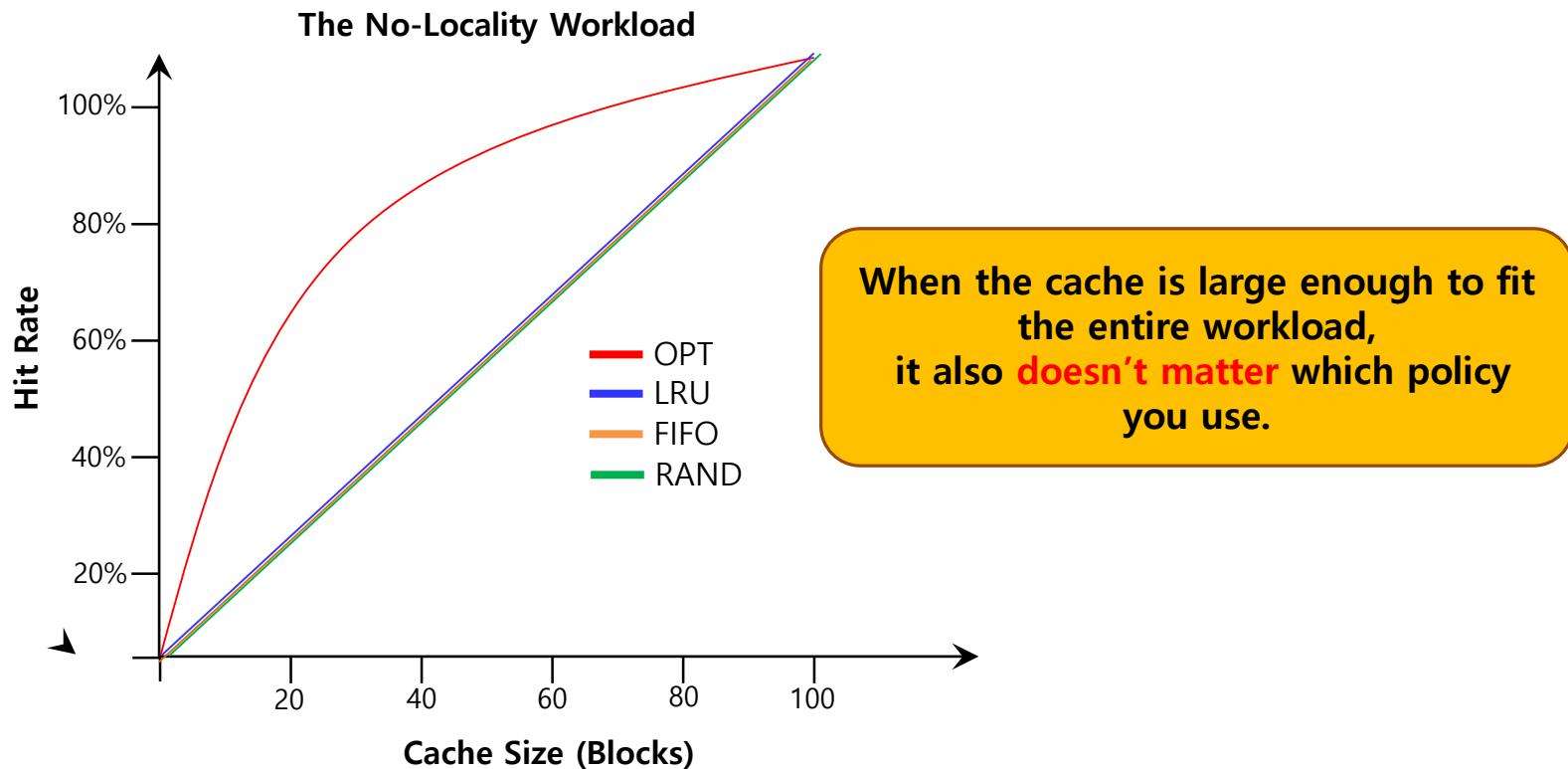
- Replaces the least-recently-used page.

Reference Row										
0	1	2	0	1	3	0	3	1	2	1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		1,2,0
1	Hit		2,0,1
3	Miss	2	0,1,3
0	Hit		1,3,0
3	Hit		1,0,3
1	Hit		0,3,1
2	Miss	0	3,1,2
1	Hit		3,2,1

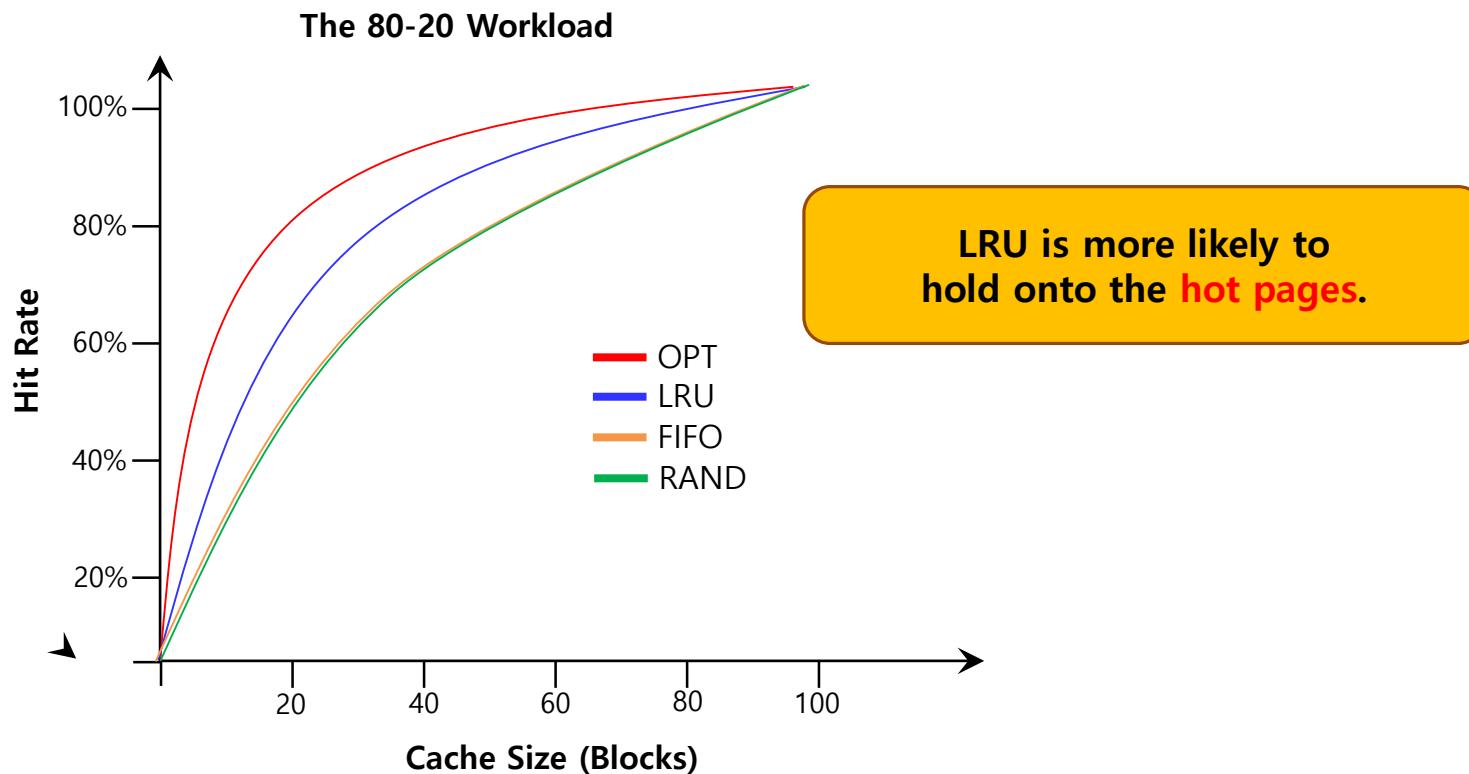
Workload Example : The No-Locality Workload

- Each reference is to a random page within the set of accessed pages.
 - Workload accesses 100 unique pages over time.
 - Choosing the next page to refer to at random



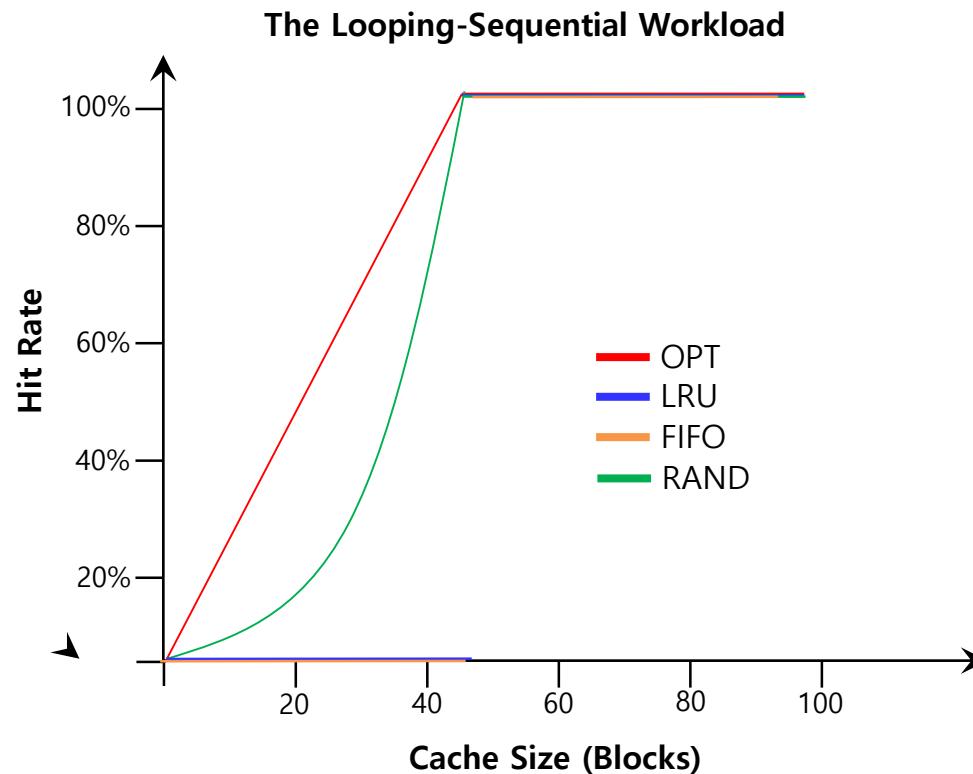
Workload Example : The 80-20 Workload

- Exhibits locality: 80% of the **reference** are made to 20% of the page
- The remaining 20% of the **reference** are made to the remaining 80% of the pages.



Workload Example : The Looping Sequential

- Refer to 50 pages in sequence.
 - Starting at 0, then 1, ... up to page 49, and then we Loop, repeating those accesses, for total of 10,000 accesses to 50 unique pages.



Implementing Historical Algorithms

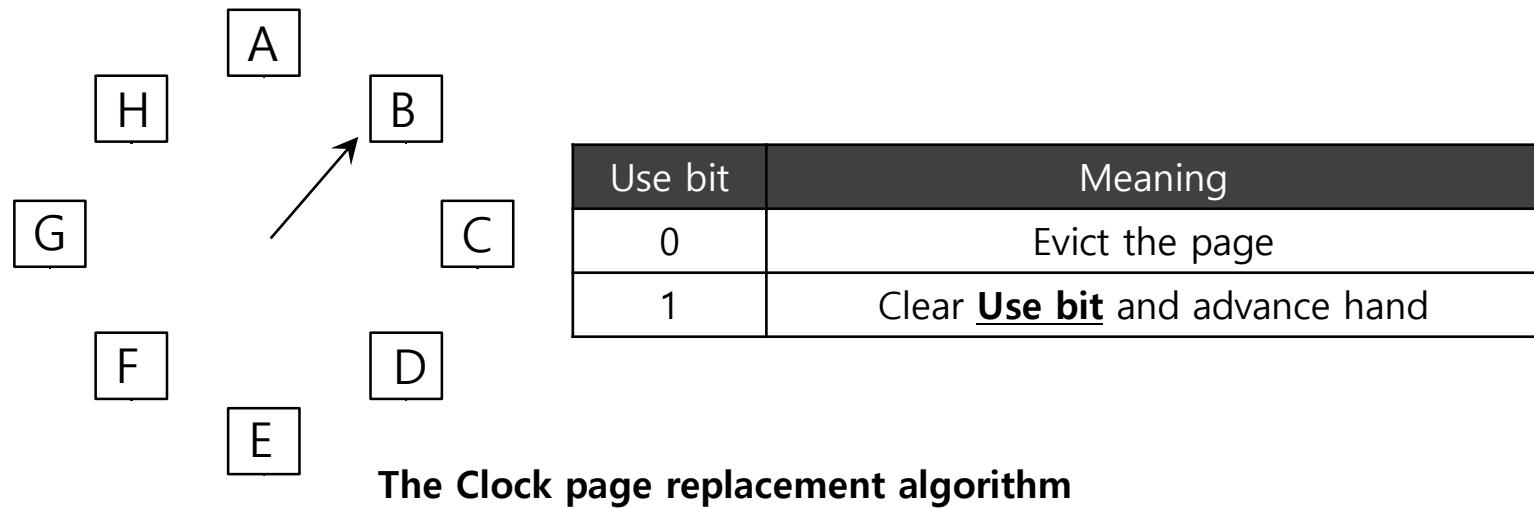
- ❑ To keep track of which pages have been least-and-recently used, the system has to do some accounting work on **every memory reference.**
 - ◆ Add a little bit of hardware support.

Approximating LRU

- ▣ Require some hardware support, in the form of a use bit
 - ◆ Whenever a page is referenced, the use bit is set by hardware to 1.
 - ◆ Hardware never clears the bit, though; that is the responsibility of the OS
- ▣ Clock Algorithm
 - ◆ All pages of the system arrange in a circular list.
 - ◆ A clock hand points to some particular page to begin with.

Clock Algorithm

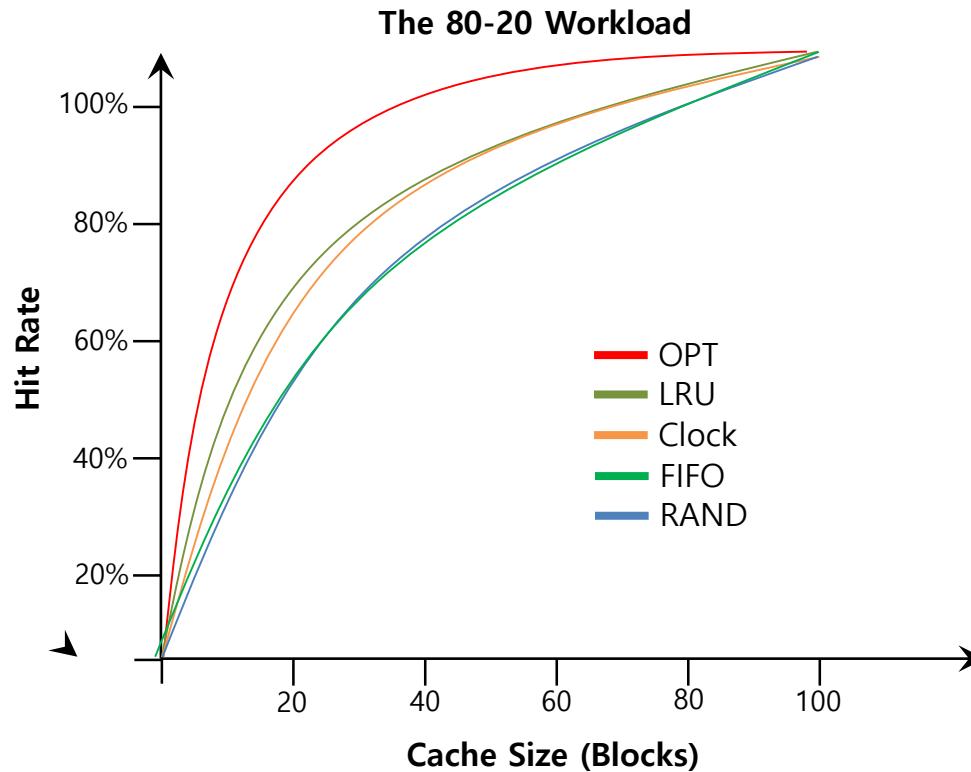
- The algorithm continues until it finds a use bit that is set to 0.



When a page fault occurs, the page the hand is pointing to is inspected.
The action taken depends on the Use bit

Workload with Clock Algorithm

- Clock algorithm doesn't do as well as perfect LRU, it does better than approach that don't consider history at all.



Considering Dirty Pages

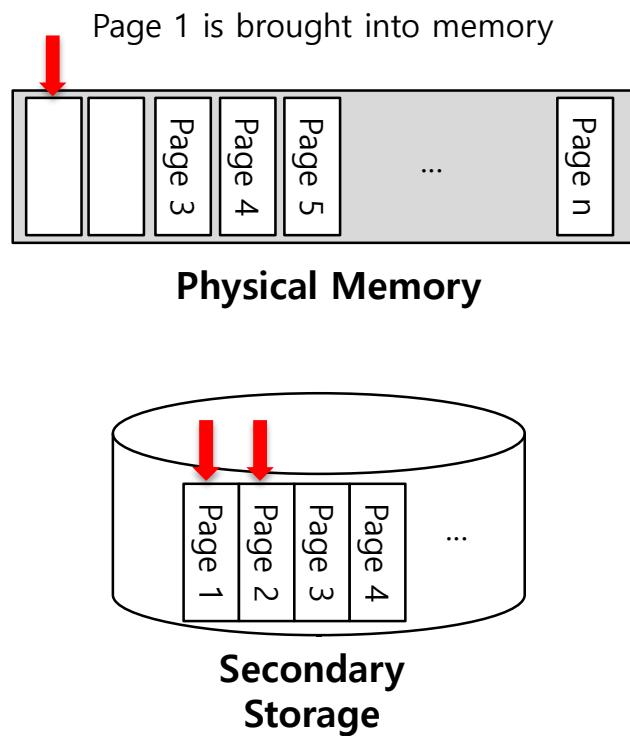
- ▣ The hardware include a modified bit (a.k.a dirty bit)
 - ◆ Page has been modified and is thus dirty, it must be written back to disk to evict it.
 - ◆ Page has not been modified, the eviction is free.

Page Selection Policy

- The OS has to decide when to bring a page into memory.
- Presents the OS with some **different options**.

Prefetching

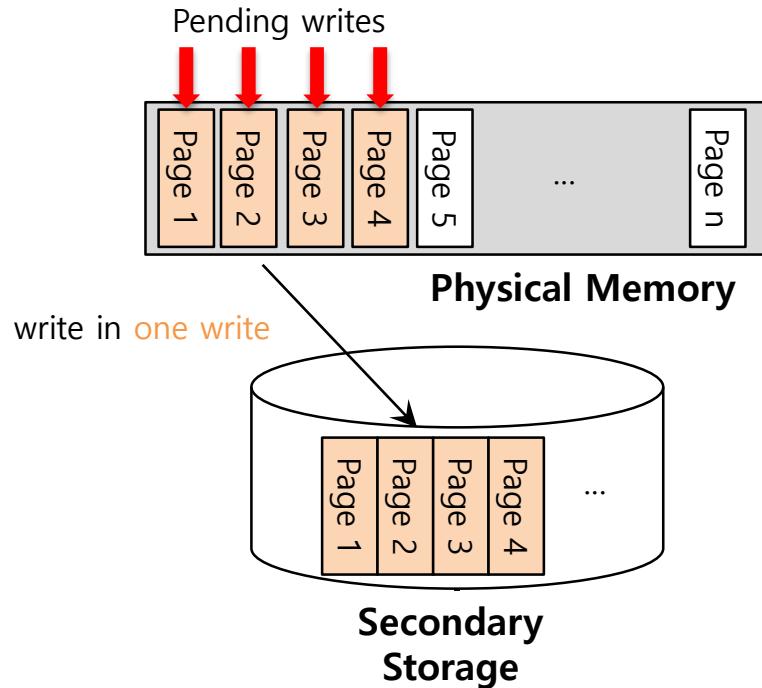
- The OS guess that a page is about to be used, and thus bring it in ahead of time.



Page 2 likely **soon be accessed** and
thus should be brought into memory too

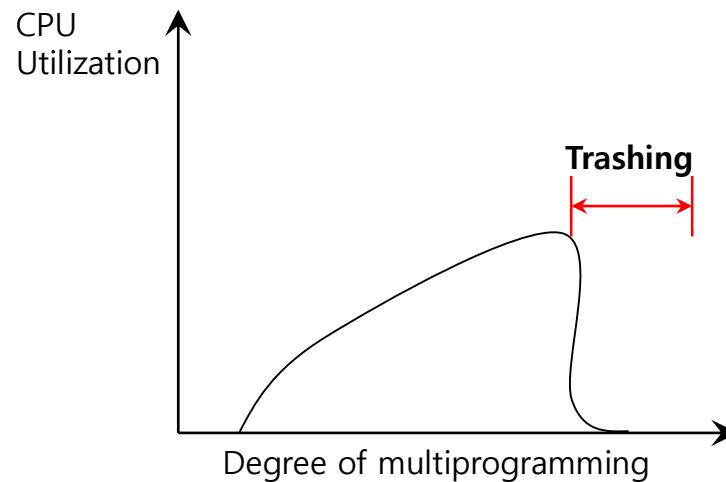
Clustering, Grouping

- Collect a number of **pending writes** together in memory and write them to disk in **one write**.
 - Perform a **single large write** more efficiently than **many small ones**.



Thrashing

- Memory is **oversubscribed** and the memory demands of the set of running processes **exceeds** the available physical memory.
 - Decide not to run a subset of processes.
 - Reduced set of processes working sets fit in memory.



26. Concurrency: An Introduction

Thread

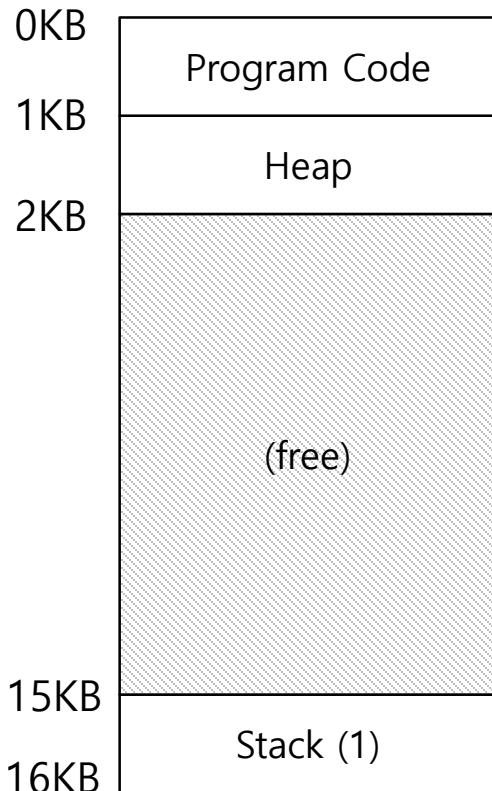
- ▣ A new abstraction for a single running process
- ▣ Multi-threaded program
 - ◆ A multi-threaded program has more than one point of execution.
 - ◆ Multiple PCs (Program Counter)
 - ◆ They **share** the share the same **address space**.

Context switch between threads

- ▣ Each thread has its own program counter and set of registers.
 - ◆ One or more **thread control blocks(TCBs)** are needed to store the state of each thread.
- ▣ When switching from running one (T1) to running the other (T2),
 - ◆ The register state of T1 be saved.
 - ◆ The register state of T2 restored.
 - ◆ The **address space remains** the same.

The stack of the relevant thread

- There will be one stack per thread.

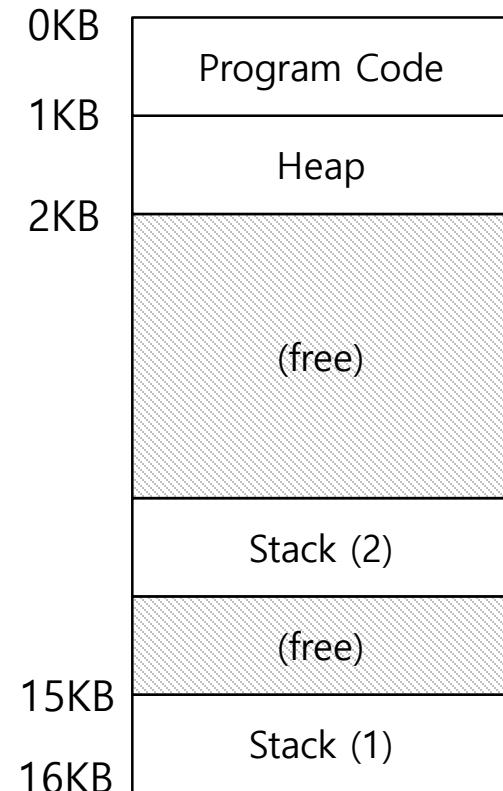


A Single-Threaded Address Space

The code segment:
where instructions live

The heap segment:
contains malloc'd data
dynamic data structures
(it grows downward)

(it grows upward)
The stack segment:
contains local variables
arguments to routines,
return values, etc.



Two threaded Address Space

Race condition

- Example with two threads

- counter = counter + 1 (default is 50)
- We expect the result is 52. However,

OS	Thread1	Thread2	(after instruction)		
			PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add \$0x1, %eax		108	51	50
interrupt					
	save T1's state				
	restore T2's state		100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add \$0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
	save T2's state				
	restore T1's state		108	51	50
		mov %eax, 0x8049a1c	113	51	51

Critical section

- ▣ A piece of code that **accesses a shared variable** and must not be concurrently executed by more than one thread.
 - ◆ Multiple threads executing critical section can result in a race condition.
 - ◆ Need to support **atomicity** for critical sections (**mutual exclusion**)

Locks

- Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).

```
1  lock_t mutex;
2  . . .
3  lock(&mutex);
4  balance = balance + 1; → Critical section
5  unlock(&mutex);
```

27. Interlude: Thread API

Thread Creation

▣ How to create and control threads?

```
#include <pthread.h>

int
pthread_create(      pthread_t*      thread,
                     const pthread_attr_t* attr,
                     void*                  (*start_routine)(void*),
                     void*                  arg);
```

- ◆ **thread**: Used to interact with this thread.
- ◆ **attr**: Used to specify any attributes this thread might have.
 - Stack size, Scheduling priority, ...
- ◆ **start_routine**: the function this thread start running in.
- ◆ **arg**: the argument to be passed to the function (start routine)
 - *a void pointer* allows us to pass in *any type of* argument.

Thread Creation (Cont.)

- If `start_routine` instead required another type argument, the declaration would look like this:
 - ◆ An integer argument:

```
int
pthread_create(..., // first two args are the same
               void* (*start_routine)(int),
               int      arg);
```

- ◆ Return an integer:

```
int
pthread_create(..., // first two args are the same
               int   (*start_routine)(void*),
               void*    arg);
```

Example: Creating a Thread

```
#include <pthread.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Wait for a thread to complete

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- ◆ thread: Specify which thread *to wait for*
- ◆ value_ptr: A pointer to the return value
 - Because pthread_join() routine changes the value, you need to **pass in a pointer** to that value.

Example: Waiting for Thread Completion

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <assert.h>
4 #include <stdlib.h>
5
6 typedef struct __myarg_t {
7     int a;
8     int b;
9 } myarg_t;
10
11 typedef struct __myret_t {
12     int x;
13     int y;
14 } myret_t;
15
16 void *mythread(void *arg) {
17     myarg_t *m = (myarg_t *) arg;
18     printf("%d %d\n", m->a, m->b);
19     myret_t *r = malloc(sizeof(myret_t));
20     r->x = 1;
21     r->y = 2;
22     return (void *) r;
23 }
24
```

Example: Waiting for Thread Completion (Cont.)

```
25 int main(int argc, char *argv[]) {
26     int rc;
27     pthread_t p;
28     myret_t *m;
29
30     myarg_t args;
31     args.a = 10;
32     args.b = 20;
33     pthread_create(&p, NULL, mythread, &args);
34     pthread_join(p, (void **) &m); // this thread has been
                                    // waiting inside of the
                                    // pthread_join() routine.
35     printf("returned %d %d\n", m->x, m->y);
36     return 0;
37 }
```

Example: Dangerous code

- Be careful with how values are returned from a thread.

```
1 void *mythread(void *arg) {  
2     myarg_t *m = (myarg_t *) arg;  
3     printf("%d %d\n", m->a, m->b);  
4     myret_t r; // ALLOCATED ON STACK: BAD!  
5     r.x = 1;  
6     r.y = 2;  
7     return (void *) &r;  
8 }
```

- When the variable `r` returns, it is automatically **de-allocated**.

Example: Simpler Argument Passing to a Thread

- Just passing in a single value

```
1 void *mythread(void *arg) {
2     int m = (int) arg;
3     printf("%d\n", m);
4     return (void *) (arg + 1);
5 }
6
7 int main(int argc, char *argv[]) {
8     pthread_t p;
9     int rc, m;
10    pthread_create(&p, NULL, mythread, (void *) 100);
11    pthread_join(p, (void **) &m);
12    printf("returned %d\n", m);
13    return 0;
14 }
```

Locks

- Provide mutual exclusion to a critical section

- ◆ Interface

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- ◆ Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;  
pthread_mutex_lock(&lock);  
x = x + 1; // or whatever your critical section is  
pthread_mutex_unlock(&lock);
```

- No other thread holds the lock → the thread will acquire the lock and enter the critical section.
 - If another thread hold the lock → the thread will not return from the call until it has acquired the lock.

Locks (Cont.)

- All locks must be properly initialized.
 - ◆ One way: using PTHREAD_MUTEX_INITIALIZER

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- ◆ The dynamic way: using pthread_mutex_init()

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Locks (Cont.)

- Check errors code when calling lock and unlock
 - ◆ An example wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

- These two calls are used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

- ◆ trylock: return failure if the lock is already held
- ◆ timelock: return after a timeout

Locks (Cont.)

- These two calls are also used in **lock acquisition**

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_timelock(pthread_mutex_t *mutex,  
                           struct timespec *abs_timeout);
```

- trylock: return failure if the lock is already held
- timelock: return after a timeout or after acquiring the lock

Condition Variables

- Condition variables are useful when some kind of **signaling** must take place between threads.

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);  
int pthread_cond_signal(pthread_cond_t *cond);
```

- ◆ `pthread_cond_wait`:
 - Put the calling thread to sleep.
 - Wait for some other thread to signal it.
- ◆ `pthread_cond_signal`:
 - Unblock at least one of the threads that are blocked on the condition variable

Condition Variables (Cont.)

- ❑ A thread calling wait routine:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t init = PTHREAD_COND_INITIALIZER;  
  
pthread_mutex_lock(&lock);  
while (initialized == 0)  
    pthread_cond_wait(&init, &lock);  
pthread_mutex_unlock(&lock);
```

- ◆ The wait call **releases the lock** when putting said caller to sleep.
- ◆ Before returning after being woken, the wait call **re-acquire the lock**.

- ❑ A thread calling signal routine:

```
pthread_mutex_lock(&lock);  
initialized = 1;  
pthread_cond_signal(&init);  
pthread_mutex_unlock(&lock);
```

Condition Variables (Cont.)

- The waiting thread **re-checks** the condition **in a while loop**, instead of a simple if statement.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t init = PTHREAD_COND_INITIALIZER;  
  
pthread_mutex_lock(&lock);  
while (initialized == 0)  
    pthread_cond_wait(&init, &lock);  
pthread_mutex_unlock(&lock);
```

- Without rechecking, the waiting thread will continue thinking that the condition has changed even though it has not.

Condition Variables (Cont.)

- ❑ Don't ever do this.
 - ◆ A thread calling wait routine:

```
while(initialized == 0)  
    ; // spin
```

- ◆ A thread calling signal routine:

```
initialized = 1;
```

- ◆ It performs poorly in many cases. → just wastes CPU cycles.
- ◆ It is error prone.

Compiling and Running

- ❑ To compile them, you must include the header `pthread.h`
 - ◆ Explicitly link with the **pthreads library**, by adding the `-pthread` flag.

```
prompt> gcc -o main main.c -Wall -pthread
```

- ◆ For more information,

```
man -k pthread
```

28. Locks

Locks: The Basic Idea

- ❑ Ensure that any **critical section** executes as if it were a single atomic instruction.

- ◆ An example: the canonical update of a shared variable

```
balance = balance + 1;
```

- ◆ Add some code around the critical section

```
1  lock_t mutex; // some globally-allocated lock 'mutex'  
2  ...  
3  lock(&mutex);  
4  balance = balance + 1;  
5  unlock(&mutex);
```

Locks: The Basic Idea

- ▣ Lock variable holds the state of the lock.
 - ◆ **available** (or **unlocked** or **free**)
 - No thread holds the lock.
 - ◆ **acquired** (or **locked** or **held**)
 - Exactly one thread holds the lock and presumably is in a critical section.

The semantics of the lock()

- ▣ lock()
 - ◆ Try to acquire the lock.
 - ◆ If no other thread holds the lock, the thread will **acquire** the lock.
 - ◆ **Enter** the *critical section*.
 - This thread is said to be the owner of the lock.
 - ◆ Other threads are *prevented from* entering the critical section while the first thread that holds the lock is in there.

Pthread Locks – mutex

- The name that the POSIX library uses for a lock.

- ◆ Used to provide **mutual exclusion** between threads.

```
1  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
4  balance = balance + 1;
5  Pthread_mutex_unlock(&lock);
```

- ◆ We may be using *different locks* to protect *different variables* → Increase **concurrency** (a more **fine-grained** approach).

Building A Lock

- ▣ Efficient locks provided mutual exclusion at **low cost**.
- ▣ Building a lock need some help from the **hardware** and the **OS**.

Evaluating locks – Basic criteria

□ Mutual exclusion

- ◆ Does the lock work, preventing multiple threads from entering *a critical section*?

□ Fairness

- ◆ Does each thread contending for the lock get a fair shot at acquiring it once it is free? (Starvation)

□ Performance

- ◆ The time overheads added by using the lock

Controlling Interrupts

▫ Disable Interrupts for critical sections

- ◆ One of the earliest solutions used to provide mutual exclusion
- ◆ Invented for single-processor systems.

```
1 void lock() {  
2     DisableInterrupts();  
3 }  
4 void unlock() {  
5     EnableInterrupts();  
6 }
```

- ◆ Problem:
 - Require too much *trust* in applications
 - Greedy (or malicious) program could monopolize the processor.
 - Do not work on **multiprocessors**
 - Code that masks or unmasks interrupts be executed *slowly* by modern CPUs

Why hardware support needed?

- ▣ **First attempt:** Using a *flag* denoting whether the lock is held or not.
 - ◆ The code below has problems.

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 → lock is available, 1 → held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1) // TEST the flag
10         ; // spin-wait (do nothing)
11      mutex->flag = 1; // now SET it !
12  }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

Why hardware support needed? (Cont.)

- ◆ **Problem 1:** No Mutual Exclusion (assume flag=0 to begin)

Thread1

```
call lock()
while (flag == 1)
interrupt: switch to Thread 2
```

flag = 1; // set flag to 1 (too!)

Thread2

```
call lock()
while (flag == 1)
flag = 1;
interrupt: switch to Thread 1
```

- ◆ **Problem 2:** Spin-waiting wastes time waiting for another thread.
- ▣ So, we need an atomic instruction supported by **Hardware!**
 - ◆ *test-and-set* instruction, also known as *atomic exchange*

Test And Set (Atomic Exchange)

- An instruction to support the creation of simple locks

```
1 int TestAndSet(int *ptr, int new) {  
2     int old = *ptr;    // fetch old value at ptr  
3     *ptr = new;        // store 'new' into ptr  
4     return old;        // return the old value  
5 }
```

- ◆ **return**(testing) old value pointed to by the `ptr`.
- ◆ *Simultaneously update*(setting) said value to `new`.
- ◆ This sequence of operations is **performed atomically**.

A Simple Spin Lock using test-and-set

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 indicates that lock is available,
7      // 1 that it is held
8      lock->flag = 0;
9  }
10
11 void lock(lock_t *lock) {
12     while (TestAndSet(&lock->flag, 1) == 1)
13         ;           // spin-wait
14 }
15
16 void unlock(lock_t *lock) {
17     lock->flag = 0;
18 }
```

- ◆ **Note:** To work correctly on *a single processor*, it requires a preemptive scheduler.

Evaluating Spin Locks

- **Correctness:** yes

- ◆ The spin lock only allows a single thread to entry the critical section.

- **Fairness:** no

- ◆ Spin locks don't provide any fairness guarantees.
 - ◆ Indeed, a thread spinning may spin *forever*.

- **Performance:**

- ◆ In the single CPU, performance overheads can be quite *painful*.
 - ◆ If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*.

Compare-And-Swap

- Test whether the value at the address(ptr) is equal to expected.
 - If so, update the memory location pointed to by ptr with the new value.
 - In either case, return the actual value at that memory location.

```
1 int CompareAndSwap(int *ptr, int expected, int new) {  
2     int actual = *ptr;  
3     if (actual == expected)  
4         *ptr = new;  
5     return actual;  
6 }
```

Compare-and-Swap hardware atomic instruction (C-style)

```
1 void lock(lock_t *lock) {  
2     while (!CompareAndSwap(&lock->flag, 0, 1))  
3         ; // spin  
4 }
```

Spin lock with compare-and-swap

Compare-And-Swap (Cont.)

▣ C-callable x86-version of compare-and-swap

```
1  char CompareAndSwap(int *ptr, int old, int new) {
2      unsigned char ret;
3
4      // Note that sete sets a 'byte' not the word
5      __asm__ __volatile__ (
6          " lock\n"
7          " cmpxchgl %2,%1\n"
8          " sete %0\n"
9          : "=q" (ret), "=m" (*ptr)
10         : "r" (new), "m" (*ptr), "a" (old)
11         : "memory");
12
13 }
```

Load-Linked and Store-Conditional

```
1 int LoadLinked(int *ptr) {
2     return *ptr;
3 }
4
5 int StoreConditional(int *ptr, int value) {
6     if (no one has updated *ptr since the LoadLinked to this address) {
7         *ptr = value;
8         return 1; // success!
9     } else {
10        return 0; // failed to update
11    }
12 }
```

Load-linked And Store-conditional

- ◆ The store-conditional *only succeeds* if **no intermittent store** to the address has taken place.
 - **success:** return 1 and update the value at `ptr` to `value`.
 - **fail:** the value at `ptr` is not updates and 0 is returned.

Load-Linked and Store-Conditional (Cont.)

```
1 void lock(lock_t *lock) {
2     while (1) {
3         while (LoadLinked(&lock->flag) == 1)
4             ; // spin until it's zero
5         if (StoreConditional(&lock->flag, 1) == 1)
6             return; // if set-it-to-1 was a success: all done
7         otherwise: try it all over again
8     }
9 }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

Using LL/SC To Build A Lock

```
1 void lock(lock_t *lock) {
2     while (LoadLinked(&lock->flag) || !StoreConditional(&lock->flag, 1))
3         ; // spin
4 }
```

A more concise form of the lock() using LL/SC

Fetch-And-Add

- Atomically increment a value while returning the old value at a particular address.

```
1 int FetchAndAdd(int *ptr) {  
2     int old = *ptr;  
3     *ptr = old + 1;  
4     return old;  
5 }
```

Fetch-And-Add Hardware atomic instruction (C-style)

Ticket Lock

- ▣ **Ticket lock** can be built with fetch-and add.
 - ◆ Ensure progress for all threads. → **fairness**

```
1  typedef struct __lock_t {  
2      int ticket;  
3      int turn;  
4  } lock_t;  
5  
6  void lock_init(lock_t *lock) {  
7      lock->ticket = 0;  
8      lock->turn = 0;  
9  }  
10  
11 void lock(lock_t *lock) {  
12     int myturn = FetchAndAdd(&lock->ticket);  
13     while (lock->turn != myturn)  
14         ; // spin  
15 }  
16 void unlock(lock_t *lock) {  
17     FetchAndAdd(&lock->turn);  
18 }
```

So Much Spinning

- ❑ Hardware-based spin locks are **simple** and they work.
- ❑ In some cases, these solutions can be quite **inefficient**.
 - ◆ Any time a thread gets caught *spinning*, it **wastes an entire time slice** doing nothing but checking a value.

How To Avoid *Spinning*?
We'll need OS Support too!

A Simple Approach: Just Yield

- When you are going to spin, **give up the CPU** to another thread.
 - OS system call moves the caller from the *running state* to the *ready state*.
 - The cost of a **context switch** can be substantial and the **starvation** problem still exists.

```
1 void init() {
2     flag = 0;
3 }
4
5 void lock() {
6     while (TestAndSet(&flag, 1) == 1)
7         yield(); // give up the CPU
8 }
9
10 void unlock() {
11     flag = 0;
12 }
```

Lock with Test-and-set and Yield

Using Queues: Sleeping Instead of Spinning

- ▣ **Queue** to keep track of which threads are waiting to enter the lock.
- ▣ park()
 - ◆ Put a calling thread to sleep
- ▣ unpark(threadID)
 - ◆ Wake a particular thread as designated by threadID.

Using Queues: Sleeping Instead of Spinning

```
1  typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3  void lock_init(lock_t *m) {
4      m->flag = 0;
5      m->guard = 0;
6      queue_init(m->q);
7  }
8
9  void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
11         ; // acquire guard lock by spinning
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }
21 ...
```

Lock With Queues, Test-and-set, Yield, And Wakeup

Using Queues: Sleeping Instead of Spinning

```
22 void unlock(lock_t *m) {  
23     while (TestAndSet(&m->guard, 1) == 1)  
24         ; // acquire guard lock by spinning  
25     if (queue_empty(m->q))  
26         m->flag = 0; // let go of lock; no one wants it  
27     else  
28         unpark(queue_remove(m->q)); // hold lock (for next thread!)  
29     m->guard = 0;  
30 }
```

Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

Wakeup/waiting race

- In case of releasing the lock (*thread A*) just before the call to `park()` (*thread B*) → Thread B would **sleep forever** (potentially).
- **Solaris** solves this problem by adding a third system call: `setpark()`.
 - ◆ By calling this routine, a thread can indicate it *is about to park*.
 - ◆ If it happens to be interrupted and another thread calls `unpark` before `park` is actually called, the subsequent `park` returns immediately instead of sleeping.

```
1           queue_add(m->q, gettid());
2           setpark(); // new code
3           m->guard = 0;
4           park();
```

Code modification inside of `lock()`

Futex

- ▣ Linux provides a **futex** (is similar to Solaris's park and unpark).
 - ◆ `futex_wait(address, expected)`
 - Put the calling thread to sleep
 - If the value at address is not equal to expected, the call returns immediately.
 - ◆ `futex_wake(address)`
 - Wake one thread that is waiting on the queue.

Futex (Cont.)

- Snippet from `lowlevellock.h` in the `nptl` library

- The high bit of the integer `v`: track whether the lock is held or not
- All the other bits : the number of waiters

```
1 void mutex_lock(int *mutex) {
2     int v;
3     /* Bit 31 was clear, we got the mutex (this is the fastpath) */
4     if (atomic_bit_test_set(mutex, 31) == 0)
5         return;
6     atomic_increment(mutex);
7     while (1) {
8         if (atomic_bit_test_set(mutex, 31) == 0) {
9             atomic_decrement(mutex);
10            return;
11        }
12        /* We have to wait now. First make sure the futex value
13           we are monitoring is truly negative (i.e. locked). */
14        v = *mutex;
15        ...
```

Linux-based Futex Locks

Futex (Cont.)

```
16             if (v >= 0)
17                 continue;
18             futex_wait(mutex, v);
19         }
20     }
21
22 void mutex_unlock(int *mutex) {
23     /* Adding 0x80000000 to the counter results in 0 if and only if
24      there are not other interested threads */
25     if (atomic_add_zero(mutex, 0x80000000))
26         return;
27     /* There are other threads waiting for this mutex,
28      wake one of them up */
29     futex_wake(mutex);
30 }
```

Linux-based Futex Locks (Cont.)

Two-Phase Locks

- ▣ A two-phase lock realizes that **spinning can be useful** if the lock *is about to be released*.
 - ◆ **First phase**
 - The lock spins for a while, *hoping that* it can acquire the lock.
 - If the lock is not acquired during the first spin phase, a second phase is entered,
 - ◆ **Second phase**
 - The caller is put to sleep.
 - The caller is only woken up when the lock becomes free later.

29. Lock-based Concurrent Data Structures

Lock-based Concurrent Data structure

- ▣ Adding locks to a data structure makes the structure **thread safe**.
 - ◆ How locks are added determine both the **correctness** and **performance** of the data structure.

Example: Concurrent Counters without Locks

- Simple but not scalable

```
1  typedef struct __counter_t {
2      int value;
3  } counter_t;
4
5  void init(counter_t *c) {
6      c->value = 0;
7  }
8
9  void increment(counter_t *c) {
10     c->value++;
11 }
12
13 void decrement(counter_t *c) {
14     c->value--;
15 }
16
17 int get(counter_t *c) {
18     return c->value;
19 }
```

Example: Concurrent Counters with Locks

- Add a **single lock**.

- ◆ The lock is acquired when calling a routine that manipulates the data structure.

```
1  typedef struct __counter_t {
2      int value;
3      pthread_lock_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
```

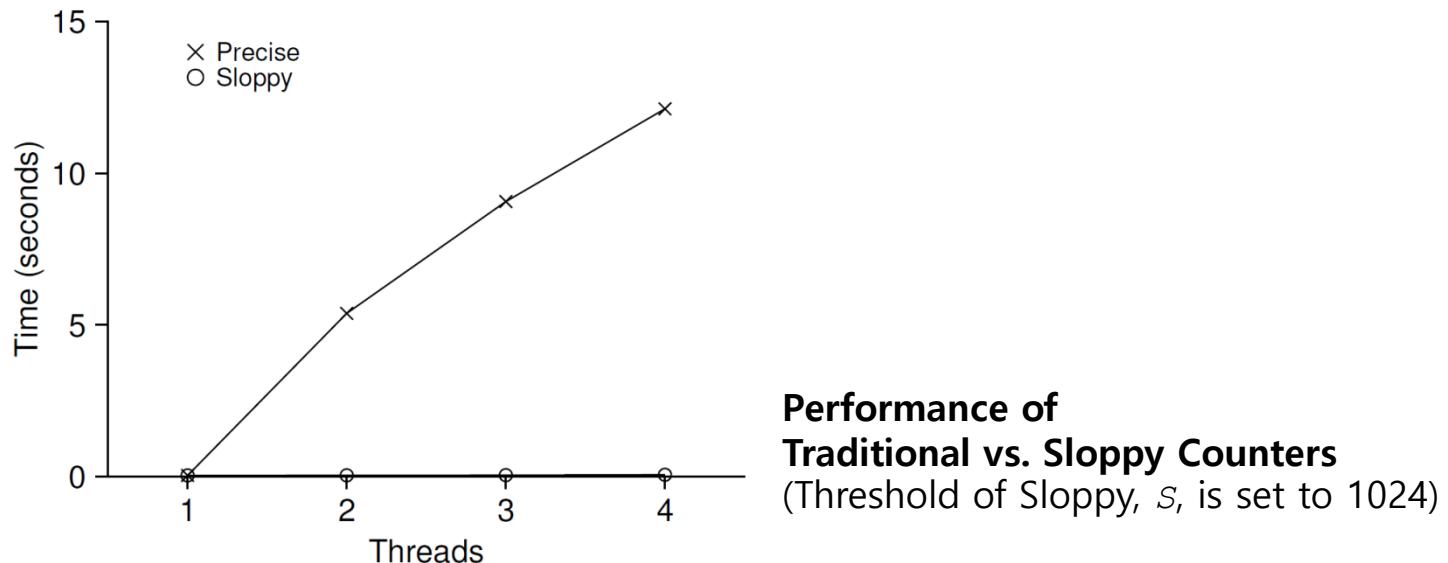
Example: Concurrent Counters with Locks (Cont.)

(Cont.)

```
17     void decrement(counter_t *c) {
18         Pthread_mutex_lock(&c->lock);
19         c->value--;
20         Pthread_mutex_unlock(&c->lock);
21     }
22
23     int get(counter_t *c) {
24         Pthread_mutex_lock(&c->lock);
25         int rc = c->value;
26         Pthread_mutex_unlock(&c->lock);
27         return rc;
28     }
```

The performance costs of the simple approach

- Each thread updates a single shared counter.
 - Each thread updates the counter one million times.
 - iMac with four Intel 2.7GHz i5 CPUs.



Synchronized counter scales poorly.

Perfect Scaling

- ❑ Even though more work is done, it is **done in parallel**.
- ❑ The time taken to complete the task is *not increased*.

Sloppy counter

- ▣ The sloppy counter works by representing ...
 - ◆ A single **logical counter** via numerous local physical counters, on per CPU core
 - ◆ A single **global counter**
 - ◆ There are **locks**:
 - One for each local counter and one for the global counter
- ▣ Example: on a machine with four CPUs
 - ◆ Four local counters
 - ◆ One global counter

The basic idea of sloppy counting

- ▣ When a thread running on a core wishes to increment the counter.
 - ◆ It increments its local counter.
 - ◆ Each CPU has its own local counter:
 - Threads across CPUs can update local counters *without contention*.
 - Thus counter updates are **scalable**.
 - ◆ The local values are periodically transferred to the global counter.
 - Acquire the global lock
 - Increment it by the local counter's value
 - The local counter is then reset to zero.

The basic idea of sloppy counting (Cont.)

- ▣ How often the local-to-global transfer occurs is determined by a threshold, S (sloppiness).
 - ◆ The smaller S :
 - The more the counter behaves like the *non-scalable counter*.
 - ◆ The bigger S :
 - The more scalable the counter.
 - The further off the global value might be from the *actual count*.

Sloppy counter example

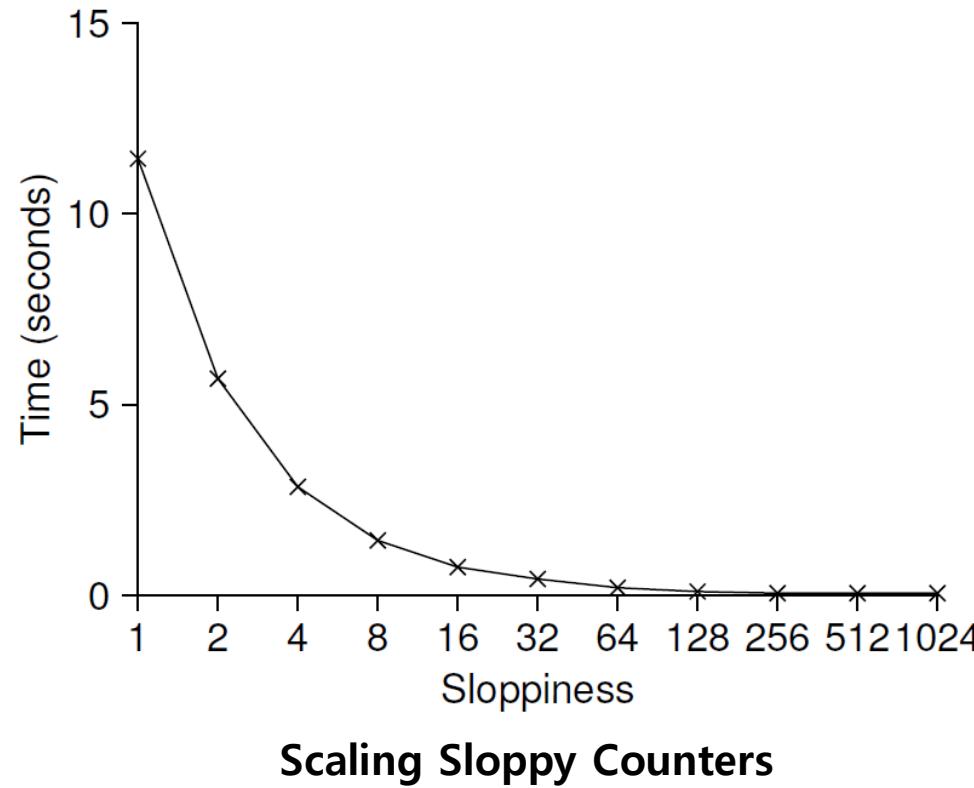
□ Tracing the Sloppy Counters

- ◆ The threshold S is set to 5.
- ◆ There are threads on each of four CPUs
- ◆ Each thread updates their local counters $L_1 \dots L_4$.

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

Importance of the threshold value S

- Each four threads increments a counter 1 million times on four CPUs.
 - Low $S \rightarrow$ Performance is **poor**, The global count is always quite **accurate**.
 - High $S \rightarrow$ Performance is **excellent**, The global count **lags**.



Sloppy Counter Implementation

```
1  typedef struct __counter_t {
2      int global;           // global count
3      pthread_mutex_t glock; // global lock
4      int local[NUMCPUS];   // local count (per cpu)
5      pthread_mutex_t llock[NUMCPUS]; // ... and locks
6      int threshold;        // update frequency
7  } counter_t;
8
9  // init: record threshold, init locks, init values
10 //       of all local counts and global count
11 void init(counter_t *c, int threshold) {
12     c->threshold = threshold;
13
14     c->global = 0;
15     pthread_mutex_init(&c->glock, NULL);
16
17     int i;
18     for (i = 0; i < NUMCPUS; i++) {
19         c->local[i] = 0;
20         pthread_mutex_init(&c->llock[i], NULL);
21     }
22 }
23 }
```

Sloppy Counter Implementation (Cont.)

(Cont.)

```
24     // update: usually, just grab local lock and update local amount
25     //           once local count has risen by 'threshold', grab global
26     //           lock and transfer local values to it
27     void update(counter_t *c, int threadID, int amt) {
28         pthread_mutex_lock(&c->llock[threadID]);
29         c->local[threadID] += amt;          // assumes amt > 0
30         if (c->local[threadID] >= c->threshold) { // transfer to global
31             pthread_mutex_lock(&c->glock);
32             c->global += c->local[threadID];
33             pthread_mutex_unlock(&c->glock);
34             c->local[threadID] = 0;
35         }
36         pthread_mutex_unlock(&c->llock[threadID]);
37     }
38
39     // get: just return global amount (which may not be perfect)
40     int get(counter_t *c) {
41         pthread_mutex_lock(&c->glock);
42         int val = c->global;
43         pthread_mutex_unlock(&c->glock);
44         return val;    // only approximate!
45     }
```

Concurrent Linked Lists

```
1 // basic node structure
2 typedef struct __node_t {
3     int key;
4     struct __node_t *next;
5 } node_t;
6
7 // basic list structure (one used per list)
8 typedef struct __list_t {
9     node_t *head;
10    pthread_mutex_t lock;
11 } list_t;
12
13 void List_Init(list_t *L) {
14     L->head = NULL;
15     pthread_mutex_init(&L->lock, NULL);
16 }
17
18 int List_Insert(list_t *L, int key) {
19     pthread_mutex_lock(&L->lock);
20     node_t *new = malloc(sizeof(node_t));
21     if (new == NULL) {
22         perror("malloc");
23         pthread_mutex_unlock(&L->lock);
24         return -1; // fail
25 }
```

Concurrent Linked Lists (Cont.)

```
(Cont.)  
26             new->key = key;  
27             new->next = L->head;  
28             L->head = new;  
29             pthread_mutex_unlock(&L->lock);  
30             return 0; // success  
31     }  
32  
33     int List_Lookup(list_t *L, int key) {  
34         pthread_mutex_lock(&L->lock);  
35         node_t *curr = L->head;  
36         while (curr) {  
37             if (curr->key == key) {  
38                 pthread_mutex_unlock(&L->lock);  
39                 return 0; // success  
40             }  
41             curr = curr->next;  
42         }  
43         pthread_mutex_unlock(&L->lock);  
44         return -1; // failure  
45     }
```

Concurrent Linked Lists (Cont.)

- ▣ The code **acquires** a lock in the insert routine upon entry.
- ▣ The code **releases** the lock upon exit.
 - ◆ If `malloc()` happens to *fail*, the code must also release the lock before failing the insert.
 - ◆ This kind of exceptional control flow has been shown to be **quite error prone**.
 - ◆ **Solution:** The lock and release *only surround* the actual critical section in the insert code

Concurrent Linked List: Rewritten

```
1 void List_Init(list_t *L) {
2     L->head = NULL;
3     pthread_mutex_init(&L->lock, NULL);
4 }
5
6 void List_Insert(list_t *L, int key) {
7     // synchronization not needed
8     node_t *new = malloc(sizeof(node_t));
9     if (new == NULL) {
10         perror("malloc");
11         return;
12     }
13     new->key = key;
14
15     // just lock critical section
16     pthread_mutex_lock(&L->lock);
17     new->next = L->head;
18     L->head = new;
19     pthread_mutex_unlock(&L->lock);
20 }
21
```

Concurrent Linked List: Rewritten (Cont.)

(Cont.)

```
22     int List_Lookup(list_t *L, int key) {
23         int rv = -1;
24         pthread_mutex_lock(&L->lock);
25         node_t *curr = L->head;
26         while (curr) {
27             if (curr->key == key) {
28                 rv = 0;
29                 break;
30             }
31             curr = curr->next;
32         }
33         pthread_mutex_unlock(&L->lock);
34         return rv; // now both success and failure
35     }
```

Scaling Linked List

- ▣ Hand-over-hand locking (lock coupling)
 - ◆ Add **a lock per node** of the list instead of having a single lock for the entire list.
 - ◆ When traversing the list,
 - First grabs the next node's lock.
 - And then releases the current node's lock.
 - ◆ Enable a high degree of concurrency in list operations.
 - However, in practice, the overheads of acquiring and releasing locks for each node of a list traversal is *prohibitive*.

Michael and Scott Concurrent Queues

- ▣ There are two locks.
 - ◆ One for the **head** of the queue.
 - ◆ One for the **tail**.
 - ◆ The goal of these two locks is to enable concurrency of *enqueue* and *dequeue* operations.
- ▣ Add a dummy node
 - ◆ Allocated in the queue initialization code
 - ◆ Enable the separation of head and tail operations

Concurrent Queues (Cont.)

```
1     typedef struct __node_t {
2         int value;
3         struct __node_t *next;
4     } node_t;
5
6     typedef struct __queue_t {
7         node_t *head;
8         node_t *tail;
9         pthread_mutex_t headLock;
10        pthread_mutex_t tailLock;
11    } queue_t;
12
13    void Queue_Init(queue_t *q) {
14        node_t *tmp = malloc(sizeof(node_t));
15        tmp->next = NULL;
16        q->head = q->tail = tmp;
17        pthread_mutex_init(&q->headLock, NULL);
18        pthread_mutex_init(&q->tailLock, NULL);
19    }
20
21    void Queue_Enqueue(queue_t *q, int value) {
22        node_t *tmp = malloc(sizeof(node_t));
23        assert(tmp != NULL);
24        tmp->value = value;
```

Concurrent Queues (Cont.)

(Cont.)

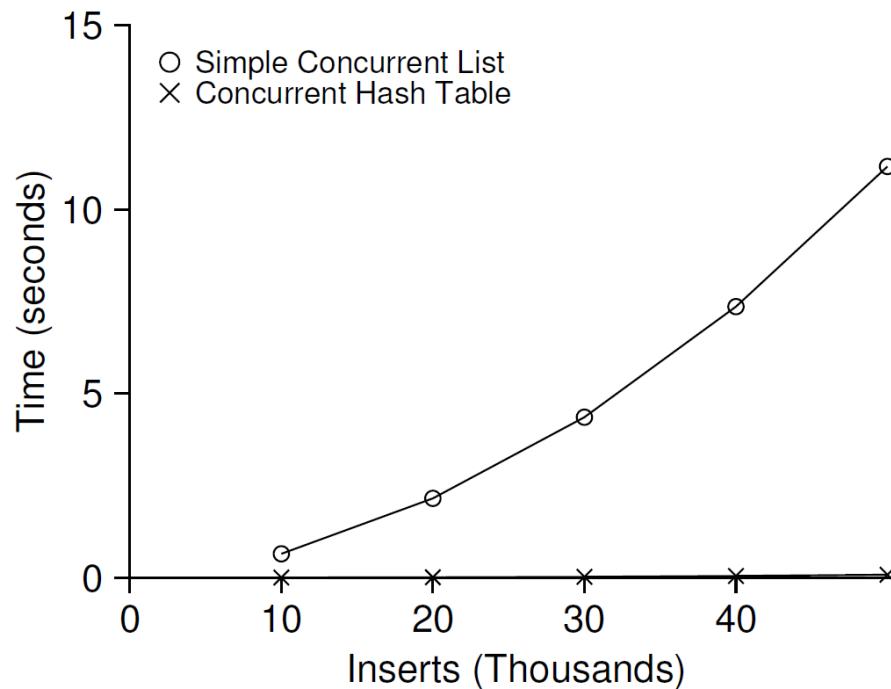
```
25         tmp->next = NULL;
26
27         pthread_mutex_lock(&q->tailLock);
28         q->tail->next = tmp;
29         q->tail = tmp;
30         pthread_mutex_unlock(&q->tailLock);
31     }
32
33     int Queue_Dequeue(queue_t *q, int *value) {
34         pthread_mutex_lock(&q->headLock);
35         node_t *tmp = q->head;
36         node_t *newHead = tmp->next;
37         if (newHead == NULL) {
38             pthread_mutex_unlock(&q->headLock);
39             return -1; // queue was empty
40         }
41         *value = newHead->value;
42         q->head = newHead;
43         pthread_mutex_unlock(&q->headLock);
44         free(tmp);
45         return 0;
46     }
```

Concurrent Hash Table

- ▣ Focus on a simple hash table
 - ◆ The hash table does not resize.
 - ◆ Built using the concurrent lists
 - ◆ It uses a **lock per hash bucket** each of which is represented by *a list*.

Performance of Concurrent Hash Table

- From 10,000 to 50,000 concurrent updates from each of four threads.
 - iMac with four Intel 2.7GHz i5 CPUs.



The simple concurrent hash table scales magnificently.

Concurrent Hash Table

```
1      #define BUCKETS (101)
2
3      typedef struct __hash_t {
4          list_t lists[BUCKETS];
5      } hash_t;
6
7      void Hash_Init(hash_t *H) {
8          int i;
9          for (i = 0; i < BUCKETS; i++) {
10              List_Init(&H->lists[i]);
11          }
12      }
13
14      int Hash_Insert(hash_t *H, int key) {
15          int bucket = key % BUCKETS;
16          return List_Insert(&H->lists[bucket], key);
17      }
18
19      int Hash_Lookup(hash_t *H, int key) {
20          int bucket = key % BUCKETS;
21          return List_Lookup(&H->lists[bucket], key);
22      }
```

30. Condition Variables

Condition Variables

- ▣ There are many cases where a thread wishes to check whether a **condition** is true before continuing its execution.
- ▣ Example:
 - ◆ A parent thread might wish to check whether a child thread has *completed*.
 - ◆ This is often called a `join()`.

Condition Variables (Cont.)

A Parent Waiting For Its Child

```
1      void *child(void *arg) {
2          printf("child\n");
3          // XXX how to indicate we are done?
4          return NULL;
5      }
6
7      int main(int argc, char *argv[]) {
8          printf("parent: begin\n");
9          pthread_t c;
10         Pthread_create(&c, NULL, child, NULL); // create child
11         // XXX how to wait for child?
12         printf("parent: end\n");
13         return 0;
14     }
```

What we would like to see here is:

```
parent: begin
child
parent: end
```

Parent waiting fore child: Spin-based Approach

```
1     volatile int done = 0;
2
3     void *child(void *arg) {
4         printf("child\n");
5         done = 1;
6         return NULL;
7     }
8
9     int main(int argc, char *argv[]) {
10        printf("parent: begin\n");
11        pthread_t c;
12        Pthread_create(&c, NULL, child, NULL); // create child
13        while (done == 0)
14            ; // spin
15        printf("parent: end\n");
16        return 0;
17    }
```

- ◆ This is hugely inefficient as the parent spins and **wastes CPU time**.

How to wait for a condition

- ▣ Condition variable
 - ◆ **Waiting** on the condition
 - An explicit queue that threads can put themselves on when some state of execution is not as desired.
 - ◆ **Signaling** on the condition
 - Some other thread, *when it changes said state*, can wake one of those waiting threads and allow them to continue.

Definition and Routines

❑ Declare condition variable

```
pthread_cond_t c;
```

- ◆ Proper initialization is required.

❑ Operation (the POSIX calls)

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);      // wait()
pthread_cond_signal(pthread_cond_t *c);                          // signal()
```

- ◆ The `wait()` call takes a mutex as a parameter.
 - The `wait()` call release the lock and put the calling thread to sleep.
 - When the thread wakes up, it must re-acquire the lock.

Parent waiting for Child: Use a condition variable

```
1      int done = 0;
2      pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3      pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5      void thr_exit() {
6          Pthread_mutex_lock(&m);
7          done = 1;
8          Pthread_cond_signal(&c);
9          Pthread_mutex_unlock(&m);
10     }
11
12     void *child(void *arg) {
13         printf("child\n");
14         thr_exit();
15         return NULL;
16     }
17
18     void thr_join() {
19         Pthread_mutex_lock(&m);
20         while (done == 0)
21             Pthread_cond_wait(&c, &m);
22         Pthread_mutex_unlock(&m);
23     }
24 }
```

Parent waiting for Child: Use a condition variable

(cont.)

```
25     int main(int argc, char *argv[]) {  
26         printf("parent: begin\n");  
27         pthread_t p;  
28         Pthread_create(&p, NULL, child, NULL);  
29         thr_join();  
30         printf("parent: end\n");  
31         return 0;  
32     }
```

Parent waiting for Child: Use a condition variable

□ Parent:

- ◆ Create the child thread and continues running itself.
- ◆ Call into `thr_join()` to wait for the child thread to complete.
 - Acquire the lock
 - Check if the child is done
 - Put itself to sleep by calling `wait()`
 - Release the lock

□ Child:

- ◆ Print the message "child"
- ◆ Call `thr_exit()` to wake the parent thread
 - Grab the lock
 - Set the state variable `done`
 - Signal the parent thus waking it.

The importance of the state variable done

```
1     void thr_exit() {
2             Pthread_mutex_lock(&m);
3             Pthread_cond_signal(&c);
4             Pthread_mutex_unlock(&m);
5     }
6
7     void thr_join() {
8             Pthread_mutex_lock(&m);
9             Pthread_cond_wait(&c, &m);
10            Pthread_mutex_unlock(&m);
11    }
```

thr_exit() and thr_join() without variable done

- ◆ Imagine the case where the *child runs immediately*.
 - The child will signal, but there is no thread asleep on the condition.
 - When the parent runs, it will call wait and be stuck.
 - No thread will ever wake it.

Another poor implementation

```
1     void thr_exit() {
2             done = 1;
3             Pthread_cond_signal(&c);
4     }
5
6     void thr_join() {
7         if (done == 0)
8             Pthread_cond_wait(&c);
9 }
```

- ◆ The issue here is a subtle **race condition**.
 - The parent calls `thr_join()`.
 - The parent checks the value of `done`.
 - It will see that it is 0 and try to go to sleep.
 - *Just before* it calls `wait` to go to sleep, the parent is interrupted and the child runs.
 - The child changes the state variable `done` to 1 and signals.
 - But no thread is waiting and thus no thread is woken.
 - When the parent runs again, it sleeps forever.

The Producer / Consumer (Bound Buffer) Problem

□ Producer

- ◆ Produce data items
- ◆ Wish to place data items in a buffer

□ Consumer

- ◆ Grab data items out of the buffer consume them in some way

□ Example: Multi-threaded web server

- ◆ *A producer* puts HTTP requests in to a work queue
- ◆ *Consumer threads* take requests out of this queue and process them

Bounded buffer

- ▣ A bounded buffer is used when you pipe the output of one program into another.
 - ◆ Example: `grep foo file.txt | wc -l`
 - The grep process is the producer.
 - The wc process is the consumer.
 - Between them is an in-kernel bounded buffer.
 - ◆ Bounded buffer is Shared resource → **Synchronized access** is required.

The Put and Get Routines (Version 1)

```
1      int buffer;
2      int count = 0;    // initially, empty
3
4      void put(int value) {
5          assert(count == 0);
6          count = 1;
7          buffer = value;
8      }
9
10     int get() {
11         assert(count == 1);
12         count = 0;
13         return buffer;
14     }
```

- ◆ Only put data into the buffer when `count` is zero.
 - i.e., when the buffer is *empty*.
- ◆ Only get data from the buffer when `count` is one.
 - i.e., when the buffer is *full*.

Producer/Consumer Threads (Version 1)

```
1      void *producer(void *arg) {
2          int i;
3          int loops = (int) arg;
4          for (i = 0; i < loops; i++) {
5              put(i);
6          }
7      }
8
9      void *consumer(void *arg) {
10         int i;
11         while (1) {
12             int tmp = get();
13             printf("%d\n", tmp);
14         }
15     }
```

- ◆ **Producer** puts an integer into the shared buffer loops number of times.
- ◆ **Consumer** gets the data out of that shared buffer.

Producer/Consumer: Single CV and If Statement

- A single condition variable cond and associated lock mutex

```
1      cond_t cond;
2      mutex_t mutex;
3
4      void *producer(void *arg) {
5          int i;
6          for (i = 0; i < loops; i++) {
7              Pthread_mutex_lock(&mutex);                                // p1
8              if (count == 1)                                         // p2
9                  Pthread_cond_wait(&cond, &mutex);                      // p3
10             put(i);                                              // p4
11             Pthread_cond_signal(&cond);                            // p5
12             Pthread_mutex_unlock(&mutex);                          // p6
13         }
14     }
15
16     void *consumer(void *arg) {
17         int i;
18         for (i = 0; i < loops; i++) {
19             Pthread_mutex_lock(&mutex);                                // c1
```

Producer/Consumer: Single CV and If Statement

```
20         if (count == 0)                                // c2
21             Pthread_cond_wait(&cond, &mutex);          // c3
22         int tmp = get();                             // c4
23         Pthread_cond_signal(&cond);                // c5
24         Pthread_mutex_unlock(&mutex);              // c6
25         printf("%d\n", tmp);
26     }
27 }
```

- ◆ p1-p3: A producer waits for the buffer to be empty.
- ◆ c1-c3: A consumer waits for the buffer to be full.
- ◆ With just *a single producer* and *a single consumer*, the code works.

If we have **more than** one of producer and consumer?

Thread Trace: Broken Solution (Version 1)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
c4	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
	Running		Ready		Ready	0	Oh oh! No data

Thread Trace: Broken Solution (Version 1)

- ▣ The problem arises for a simple reason:
 - ◆ After the producer woke T_{c1} , but before T_{c1} ever ran, the state of the bounded buffer *changed by T_{c2}* .
 - ◆ There is no guarantee that when the woken thread runs, the state will still be as desired → Mesa semantics.
 - Virtually every system ever built employs *Mesa semantics*.
 - ◆ Hoare semantics provides a stronger guarantee that the woken thread will run immediately upon being woken.

Producer/Consumer: Single CV and While

- Consumer T_{c1} wakes up and **re-checks** the state of the shared variable.
 - If the buffer is empty, the consumer simply goes back to sleep.

```
1      cond_t cond;
2      mutex_t mutex;
3
4      void *producer(void *arg) {
5          int i;
6          for (i = 0; i < loops; i++) {
7              Pthread_mutex_lock(&mutex);                                // p1
8              while (count == 1)                                         // p2
9                  Pthread_cond_wait(&cond, &mutex);                      // p3
10             put(i);                                              // p4
11             Pthread_cond_signal(&cond);                            // p5
12             Pthread_mutex_unlock(&mutex);                          // p6
13         }
14     }
15 }
```

Producer/Consumer: Single CV and While

(Cont.)

```
16     void *consumer(void *arg) {
17         int i;
18         for (i = 0; i < loops; i++) {
19             Pthread_mutex_lock(&mutex);                                // c1
20             while (count == 0)                                       // c2
21                 Pthread_cond_wait(&cond, &mutex);                  // c3
22             int tmp = get();                                       // c4
23             Pthread_cond_signal(&cond);                           // c5
24             Pthread_mutex_unlock(&mutex);                         // c6
25             printf("%d\n", tmp);
26         }
27     }
```

- ◆ A simple rule to remember with condition variables is to **always use while loops.**
- ◆ However, this code still has a bug (*next page*).

Thread Trace: Broken Solution (Version 2)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}

Thread Trace: Broken Solution (Version 2) (Cont.)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
...	(cont.)
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep ...

- ◆ A consumer should not wake other consumers, only producers, and vice-versa.

The single Buffer Producer/Consumer Solution

- ❑ Use **two** condition variables and while
 - ◆ **Producer** threads wait on the condition `empty`, and signals `fill`.
 - ◆ **Consumer** threads wait on `fill` and signal `empty`.

```
1      cond_t empty, fill;
2      mutex_t mutex;
3
4      void *producer(void *arg) {
5          int i;
6          for (i = 0; i < loops; i++) {
7              Pthread_mutex_lock(&mutex);
8              while (count == 1)
9                  Pthread_cond_wait(&empty, &mutex);
10             put(i);
11             Pthread_cond_signal(&fill);
12             Pthread_mutex_unlock(&mutex);
13         }
14     }
15 }
```

The single Buffer Producer/Consumer Solution

(Cont.)

```
16     void *consumer(void *arg) {
17         int i;
18         for (i = 0; i < loops; i++) {
19             Pthread_mutex_lock(&mutex);
20             while (count == 0)
21                 Pthread_cond_wait(&fill, &mutex);
22             int tmp = get();
23             Pthread_cond_signal(&empty);
24             Pthread_mutex_unlock(&mutex);
25             printf("%d\n", tmp);
26         }
27     }
```

The Final Producer/Consumer Solution

- More **concurrency** and **efficiency** → Add more buffer slots.
 - Allow concurrent production or consuming to take place.
 - Reduce context switches.

```
1      int buffer[MAX];
2      int fill = 0;
3      int use = 0;
4      int count = 0;
5
6      void put(int value) {
7          buffer[fill] = value;
8          fill = (fill + 1) % MAX;
9          count++;
10     }
11
12     int get() {
13         int tmp = buffer[use];
14         use = (use + 1) % MAX;
15         count--;
16         return tmp;
17     }
```

The Final Put and Get Routines

The Final Producer/Consumer Solution (Cont.)

```
1      cond_t empty, fill;
2      mutex_t mutex;
3
4      void *producer(void *arg) {
5          int i;
6          for (i = 0; i < loops; i++) {
7              Pthread_mutex_lock(&mutex);                                // p1
8              while (count == MAX)                                     // p2
9                  Pthread_cond_wait(&empty, &mutex);                // p3
10             put(i);
11             Pthread_cond_signal(&fill);                            // p4
12             Pthread_mutex_unlock(&mutex);                          // p5
13         }
14     }
15
16     void *consumer(void *arg) {
17         int i;
18         for (i = 0; i < loops; i++) {
19             Pthread_mutex_lock(&mutex);                                // c1
20             while (count == 0)                                       // c2
21                 Pthread_cond_wait(&fill, &mutex);                // c3
22             int tmp = get();                                      // c4
```

The Final Producer/Consumer Solution (Cont.)

(Cont.)

```
23             Pthread_cond_signal(&empty);           // c5
24             Pthread_mutex_unlock(&mutex);          // c6
25             printf("%d\n", tmp);
26         }
27 }
```

The Final Working Solution (Cont.)

- ◆ p2: A **producer** only sleeps if all buffers are currently filled.
- ◆ c2: A **consumer** only sleeps if all buffers are currently empty.

Covering Conditions

- ❑ Assume there are zero bytes free
 - ◆ Thread T_a calls `allocate(100)`.
 - ◆ Thread T_b calls `allocate(10)`.
 - ◆ Both T_a and T_b wait on the condition and go to sleep.
 - ◆ Thread T_c calls `free(50)`.

Which waiting thread should be woken up?

Covering Conditions (Cont.)

```
1      // how many bytes of the heap are free?
2      int bytesLeft = MAX_HEAP_SIZE;
3
4      // need lock and condition too
5      cond_t c;
6      mutex_t m;
7
8      void *
9      allocate(int size) {
10         Pthread_mutex_lock(&m);
11         while (bytesLeft < size)
12             Pthread_cond_wait(&c, &m);
13         void *ptr = ...;           // get mem from heap
14         bytesLeft -= size;
15         Pthread_mutex_unlock(&m);
16         return ptr;
17     }
18
19     void free(void *ptr, int size) {
20         Pthread_mutex_lock(&m);
21         bytesLeft += size;
22         Pthread_cond_signal(&c);    // whom to signal??
23         Pthread_mutex_unlock(&m);
24     }
```

Covering Conditions (Cont.)

- ▣ Solution (Suggested by Lampson and Redell)
 - ◆ Replace `pthread_cond_signal()` with `pthread_cond_broadcast()`
 - ◆ `pthread_cond_broadcast()`
 - Wake up **all waiting threads.**
 - Cost: too many threads might be woken.
 - Threads that shouldn't be awake will simply wake up, re-check the condition, and then go back to sleep.

31. Semaphore

Semaphore: A definition

- ❑ An object **with an integer value**

- ◆ We can manipulate with two routines; `sem_wait()` and `sem_post()`.
- ◆ Initialization

```
1 #include <semaphore.h>
2 sem_t s;
3 sem_init(&s, 0, 1); // initialize s to the value 1
```

- Declare a semaphore `s` and initialize it to the value 1
- The second argument, 0, indicates that the semaphore is shared between
threads in the same process.

Semaphore: Interact with semaphore

▣ sem_wait()

```
1 int sem_wait(sem_t *s) {  
2     decrement the value of semaphore s by one  
3     wait if value of semaphore s is negative  
4 }
```

- ◆ If the value of the semaphore was *one* or *higher* when called `sem_wait()`, **return right away**.
- ◆ It will cause the caller to suspend execution waiting for a subsequent post.
- ◆ When negative, the value of the semaphore is equal to the number of waiting threads.

Semaphore: Interact with semaphore (Cont.)

▣ sem_post()

```
1 int sem_post(sem_t *s) {  
2     increment the value of semaphore s by one  
3     if there are one or more threads waiting, wake one  
4 }
```

- ◆ Simply **increments** the value of the semaphore.
- ◆ If there is a thread waiting to be woken, **wakes** one of them up.

Binary Semaphores (Locks)

- ❑ What should **x** be?
 - ◆ The initial value should be **1**.

```
1 sem_t m;
2 sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
3
4 sem_wait(&m);
5 //critical section here
6 sem_post(&m);
```

Thread Trace: Single Thread Using A Semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	call sema_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
1	sem_post() returns	

Thread Trace: Two Threads Using A Semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() retruns	Running		Ready
0	(crit set: begin)	Running		Ready
0	<i>Interrupt; Switch → T1</i>	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem < 0) → sleep	sleeping
-1		Running	<i>Switch → T0</i>	sleeping
-1	(crit sect: end)	Running		sleeping
-1	call sem_post()	Running		sleeping
0	increment sem	Running		sleeping
0	wait(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	<i>Interrupt; Switch → T1</i>	Ready		Running
0		Ready	sem_wait() retruns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

Semaphores As Condition Variables

```
1  sem_t s;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      sem_post(&s); // signal here: child is done
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     pthread_create(c, NULL, child, NULL);
16     sem_wait(&s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```

A Parent Waiting For Its Child

```
parent: begin
child
parent: end
```

The execution result

- ◆ What should **x** be?
 - The value of semaphore should be set to is **0**.

Thread Trace: Parent Waiting For Child (Case 1)

- The parent call `sem_wait()` before the child has called `sem_post()`.

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call <code>sem_wait()</code>	Running		Ready
-1	decrement sem	Running		Ready
-1	$(\text{sem} < 0) \rightarrow \text{sleep}$	sleeping		Ready
-1	<i>Switch→Child</i>	sleeping	child runs	Running
-1		sleeping	call <code>sem_post()</code>	Running
0		sleeping	increment sem	Running
0		Ready	wake(Parent)	Running
0		Ready	<code>sem_post()</code> returns	Running
0		Ready	<i>Interrupt; Switch→Parent</i>	Ready
0	<code>sem_wait()</code> retruns	Running		Ready

Thread Trace: Parent Waiting For Child (Case 2)

- The child runs to completion before the parent call `sem_wait()`.

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	<i>Interrupt; switch→Child</i>	Ready	child runs	Running
0		Ready	call <code>sem_post()</code>	Running
1		Ready	increment sem	Running
1		Ready	wake (nobody)	Running
1		Ready	<code>sem_post()</code> returns	Running
1	parent runs	Running	<i>Interrupt; Switch→Parent</i>	Ready
1	call <code>sem_wait()</code>	Running		Ready
0	decrement sem	Running		Ready
0	(sem<0)→awake	Running		Ready
0	<code>sem_wait()</code> retruns	Running		Ready

The Producer/Consumer (Bounded-Buffer) Problem

- ▣ **Producer:** `put()` interface
 - ◆ Wait for a buffer to become *empty* in order to put data into it.
- ▣ **Consumer:** `get()` interface
 - ◆ Wait for a buffer to become *filled* before using it.

```
1  int buffer[MAX];
2  int fill = 0;
3  int use = 0;
4
5  void put(int value) {
6      buffer[fill] = value;      // line f1
7      fill = (fill + 1) % MAX; // line f2
8  }
9
10 int get() {
11     int tmp = buffer[use];    // line g1
12     use = (use + 1) % MAX;   // line g2
13     return tmp;
14 }
```

The Producer/Consumer (Bounded-Buffer) Problem

```
1  sem_t empty;
2  sem_t full;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          sem_wait(&empty);           // line P1
8          put(i);                  // line P2
9          sem_post(&full);         // line P3
10     }
11 }
12
13 void *consumer(void *arg) {
14     int i, tmp = 0;
15     while (tmp != -1) {
16         sem_wait(&full);        // line C1
17         tmp = get();            // line C2
18         sem_post(&empty);       // line C3
19         printf("%d\n", tmp);
20     }
21 }
22 ...
```

First Attempt: Adding the Full and Empty Conditions

The Producer/Consumer (Bounded-Buffer) Problem

```
21 int main(int argc, char *argv[]) {  
22     // ...  
23     sem_init(&empty, 0, MAX);           // MAX buffers are empty to begin with...  
24     sem_init(&full, 0, 0);            // ... and 0 are full  
25     // ...  
26 }
```

First Attempt: Adding the Full and Empty Conditions (Cont.)

- ◆ Imagine that `MAX` is greater than 1 .
 - If there are multiple producers, **race condition** can happen at line *f1*.
 - It means that the old data there is overwritten.
- ◆ We've forgotten here is **mutual exclusion**.
 - The filling of a buffer and incrementing of the index into the buffer is a **critical section**.

A Solution: Adding Mutual Exclusion

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&mutex);           // line p0 (NEW LINE)
9          sem_wait(&empty);         // line p1
10         put(i);                // line p2
11         sem_post(&full);        // line p3
12         sem_post(&mutex);       // line p4 (NEW LINE)
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&mutex);           // line c0 (NEW LINE)
20         sem_wait(&full);          // line c1
21         int tmp = get();          // line c2
22         ...
23 }
```

Adding Mutual Exclusion (Incorrectly)

A Solution: Adding Mutual Exclusion (Cont.)

```
22         sem_post(&empty);           // line c3
23         sem_post(&mutex);          // line c4 (NEW LINE)
24         printf("%d\n", tmp);
25     }
26 }
```

Adding Mutual Exclusion (Incorrectly) (Cont.)

- ◆ Imagine two threads: one producer and one consumer.
 - The consumer **acquire** the `mutex` (line c0).
 - The consumer **calls** `sem_wait()` on the full semaphore (line c1).
 - The consumer is **blocked** and **yield** the CPU.
 - The consumer still holds the mutex!
 - The producer **calls** `sem_wait()` on the binary `mutex` semaphore (line p0).
 - The producer is now **stuck** waiting too. **a classic deadlock.**

Finally, A Working Solution

```
1  sem_t empty;
2  sem_t full;
3  sem_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          sem_wait(&empty);           // line p1
9          sem_wait(&mutex);         // line p1.5 (MOVED MUTEX HERE...)
10         put(i);
11         sem_post(&mutex);        // line p2
12         sem_post(&full);         // line p3
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&full);          // line c1
20         sem_wait(&mutex);         // line c1.5 (MOVED MUTEX HERE...)
21         int tmp = get();          // line c2
22         sem_post(&mutex);         // line c2.5 (... AND HERE)
23         ...
24 }
```

Adding Mutual Exclusion (Correctly)

Finally, A Working Solution

```
23             sem_post(&empty);           // line c3
24             printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with ...
31     sem_init(&full, 0, 0);   // ... and 0 are full
32     sem_init(&mutex, 0, 1); // mutex=1 because it is a lock
33     // ...
34 }
```

Adding Mutual Exclusion (Correctly)

Reader-Writer Locks

- ▣ Imagine a number of concurrent list operations, including **inserts** and simple **lookups**.
 - ◆ **insert:**
 - Change the state of the list
 - A traditional critical section makes sense.
 - ◆ **lookup:**
 - Simply *read* the data structure.
 - As long as we can guarantee that no insert is on-going, we can allow many lookups to proceed **concurrently**.

This special type of lock is known as a **reader-write lock**.

A Reader-Writer Locks

- ▣ Only a **single writer** can acquire the lock.
- ▣ Once a reader has acquired a **read lock**,
 - ◆ **More readers** will be allowed to acquire the read lock too.
 - ◆ A writer will have to wait until all readers are finished.

```
1  typedef struct _rwlock_t {  
2      sem_t lock;          // binary semaphore (basic lock)  
3      sem_t writelock;    // used to allow ONE writer or MANY readers  
4      int readers;        // count of readers reading in critical section  
5  } rwlock_t;  
6  
7  void rwlock_init(rwlock_t *rw) {  
8      rw->readers = 0;  
9      sem_init(&rw->lock, 0, 1);  
10     sem_init(&rw->writelock, 0, 1);  
11  }  
12  
13  void rwlock_acquire_readlock(rwlock_t *rw) {  
14      sem_wait(&rw->lock);  
15      ...  
16  }
```

A Reader-Writer Locks (Cont.)

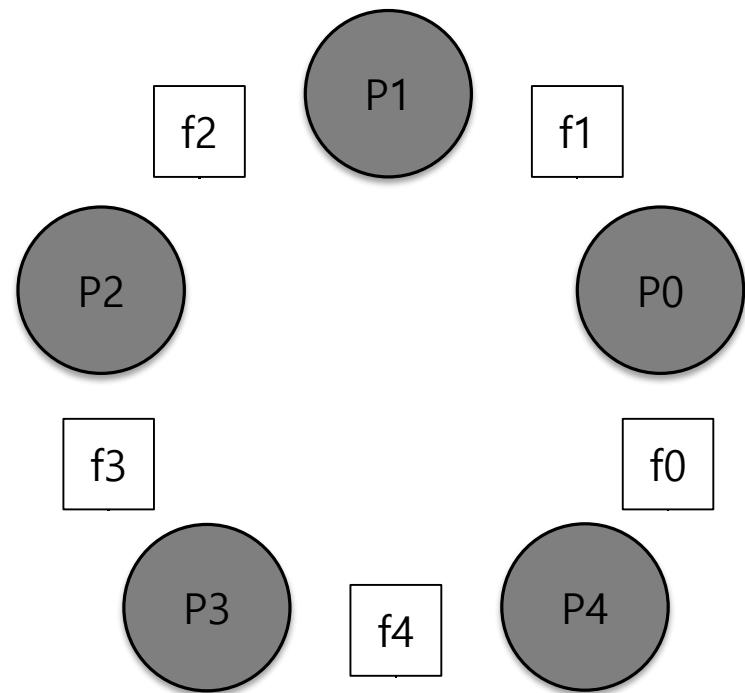
```
15     rw->readers++;
16     if (rw->readers == 1)
17         sem_wait(&rw->writelock); // first reader acquires writelock
18     sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0)
25         sem_post(&rw->writelock); // last reader releases writelock
26     sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }
```

A Reader-Writer Locks (Cont.)

- ▣ The reader-writer locks have **fairness problem**.
 - ◆ It would be relatively easy for reader to **starve writer**.
 - ◆ How to prevent more readers from entering the lock once a writer is waiting?

The Dining Philosophers

- ❑ Assume there are five “**philosophers**” sitting around a table.
 - ◆ Between each pair of philosophers is a single fork (five total).
 - ◆ The philosophers each have times where they **think**, and don’t need any forks, and times where they **eat**.
 - ◆ In order to *eat*, a philosopher needs **two forks**, both the one on their *left* and the one on their *right*.
 - ◆ **The contention for these forks.**



The Dining Philosophers (Cont.)

▣ Key challenge

- ◆ There is **no deadlock**.
- ◆ **No** philosopher **starves** and never gets to eat.
- ◆ **Concurrency** is high.

```
while (1) {  
    think();  
    getforks();  
    eat();  
    putforks();  
}
```

Basic loop of each philosopher

```
// helper functions  
int left(int p) { return p; }  
  
int right(int p) {  
    return (p + 1) % 5;  
}
```

Helper functions (Downey's solutions)

- Philosopher p wishes to refer to the fork on their left \rightarrow call `left(p)`.
- Philosopher p wishes to refer to the fork on their right \rightarrow call `right(p)`.

The Dining Philosophers (Cont.)

- We need some **semaphore**, one for each fork: `sem_t forks[5]`.

```
1 void getforks() {
2     sem_wait(forks[left(p)]);
3     sem_wait(forks[right(p)]);
4 }
5
6 void putforks() {
7     sem_post(forks[left(p)]);
8     sem_post(forks[right(p)]);
9 }
```

The `getforks()` and `putforks()` Routines (Broken Solution)

- ◆ Deadlock occur!
 - If each philosopher happens to **grab the fork on their left** before any philosopher can grab the fork on their right.
 - Each will be stuck *holding one fork* and waiting for another, *forever*.

A Solution: Breaking The Dependency

- Change how forks are acquired.
 - Let's assume that philosopher 4 acquire the forks in a *different order*.

```
1 void getforks() {
2     if (p == 4) {
3         sem_wait(forks[right(p)]);
4         sem_wait(forks[left(p)]);
5     } else {
6         sem_wait(forks[left(p)]);
7         sem_wait(forks[right(p)]);
8     }
9 }
```

- There is no situation where each philosopher grabs one fork and is stuck waiting for another. **The cycle of waiting is broken.**

How To Implement Semaphores

- Build our own version of semaphores called **Zemaphores**

```
1  typedef struct __Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21 ...
```

How To Implement Semaphores (Cont.)

```
22 void Zem_post(Zem_t *s) {  
23     Mutex_lock(&s->lock);  
24     s->value++;  
25     Cond_signal(&s->cond);  
26     Mutex_unlock(&s->lock);  
27 }
```

- ◆ Zemaphore don't maintain the invariant that *the value* of the semaphore.
 - The value never be lower than zero.
 - This behavior is **easier** to implement and **matches** the current Linux implementation.

32. Common Concurrency Problems.

Common Concurrency Problems

- ▣ More recent work focuses on studying other types of **common concurrency bugs**.
 - ◆ Take a brief look at some example concurrency problems found in real code bases.

What Types Of Bugs Exist?

- Focus on four major open-source applications
 - MySQL, Apache, Mozilla, OpenOffice.

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
Open Office	Office Suite	6	2
Total		74	31

Bugs In Modern Applications

Non-Deadlock Bugs

- ▣ Make up a majority of concurrency bugs.
- ▣ Two major types of non deadlock bugs:
 - ◆ Atomicity violation
 - ◆ Order violation

Atomicity-Violation Bugs

- ❑ The desired **serializability** among multiple memory accesses is *violated*.
 - ◆ Simple Example found in MySQL:
 - Two different threads access the field `proc_info` in the struct `thd`.

```
1  Thread1::  
2  if(thd->proc_info){  
3      ...  
4      fputs(thd->proc_info , ...);  
5      ...  
6  }  
7  
8  Thread2::  
9  thd->proc_info = NULL;
```

Atomicity-Violation Bugs (Cont.)

- ▣ **Solution:** Simply add locks around the shared-variable references.

```
1  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Thread1::
4  pthread_mutex_lock(&lock);
5  if(thd->proc_info) {
6      ...
7      fputs(thd->proc_info , ...);
8      ...
9  }
10 pthread_mutex_unlock(&lock);
11
12 Thread2::
13 pthread_mutex_lock(&lock);
14 thd->proc_info = NULL;
15 pthread_mutex_unlock(&lock);
```

Order-Violation Bugs

- The **desired order** between two memory accesses is flipped.
 - ◆ i.e., **A** should always be executed before **B**, but the order is not enforced during execution.
 - ◆ **Example:**
 - The code in Thread2 seems to assume that the variable `mThread` has already been *initialized* (and is not `NULL`).

```
1  Thread1::
2  void init() {
3      mThread = PR_CreateThread(mMain, ...);
4  }
5
6  Thread2::
7  void mMain(...) {
8      mState = mThread->State
9  }
```

Order-Violation Bugs (Cont.)

- ▣ **Solution:** Enforce ordering using **condition variables**

```
1  pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
3  int mtInit = 0;
4
5  Thread 1::
6  void init() {
7      ...
8      mThread = PR_CreateThread(mMain,...);
9
10     // signal that the thread has been created.
11     pthread_mutex_lock(&mtLock);
12     mtInit = 1;
13     pthread_cond_signal(&mtCond);
14     pthread_mutex_unlock(&mtLock);
15     ...
16 }
17
18 Thread2::
19 void mMain(...){
20     ...
```

Order-Violation Bugs (Cont.)

```
21     // wait for the thread to be initialized ...
22     pthread_mutex_lock(&mtLock);
23     while(mtInit == 0)
24         pthread_cond_wait(&mtCond, &mtLock);
25     pthread_mutex_unlock(&mtLock);
26
27     mState = mThread->State;
28     ...
29 }
```

Deadlock Bugs

Thread 1:

lock(L1);

lock(L2);

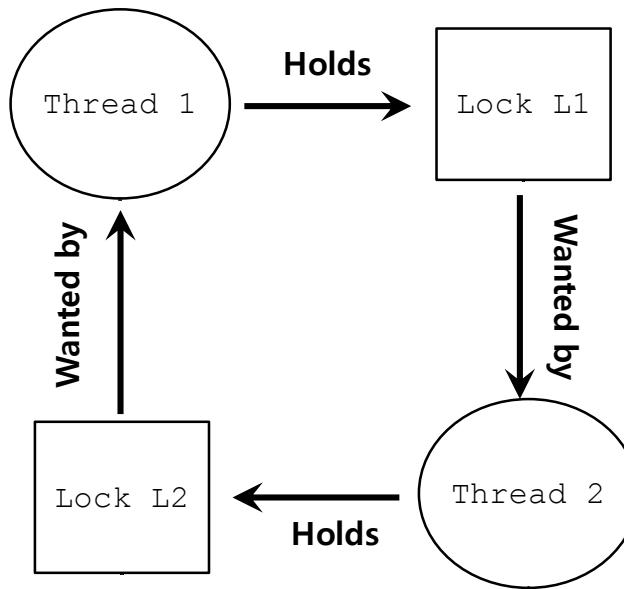
Thread 2:

lock(L2);

lock(L1);

- ◆ The presence of **a cycle**

- Thread1 is holding a lock L1 and waiting for another one, L2.
- Thread2 that holds lock L2 is waiting for L1 to be release.



Why Do Deadlocks Occur?

- Reason 1:
 - ◆ In large code bases, **complex dependencies** arise between components.
- Reason 2:
 - ◆ Due to the nature of **encapsulation**
 - Hide details of implementations and make software easier to build in a modular way.
 - Such **modularity** *does not mesh well with locking.*

Why Do Deadlocks Occur? (Cont.)

▫ Example: Java Vector class and the method AddAll()

```
1  Vector v1,v2;  
2  v1.AddAll(v2);
```

- ◆ **Locks** for both the vector being added to (`v1`) and the parameter (`v2`)
need to be acquired.
 - The routine acquires said locks in some arbitrary order (`v1` then `v2`).
 - If some other thread calls `v2.AddAll(v1)` at nearly the same time → We have the potential for **deadlock**.

Conditional for Deadlock

- Four conditions need to hold for a deadlock to occur.

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

- If any of these four conditions are not met, **deadlock cannot occur**.

Prevention – Circular Wait

- ▣ Provide a **total ordering** on lock acquisition
 - ◆ This approach requires *careful design* of global locking strategies.
- ▣ **Example:**
 - ◆ There are two locks in the system (L1 and L2)
 - ◆ We can prevent deadlock by always acquiring L1 before L2.

Prevention – Hold-and-wait

- Acquire all locks **at once, atomically**.

```
1  lock(prevention);  
2  lock(L1);  
3  lock(L2);  
4  ...  
5  unlock(prevention);
```

- This code guarantees that **no untimely thread switch can occur in the midst of lock acquisition**.
- Problem:**
 - Require us to know when calling a routine exactly which locks must be held and to acquire them ahead of time.
 - Decrease *concurrency*

Prevention – No Preemption

- ▣ **Multiple lock acquisition** often gets us into trouble because when waiting for one lock **we are holding another**.
- ▣ `trylock()`
 - ◆ Used to build a *deadlock-free, ordering-robust* lock acquisition protocol.
 - ◆ Grab the lock (if it is available).
 - ◆ Or, return -1: you should try again later.

```
1  top:  
2      lock(L1);  
3      if( tryLock(L2) == -1 ) {  
4          unlock(L1);  
5          goto top;  
6      }
```

Prevention – No Preemption (Cont.)

❑ livelock

- ◆ Both systems are running through the code sequence *over and over again*.
- ◆ Progress is not being made.
- ◆ Solution:
 - Add **a random delay** before looping back and trying the entire thing over again.

Prevention – Mutual Exclusion

- ❑ wait-free

- ◆ Using powerful **hardware instruction**.
- ◆ You can build data structures in a manner that *does not require explicit locking*.

```
1 int CompareAndSwap(int *address, int expected, int new) {  
2     if (*address == expected) {  
3         *address = new;  
4         return 1; // success  
5     }  
6     return 0;  
7 }
```

Prevention – Mutual Exclusion (Cont.)

- We now wanted to **atomically increment** a value by a certain amount:

```
1 void AtomicIncrement(int *value, int amount) {  
2     do {  
3         int old = *value;  
4         }while( CompareAndSwap(value, old, old+amount)==0);  
5 }
```

- ◆ Repeatedly tries to update the value to *the new amount* and uses the compare-and-swap to do so.
- ◆ **No lock** is acquired
- ◆ **No deadlock** can arise
- ◆ **livelock** is still a possibility.

Prevention – Mutual Exclusion (Cont.)

▣ More complex example: list insertion

```
1 void insert(int value) {  
2     node_t * n = malloc(sizeof(node_t));  
3     assert( n != NULL );  
4     n->value = value ;  
5     n->next = head;  
6     head = n;  
7 }
```

- ◆ If called by multiple threads at the "*same time*", this code has a **race condition**.

Prevention – Mutual Exclusion (Cont.)

▣ Solution:

- ◆ Surrounding this code with a **lock acquire** and **release**.

```
1 void insert(int value){  
2     node_t * n = malloc(sizeof(node_t));  
3     assert( n != NULL );  
4     n->value = value ;  
5     lock(listlock); // begin critical section  
6     n->next = head;  
7     head = n;  
8     unlock(listlock) ; //end critical section  
9 }
```

- ◆ **wait-free manner** using the compare-and-swap instruction

```
1 void insert(int value) {  
2     node_t *n = malloc(sizeof(node_t));  
3     assert(n != NULL);  
4     n->value = value;  
5     do {  
6         n->next = head;  
7     } while (CompareAndSwap(&head, n->next, n));  
8 }
```

Deadlock Avoidance via Scheduling

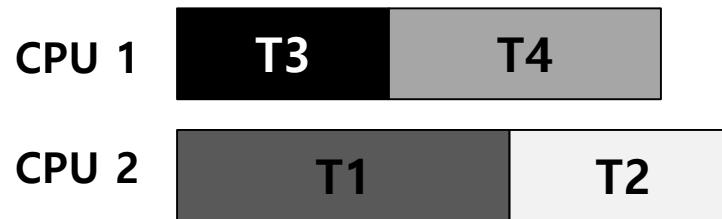
- ▣ In some scenarios **deadlock avoidance** is preferable.
 - ◆ **Global knowledge** is required:
 - Which locks various threads might grab during their execution.
 - Subsequently schedules said threads in a way as to guarantee no deadlock can occur.

Example of Deadlock Avoidance via Scheduling (1)

- We have two processors and four threads.
 - ◆ Lock acquisition demands of the threads:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

- ◆ A smart scheduler could compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise.



Example of Deadlock Avoidance via Scheduling (2)

- More contention for the same resources

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

- A possible schedule that guarantees that *no deadlock* could ever occur.



- The total time to complete the jobs is lengthened considerably.

Detect and Recover

- ▣ Allow deadlock to occasionally occur and then *take some action.*
 - ◆ Example: if an OS froze, you would reboot it.
- ▣ Many database systems employ *deadlock detection* and *recovery technique*.
 - ◆ A deadlock detector **runs periodically**.
 - ◆ Building a **resource graph** and checking it for cycles.
 - ◆ In deadlock, the system **need to be restarted**.

33. Event-based Concurrency (Advanced)

Event-based Concurrency

- ▣ A different style of **concurrent programming**
 - ◆ Used in *GUI-based applications*, some types of *internet servers*.
- ▣ **The problem** that event-based concurrency addresses is two-fold.
 - ◆ Managing concurrency correctly in multi-threaded applications.
 - Missing locks, deadlock, and other nasty problems can arise.
 - ◆ The developer has little or no control over what is scheduled at a given moment in time.

The Basic Idea: An Event Loop

- The approach:

- ◆ **Wait** for something (i.e., an "*event*") to occur.
- ◆ When it does, **check** what type of event it is.
- ◆ **Do** the small amount of work it requires.

- Example:

```
1  while(1) {  
2      events = getEvents();  
3      for( e in events )  
4          processEvent(e); // event handler  
5  }
```

A canonical event-based server (Pseudo code)

How exactly does an event-based server determine which events are taking place.

An Important API: `select()` (or `poll()`)

- Check whether there is any **incoming I/O** that should be attended to.
 - ◆ `select()`

```
int select(int nfds,
           fd_set * restrict readfds,
           fd_set * restrict writefds,
           fd_set * restrict errorfds,
           struct timeval * restrict timeout);
```

- Lets a server determine that a **new packet has arrived** and is in need of processing.
- Let the service know when **it is OK to reply**.
- `timeout`
 - `NULL`: Cause `select()` to *block indefinitely* until some descriptor is ready.
 - `0`: Use the call to `select()` to *return immediately*.

Using select()

- How to use `select()` to see which network descriptors have incoming messages upon them.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6
7 int main(void) {
8     // open and set up a bunch of sockets (not shown)
9     // main loop
10    while (1) {
11        // initialize the fd_set to all zero
12        fd_set readFDs;
13        FD_ZERO(&readFDs);
14
15        // now set the bits for the descriptors
16        // this server is interested in
17        // (for simplicity, all of them from min to max)
18        ...
```

Simple Code using `select()`

Using select() (Cont.)

```
18         int fd;
19         for (fd = minFD; fd < maxFD; fd++)
20             FD_SET(fd, &readFDs);
21
22         // do the select
23         int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);
24
25         // check which actually have data using FD_ISSET()
26         int fd;
27         for (fd = minFD; fd < maxFD; fd++)
28             if (FD_ISSET(fd, &readFDs))
29                 processFD(fd);
30     }
31 }
```

Simple Code using `select()` (Cont.)

Why Simpler? No Locks Needed

- ▣ The event-based server **cannot be interrupted** by another thread.
 - ◆ With a single CPU and an event-based application.
 - ◆ It is decidedly **single threaded**.
 - ◆ Thus, *concurrency bugs* common in threaded programs **do not manifest** in the basic event-based approach.

A Problem: Blocking System Calls

- ❑ What if an event requires that you issue **a system call** that might block?
 - ◆ There are no other threads to run: *just the main event loop*
 - ◆ The entire server will do just that: **block until the call completes.**
 - ◆ Huge potential waste of resources

In event-based systems: **no blocking calls** are allowed.

A Solution: Asynchronous I/O

- Enable an application to issue an I/O request and **return control immediately** to the caller, before the I/O has completed.
 - Example:

```
struct aiocb {  
    int aio_fildes;           /* File descriptor */  
    off_t aio_offset;        /* File offset */  
    volatile void *aio_buf;   /* Location of buffer */  
    size_t aio_nbytes;       /* Length of transfer */  
};
```

- An Interface provided on *Max OS X*
- The APIs revolve around a basic structure, the `struct aiocb` or **AIO control block** in common terminology.

A Solution: Asynchronous I/O (Cont.)

▣ Asynchronous API:

- ◆ To issue an asynchronous read to a file

```
int aio_read(struct aiocb *aiocbp);
```

- If successful, it returns right away and the application can continue with its work.

- ◆ Checks whether the request referred to by aiocbp has completed.

```
int aio_error(const struct aiocb *aiocbp);
```

- An application can **periodically pool** the system via `aio_error()`.
 - If it has completed, returns success.
 - If not, EINPROGRESS is returned.

A Solution: Asynchronous I/O (Cont.)

▣ Interrupt

- ◆ Remedy **the overhead to check** whether an I/O has completed
- ◆ Using **UNIX signals** to inform applications when an asynchronous I/O completes.
- ◆ Removing the need to *repeatedly ask the system.*

Another Problem: State Management

- ▣ The code of event-based approach is generally **more complicated** to write than *traditional thread-based* code.
 - ◆ It must package up some program state for the next event handler to use when the I/O completes.
 - ◆ The state the program needs is on the stack of the thread. → **manual stack management**

Another Problem: State Management (Cont.)

▫ Example (an event-based system):

```
int rc = read(fd, buffer, size);  
rc = write(sd, buffer, size);
```

- ◆ First **issue** the read asynchronously.
- ◆ Then, **periodically check** for completion of the read.
- ◆ That call informs us that the **read is complete**.
- ◆ How does the event-based server know **what to do?**

Another Problem: State Management (Cont.)

▣ Solution: continuation

- ◆ **Record** the needed information to finish processing this event *in some data structure*.
- ◆ When the event happens (i.e., when the disk I/O completes), **look up** the needed information and process the event.

What is still difficult with Events.

- ▣ Systems moved from a single CPU to multiple CPUs.
 - ◆ Some of the simplicity of the event-based approach disappeared.
- ▣ It does not integrate well with certain kinds of systems activity.
 - ◆ **Ex. Paging:** A server will not make progress until page fault completes (implicit blocking).
- ▣ Hard to manage overtime: The exact semantics of various routines changes.
- ▣ Asynchronous disk I/O never quite integrates with asynchronous network I/O in as simple and uniform a manner as you might think.

ASIDE: Unix Signals

- Provide a way to communicate with a process.
 - *HUP* (hang up), *INT*(interrupt), *SEGV*(segmentation violation), and etc.
 - **Example:** When your program encounters a *segmentation violation*, the OS sends it a *SIGSEGV*.

```
#include <stdio.h>
#include <signal.h>
void handle(int arg) {
    printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
    signal(SIGHUP, handle);
    while (1)
        ; // doin' nothin' except catchin' some sigs
    return 0;
}
```

A simple program that goes into an infinite loop

ASIDE: Unix Signals (Cont.)

- You can send signals to it with the **kill command** line tool.
 - ◆ Doing so will *interrupt the main while loop* in the program and run the handler code `handle()`.

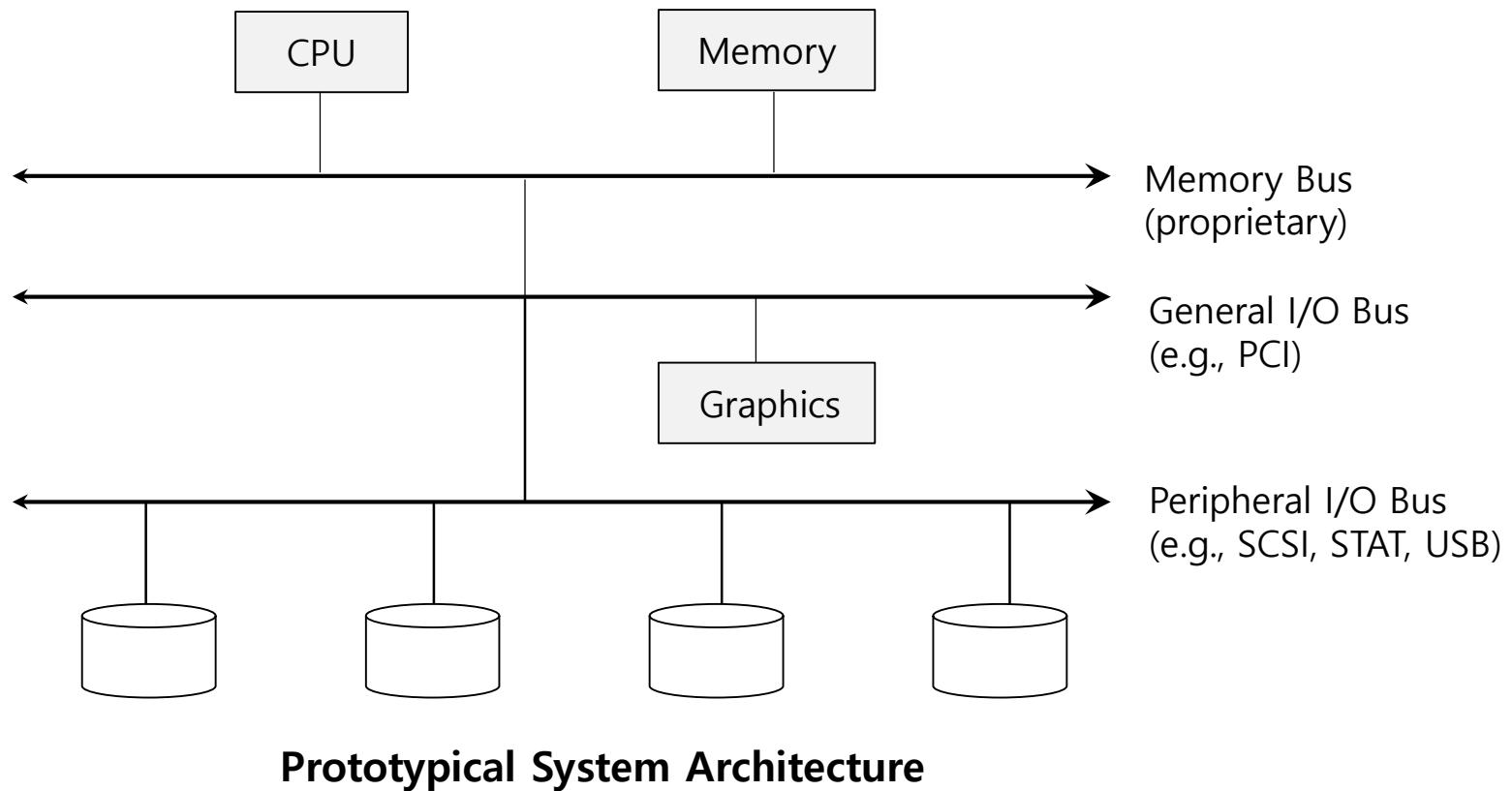
```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```

36. I/O Devices

I/O Devices

- ▣ 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**.

I/O Architecture

- ▣ Buses

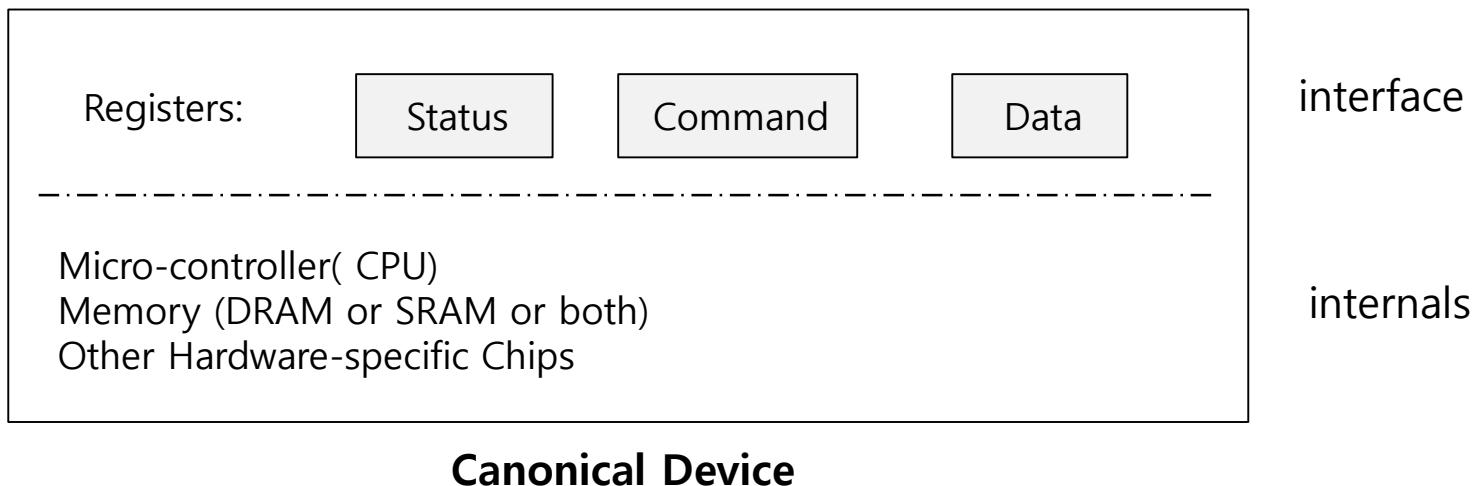
- ◆ Data paths that provided to enable information between CPU(s), RAM, and I/O devices.

- ▣ 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.

Canonical Device

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



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

```
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.

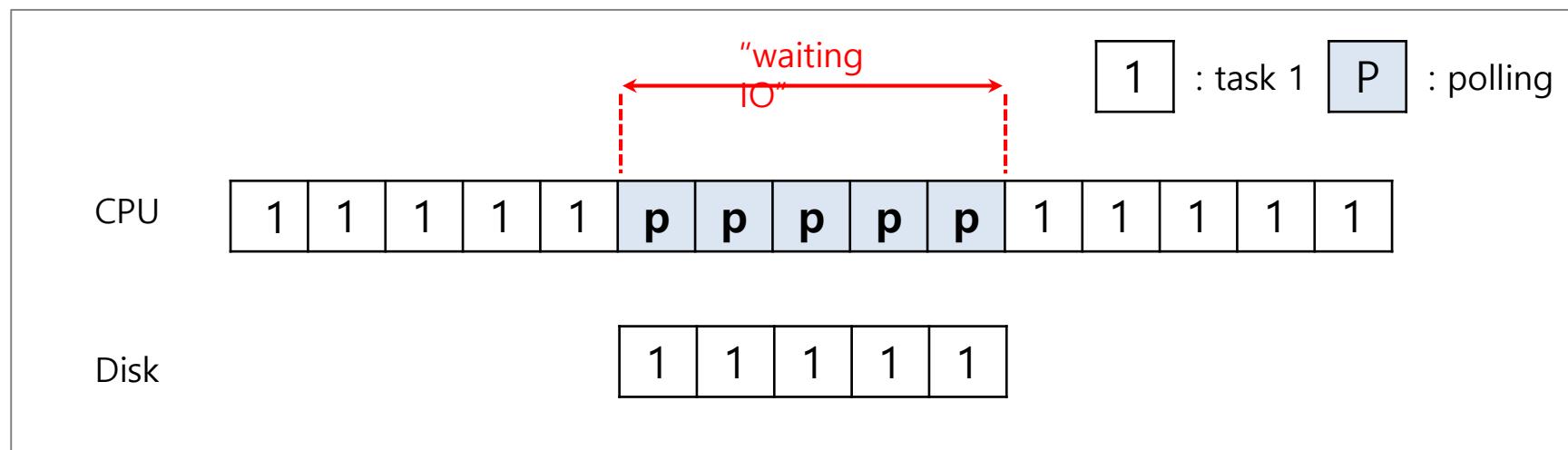


Diagram of CPU utilization by polling

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.
 - ◆ Positive aspect is allow to CPU and the disk are properly utilized.

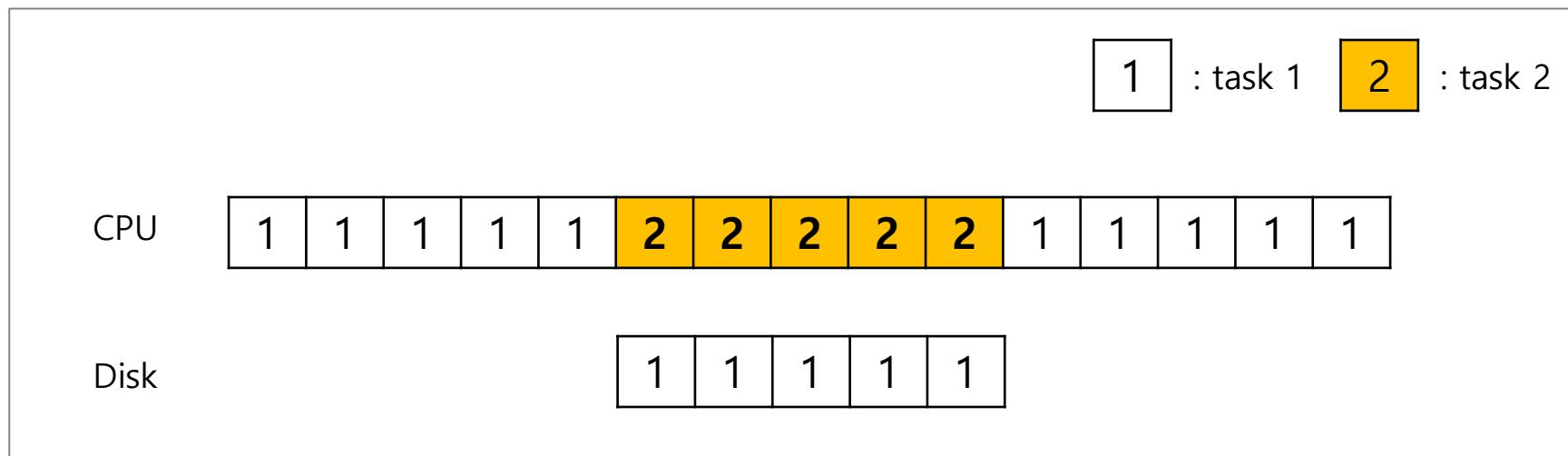


Diagram of CPU utilization by interrupt

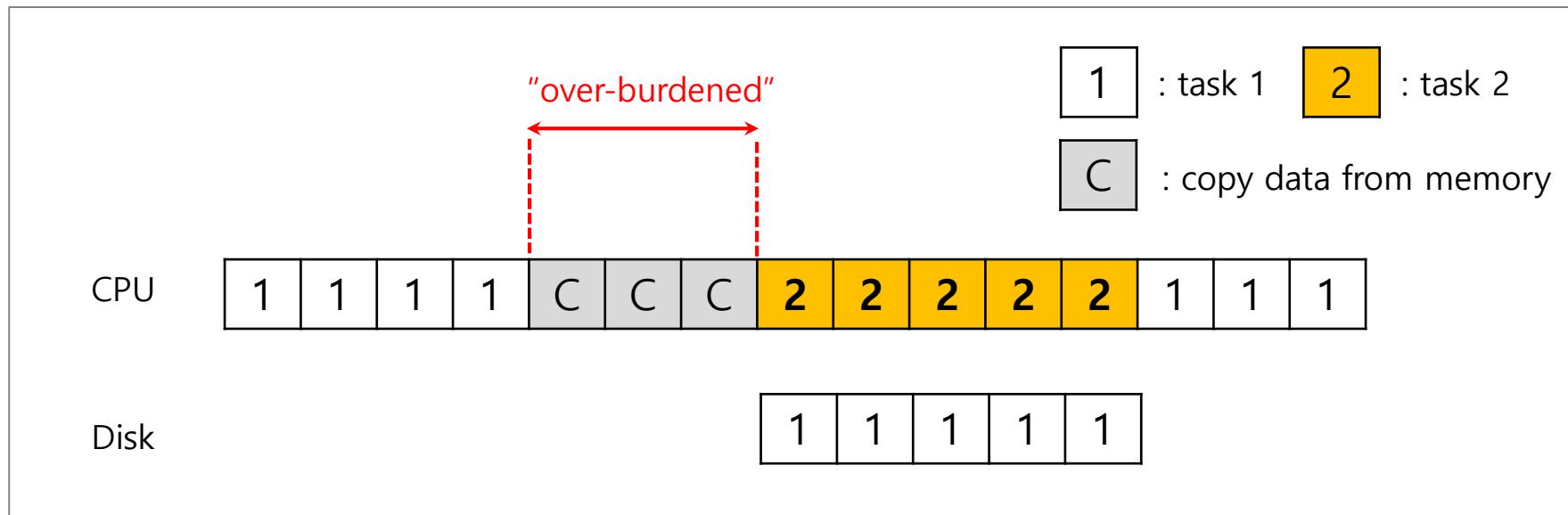
Polling vs interrupts

- ▣ However, “interrupts is not always the best solution”
 - ◆ If, device performs very quickly, 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.

CPU is once again over-burdened

- CPU wastes a lot of time to copy the *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”
- When completed, DMA raises an interrupt, I/O begins on Disk.

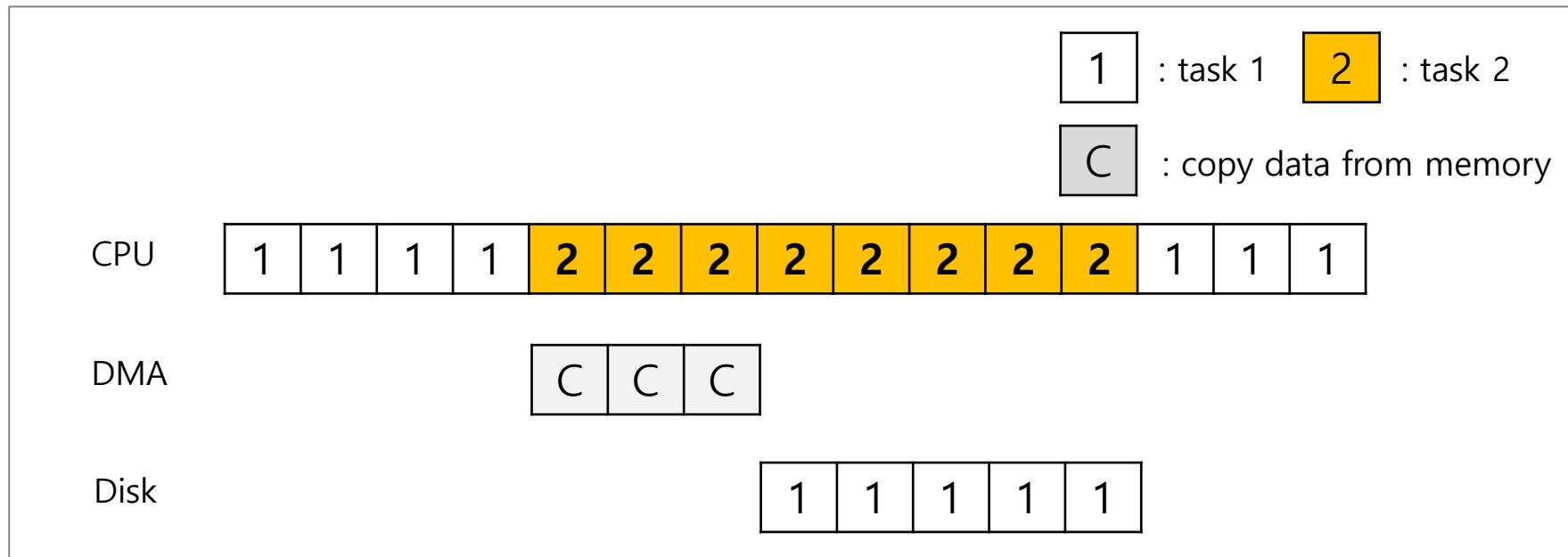


Diagram of CPU utilization by DMA

Device interaction

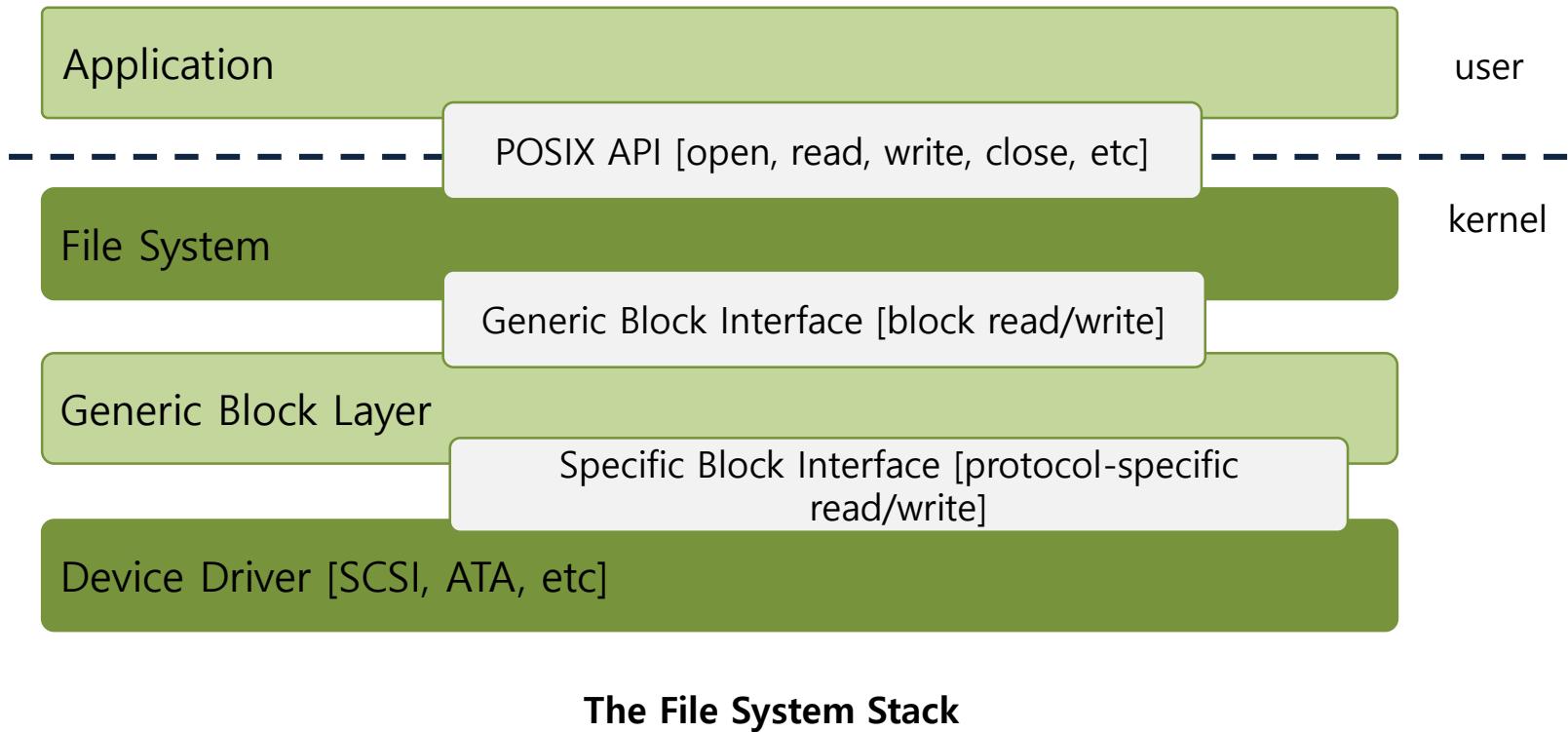
- ▣ How the OS communicates with the **device**?
- ▣ Solutions
 - ◆ I/O instructions: a way for the OS to send data to specific device registers.
 - Ex) in and out instructions on x86
 - ◆ memory-mapped I/O
 - Device registers available as if they were memory locations.
 - The OS load (to read) or store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

Device interaction (Cont.)

- ▣ 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.
- ▣ Solutions: **Abstraction**
 - ◆ Abstraction encapsulate **any specifics of device interaction.**

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

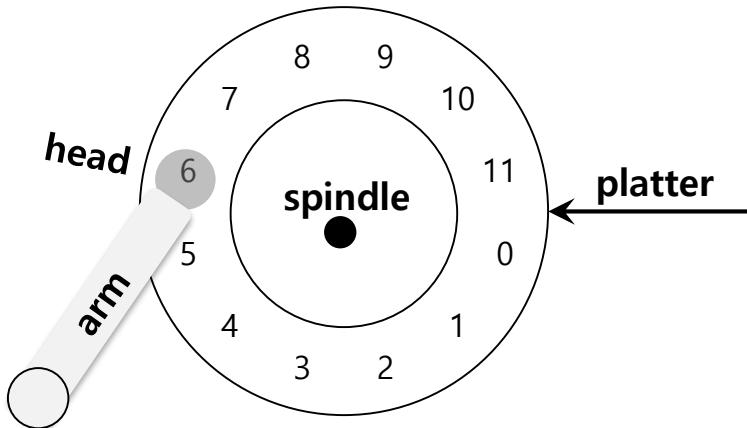
- ❑ If there is a device having many special capabilities, these capabilities **will go unused** in the generic interface layer.
- ❑ Over 70% of OS code is found in device drivers.
 - ◆ Any device drivers are needed because you might plug it to your system.
 - ◆ They are primary contributor to **kernel crashes**, making **more bugs**.

37. Hard Disk Drives

Hard Disk Driver

- Drive consists of a large number of sectors (512 byte blocks).
- Sectors are numbered from 0 to $n-1$ on a disk.
- Drive manufactures guarantee a single sector (512 byte) write atomic.

Basic Geometry of Disk

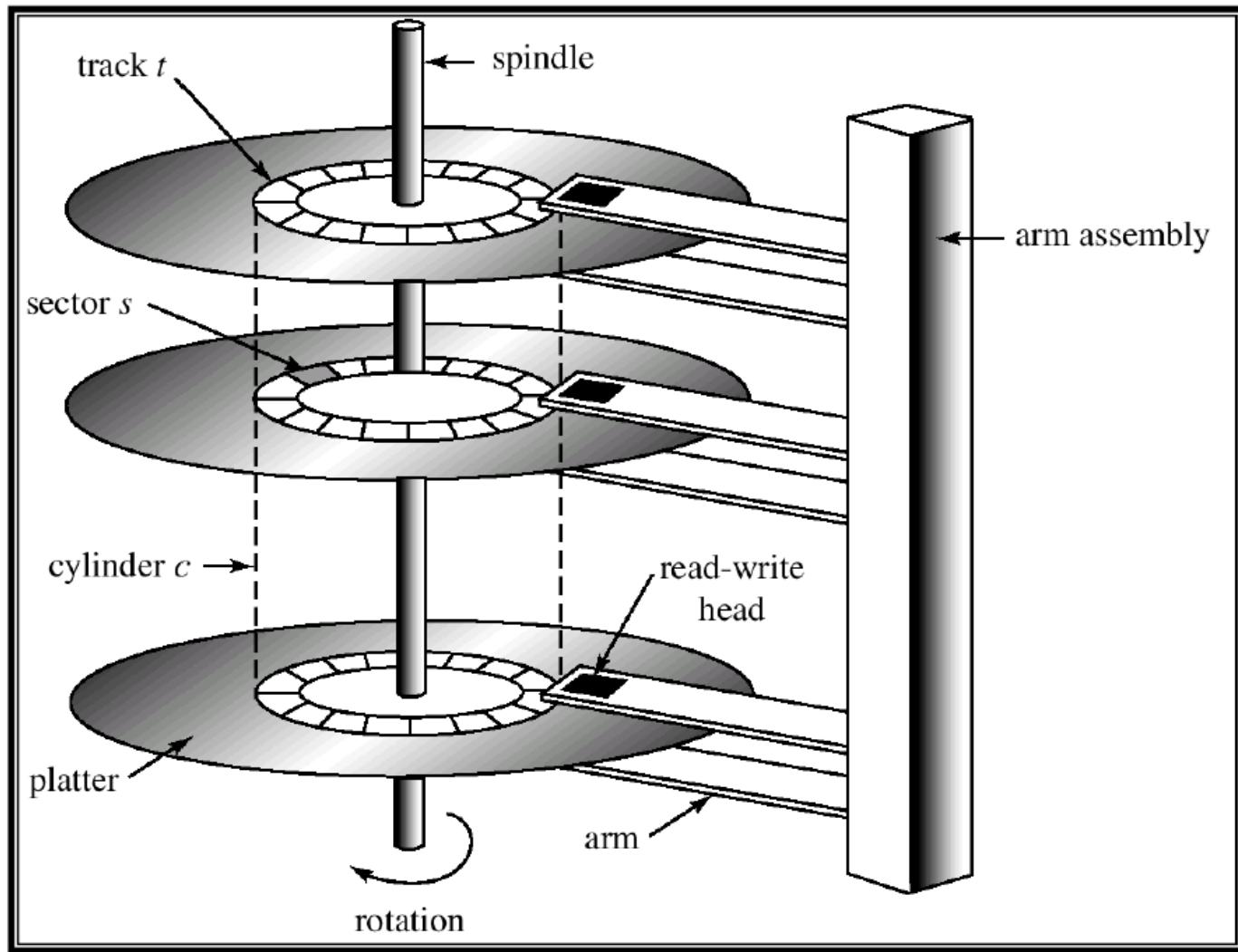


A Disk with Single Track

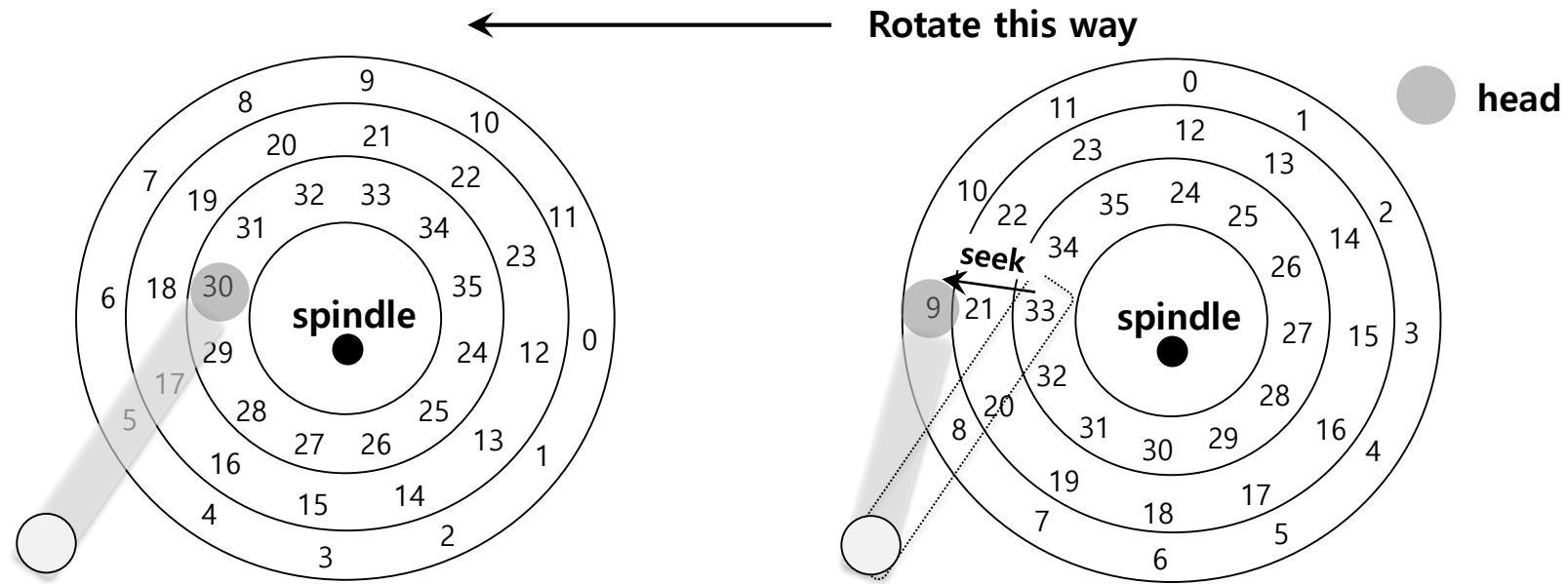
- **Platters:** A circular hard surface on which data is stored.
 - ◆ The platters are all bounded around the **spindle**, spins the platters.
 - ◆ Platter contains many thousands of **tracks**.
- **Disk head:** read and write from each tracks.
 - ◆ The **disk head** is attached to a single **disk arm**, which moves across the surface to position the head over the desired track.

Basic Geometry of Disk (Cont.)

□ Reference



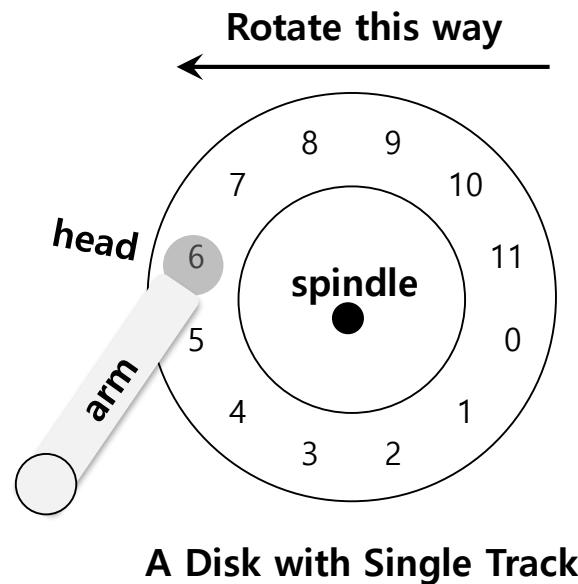
Seek Time



A Disk with Three Track (Right: Seek to Sector 9)

- ▣ **Seek time:** Time to move head to the track contain the desired sector.
 - ◆ Move the disk arm to the desired track. (Platter of course has rotated.)
 - ◆ Waiting for the rotational delay.

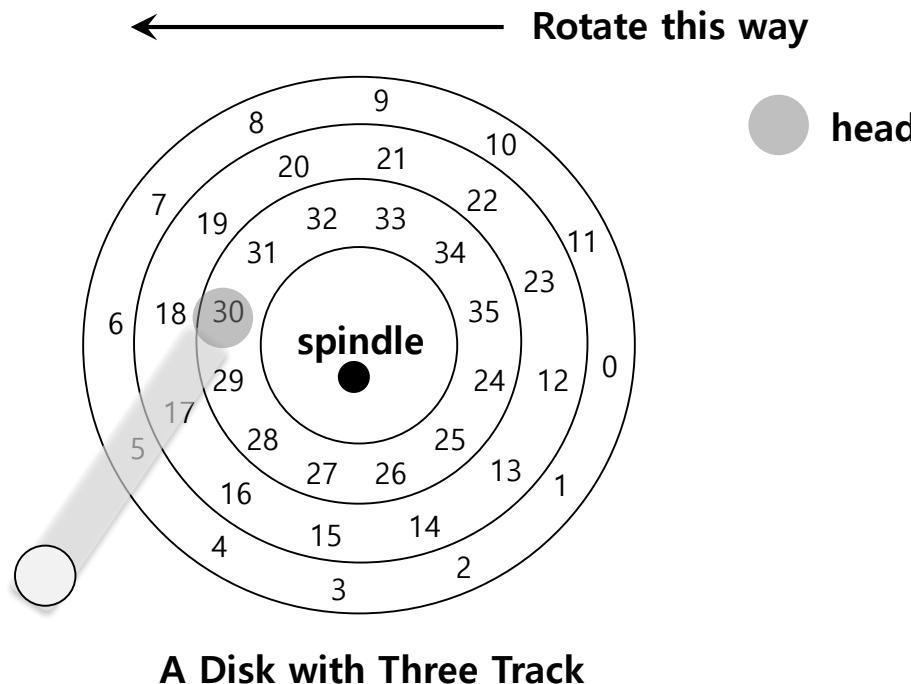
Rotational delay



- **Rotational delay:** Time for the desired sector to rotate.

- ◆ Ex) Full rotational delay is R .
 - Read sector 0: Rotational delay = $\frac{R}{2}$
 - Read sector 5: Rotational delay = $R-1$ (worst case.)

Disk Scheduling



- ▣ SSTF: Shortest Seek Time First
- ▣ Elevator (SCAN, C-SCAN)
- ▣ SPTF: Shortest Positioning Time First

38. RAID

RAID (Redundant Array of Inexpensive Disks)

- ▣ RAID is to use multiple disks in **faster, bigger, and more reliable**.
- ▣ RAID is arranged into six different levels.
 - ◆ RAID Level 0: Striping multiple disks
 - ◆ RAID Level 1: Use mirroring
 - ◆ RAID Level 4/5: Parity based redundancy

RAID Level 0

- RAID Level 0 is the simplest form as **striping** blocks.
 - Spread the blocks across the disks in a round-robin fashion.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

RAID-0: Simple Striping

RAID Level 0 (Cont.)

- An example of RAID Label 0 with a bigger chunk size
 - ◆ Chunk size : 2 blocks (8KB)

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	
1	3	5	7	chunk size: 2 blocks
5	10	12	14	
9	11	13	15	

Striping with a Bigger Chunk Size

Chunk size affects performance of the array.

However, the determining the “best” chunk sizes is hard to do.

RAID Level 0 Analysis

- ▣ Evaluate the capacity, reliability, performance of striping.
 - ◆ First way: single request latency
 - How much parallelism can exist during a single I/O operation.
 - ◆ Second way: steady-state throughput of the RAID:
 - Total bandwidth of many concurrent requests.

RAID Level 0 Analysis (Cont.)

- ▣ Workload: sequential (S) vs random (R)
 - ◆ Sequential: transfer 10 MB on average as continuous data.
 - ◆ Random: transfer 10 KB on average.
 - ◆ Average seek time: 7 ms
 - ◆ Average rotational delay: 3 ms
 - ◆ Transfer rate of disk: 50 MB/s

- ▣ Results: R is less than 1 MB/s, S is almost 50

- $$\text{◆ } S = \frac{\text{Amount of Data}}{\text{Time to access}} = \frac{10 \text{ MB}}{210 \text{ ms}} = 47.62 \text{ MB /s}$$

- $$\text{◆ } R = \frac{\text{Amount of Data}}{\text{Time to access}} = \frac{10 \text{ KB}}{10.195 \text{ ms}} = 0.981 \text{ MB /s}$$

RAID Level 1

- RAID Level 1 is mirroring
 - Copy more than one of each block in the system.
 - Copy block places on a separate disk to tolerate the disk failures.

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Simple RAID-1: Mirroring

RAID Level 4

- RAID Level 4 is to add redundancy to a disk array as **parity**.

* P: Parity

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	0	1	1	P0
2	2	3	3	P1
4	4	5	5	P2
6	6	7	7	P3

Simple RAID-4 with parity

RAID Level 4 (Cont.)

- The simple RAID Level 4 optimization known as a **Full-stripe write**.
 - ◆ Calculate the new value of P0 (Parity 0)
 - ◆ Write all of the blocks to the five disks above in parallel
 - ◆ Full-stripe writes are the most efficient way

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Full-stripe Writes In RAID-4

RAID Level 4 (Cont.)

- An example of writes in RAID-4 using the subtractive method.
 - ◆ For each write, the RAID perform **4 physical I/O**. (two read and writes)

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+1
8	9	10	11	P2
12	*13	14	15	+P3

Writes To 4, 13 And Respective Parity Blocks.

Small write problem happens

RAID Level 5

- RAID Level 5 is solution of small write problem.
 - small write problem cause parity-disk bottleneck of RAID Level 4).
 - works almost identically to RAID-4, except that it rotates the parity blocks across drives.
- RAID Level 5's Each stripe is now rotated across the disks.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID-5 with Rotated Parity

RAID Level Analysis

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	N	N/1	N-1	N-1
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	NS	(N/2)S	(N-1)S	(N-1)S
Sequential Write	NS	(N/2) S	(N-1)S	(N-1)S
Random Read	NR	NR	(N-1)R	NR
Random Write	NR	(N/2)R	$\frac{1}{2}$ R	$\frac{N}{4}$ R
Latency				
Read	D	D	D	D
Write	D	D	2D	2D

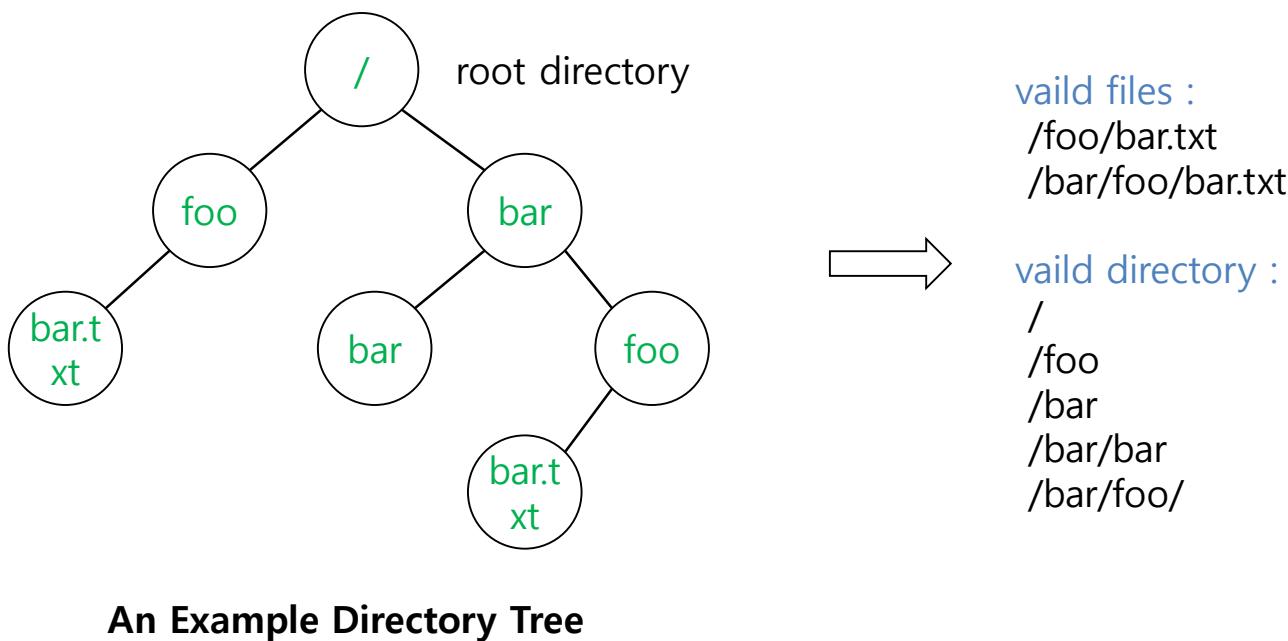
39. File and Directories

File

- File is simply a linear array of bytes.
- Each file has low-level name as 'inode number'

Directory

- Directory is like a file, also has a low-level name.
 - It contains a list of (user-readable name, low-level name) pairs.
 - Each entry in a directory refers to either files or other directories.



File system interface

- ❑ Let's understand file system interface in more detail.
 - ◆ creating, accessing, and deleting files.

Creating Files

- ▣ Use `open` system call with `O_CREAT` flag.
 - ◆ `O_CREAT` : create file.
 - ◆ `O_WRONLY` : only write to that file while opened.
 - ◆ `O_TRUNC` : make the file size zero (remove any existing content).
- ▣ `open` system call returns `file descriptor`.
 - ◆ file descriptor is an integer, is used to access files.
 - ◆ Ex) `read (file descriptor)`

Reading and Writing Files

- ❑ An Example of reading and writing 'foo' file

```
prompt> echo hello > foo    //save the output to the file foo
prompt> cat foo              //dump the contents to the screen
hello
prompt>
```

How does the cat program access the file foo ?

We can use strace to trace the system calls made by a program.

Reading and Writing Files (Cont.)

- The result of strace to figure out cat is doing.

```
prompt> stace cat foo //strace to figure out what cat is doing
...
open("foo", O_RDONLY|O_LARGEFILE)  = 3
read(3, "hello\n", 4096)          = 6 /*the number of bytes to read*/
write(1, "hello\n", 6)            = 6
hello
read(3, "hello\n", 4096)          = 0
write(1, "hello\n", 6)            = 0
..
prompt>
```

- ◆ `open()`: open file for reading with `O_RDONLY` and `O_LARGEFILE` flags.
 - returns file descriptor 3 (0,1,2, is for standard input/output/error)
- ◆ `read()`: read bytes from the file.
- ◆ `write()`: write buffer to standard output.

Reading and Writing Files (Cont.)

- ❑ How to read or write to a specific offset within a file ?
- ❑ Use `lseek()` system call.

```
off_t lseek(int fd, off_t offset /*location */, int whence);
```

- ◆ Third argument is how the seek is performed.
 - SEEK_SET : to offset bytes.
 - SEEK_CUR: to its current location plus offset bytes.
 - SEEK_END: to the size of the file plus offset bytes.

**`lseek()` change the value of variable within the kernel.
It don't require disk seek operation.**

Writing Immediately with fsync()

- ▣ `write()` : write data to persistent storage, at some point in the future.
 - ◆ Ex) 5 or 30 seconds in the kernel.
- ▣ However, some applications require more than eventual guarantee.
 - ◆ Ex) DBMS requires force writes to disk from time to time.

**Use `fsync()` for a particular file descriptor,
the file system responds by forcing all dirty data to disk.**

Writing Immediately with fsync() (Cont.)

- ▣ **fsync()** : the writes are forced immediately to disk.

```
off_t fsync(int fd /*for the file referred to by the specified file*/)
```

- ▣ An Example of `fsync()`.

```
1 int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);  
2 int rc = write(fd, buffer, size);  
3 rc = fsync(fd);
```

- ◆ If a file is created, it durably a part of the directory.
 - Above code requires `fsync()` to directory also.

Renaming Files

- ❑ `rename()` : rename a file to different name.

- ◆ It implemented as an atomic call.
- ◆ Ex) change from foo to bar

```
prompt > mv foo bar
```

- ❑ The result of strace to figure out what mv is doing.

```
1 int fint fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
2 write(fd, buffer, size); // write out new version of file
3 fsync(fd);
4 close(fd);
5 rename("foo.txt.tmp", "foo.txt");d = open("foo", O_CREAT | O_WRONLY
| O_TRUNC);
6 int rc = write(fd, buffer, size);
7 rc = fsync(fd);
```

Getting Information About Files

- ▣ **stat()** : Show the File metadata
 - ◆ metadata is information about each file Ex) size, permission, ..
- ▣ stat structure is below:

```
1  struct stat {  
2      dev_t st_dev; /* ID of device containing file */  
3      ino_t st_ino; /* inode number */  
4      mode_t st_mode; /* protection */  
5      nlink_t st_nlink; /* number of hard links */  
6      uid_t st_uid; /* user ID of owner */  
7      gid_t st_gid; /* group ID of owner */  
8      dev_t st_rdev; /* device ID (if special file) */  
9      off_t st_size; /* total size, in bytes */  
10     blksize_t st_blksize; /* blocksize for filesystem I/O */  
11     blkcnt_t st_blocks; /* number of blocks allocated */  
12     time_t st_atime; /* time of last access */  
13     time_t st_mtime; /* time of last modification */  
14     time_t st_ctime; /* time of last status change */  
15 };
```

Getting Information About Files (Cont.)

- ❑ An example of `stat()`
 - ◆ All information is in a inode

```
prompt> echo hello > file  
prompt> stat file
```

```
File: 'file'  
Size: 6 Blocks: 8 IO Block: 4096 regular file  
Device: 811h/2065d Inode: 67158084 Links: 1  
Access: (0640/-rw-r-----)Uid: (30686/ root) Gid: (30686/ remzi)  
Access: 2011-05-03 15:50:20.157594748 -0500  
Modify: 2011-05-03 15:50:20.157594748 -0500  
Change: 2011-05-03 15:50:20.157594748 -0500
```

Removing Files

- ▣ The result of strace to figure out what rm is doing.
 - ◆ rm is Linux command to remove a file
 - ◆ rm call `unlink()` to remove a file.

```
1  prompt> strace rm foo
2
3  unlink("foo")
4  ...
5  prompt>
```

**Why it calls `unlink()`? not “remove or delete”
We can get the answer later.**

Making Directories

▣ `mkdir()`: Make a directory

- ◆ When a directory is created, it is `empty`.
- ◆ Empty directory have two entries: `.` (itself), `..`(parent)

```
1  prompt> ls -al
2  total 8
3  drwxr-x--- 2 root root 6 Apr 30 16:17 ./
4  drwxr-x--- 26 root root 4096 Apr 30 16:17 ../
```

Reading Directories

- ▣ A sample code to read directory entries.

```
1 int main(int argc, char *argv[]) {
2     DIR *dp = opendir(".");
3     assert(dp != NULL);
4     struct dirent *d;
5     while ((d = readdir(dp)) != NULL) {
6         printf("%d %s\n", (int)d->d_ino, d->d_name);
7     }
8     closedir(dp);
9     return 0;
10 }
```

Deleting Directories

- ❑ `rmdir()`: Delete a directory.

- ◆ `rmdir()` requires directory be empty before it deleted.
 - ◆ If you call `rmdir()` to a non-empty directory, it will fail.

Hard Links

- ▣ `link()`: Link old file and a new file.
 - ◆ Create hard link named file2.

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2 /* create a hard link, link file to file2 */
prompt> cat file2
hello
```

- ▣ The result of `link()`
 - ◆ Two files have same inode number, but two human name(file, file2)

```
prompt> ls -i file file2
67158084 file /* inode value is 67158084 */
67158084 file2 /* inode value is 67158084 */
prompt>
```

Hard Links (Cont.)

- ▣ How to create hard link file?
 - ◆ **Step1.** Make a **inode**, track all information about the file.
 - ◆ **Step2.** **Link** a human-readable name to file.
 - ◆ **Step3.** Put link file into a current directory.
- ▣ After creating a hard link to file, old and new files have no difference.
- ▣ Thus, to remove a file, we call **unlink()**

unlink Hard Links

- ▣ What `unlink()` is doing ?
 - ◆ Check reference count within the inode number.
 - ◆ Remove link between human-readable name and inode number.
 - ◆ Decrease deference count
 - When only it reaches zero, It delete a file (free the inode and related blocks)

unlink Hard Links (Cont.)

- ▣ The result of unlink()

```
prompt> echo hello > file          /* create file*/
prompt> stat file
... Inode: 67158084 Links: 1 ...
prompt> ln file file2             /* hard link file2 */
prompt> stat file
... Inode: 67158084 Links: 2 ...
prompt> stat file2
... Inode: 67158084 Links: 2 ...
prompt> ln file2 file3            /* hard link file3 */
prompt> stat file
... Inode: 67158084 Links: 3 ...
prompt> rm file                  /* remove file */
prompt> stat file2
... Inode: 67158084 Links: 2 ...
prompt> rm file2                /* remove file2 */
prompt> stat file3
... Inode: 67158084 Links: 1 ...
prompt> rm file3
```

Symbolic Links

- **Symbolic link** is more useful than Hard link
 - ◆ Hard Link cannot create to a directory.
 - ◆ Hard Link cannot create to a file to other partition.

- An example of symbolic link

```
prompt> echo hello > file
prompt> ln -s file file2 /* option -s : create a symbolic link, */
prompt> cat file2
hello
```

Symbolic Links (Cont.)

- ▣ What is different between **Symbolic link** and **Hard Link**?
 - ◆ Symbolic link is different file type.
 - ◆ The size of Symbolic link is always **4 bytes**.
 - Points to original file.
- ▣ An example of symbolic link to see the information.

```
prompt> ls -al
drwxr-x--- 2 remzi remzi 29 May 3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May 3 15:14 ../          /* directory */
-rw-r----- 1 remzi remzi 6 May 3 19:10 file           /* regular file */
lrwxrwxrwx 1 remzi remzi 4 May 3 19:10 file2 -> file /* symbolic link */
```

Symbolic Links (Cont.)

- When remove a original file, symbolic link points noting.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

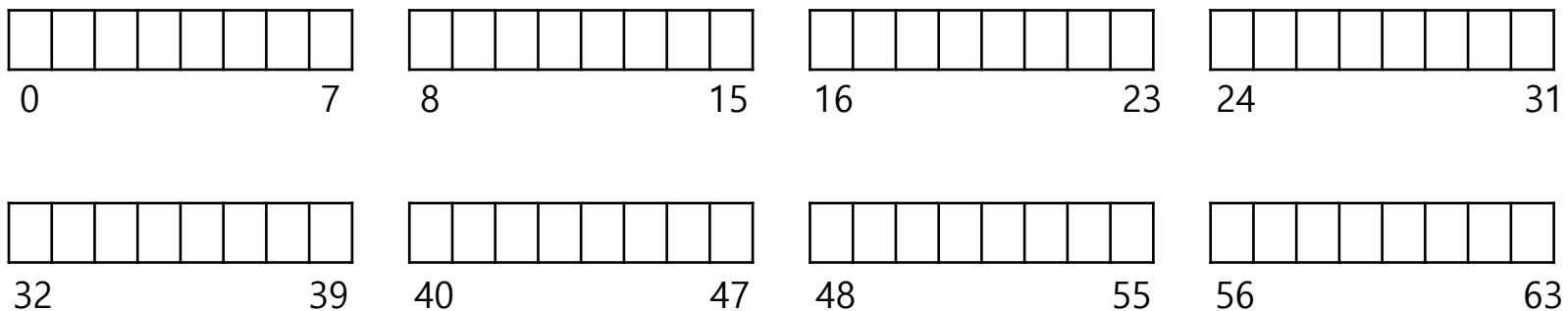
40. File system Implementation.

File system Implementation

- ▣ What types of **data structures** are utilized by the file system?
- ▣ How file system organize its data and metadata?
- ▣ Understand access methods of a file system.
 - ◆ `open()`, `read()`, `write()`, etc.

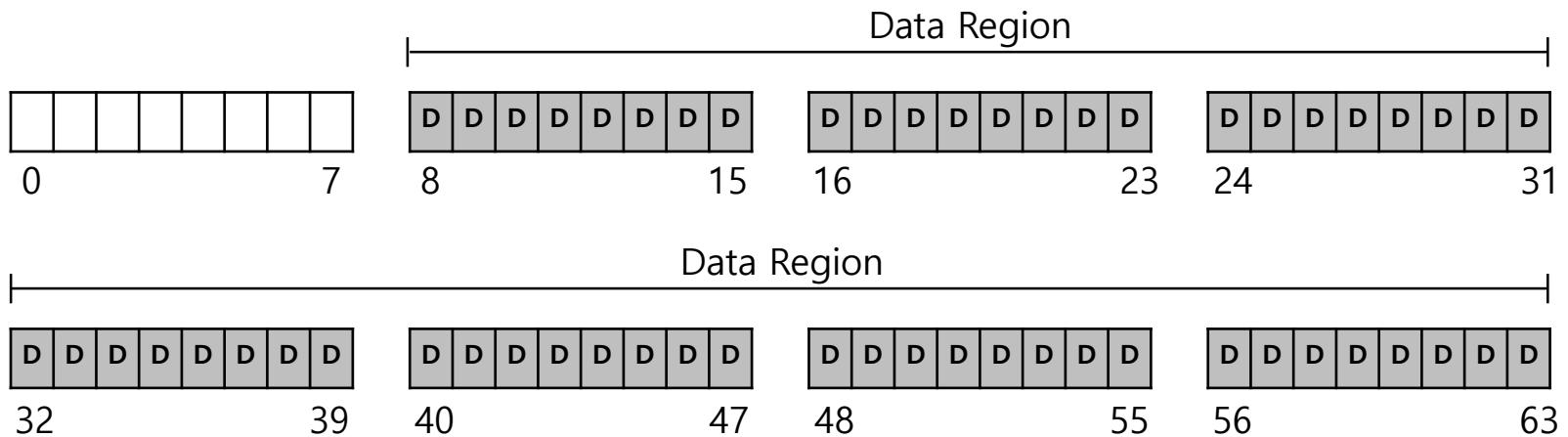
Overall Organization

- Let's develop the overall organization of the file system data structure.
- Divide the disk into **blocks**.
 - Block size is 4 KB.
 - The blocks are addressed from 0 to $N - 1$.



Data region in file system

- Reserve **data region** to store user data



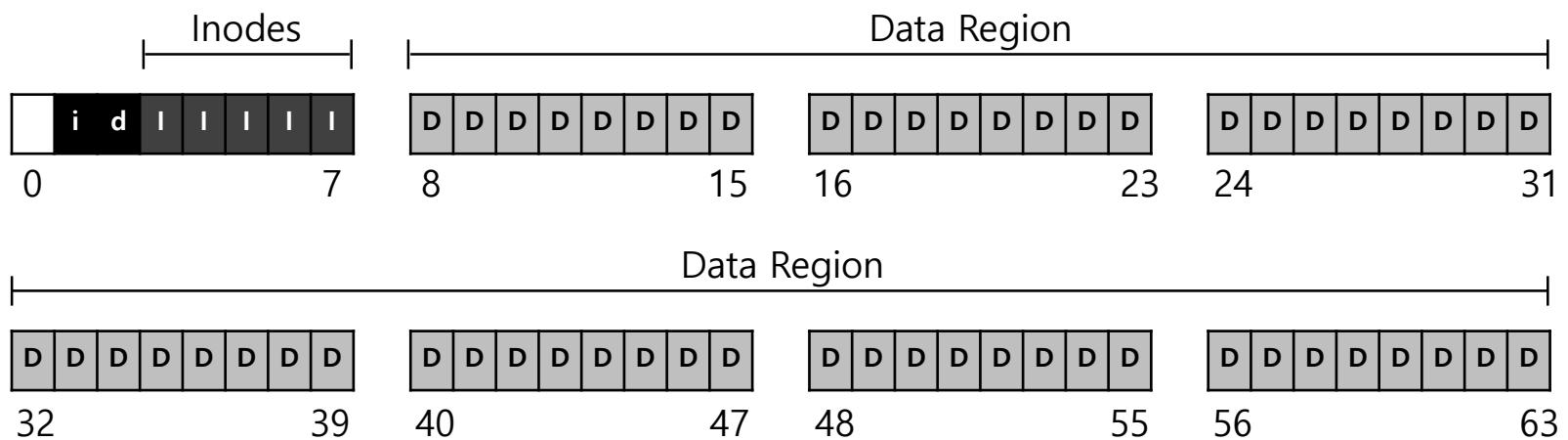
- File system has to track which data block comprise a file, the size of the file, its owner, etc.

How we store these inodes in file system?

Inode table in file system

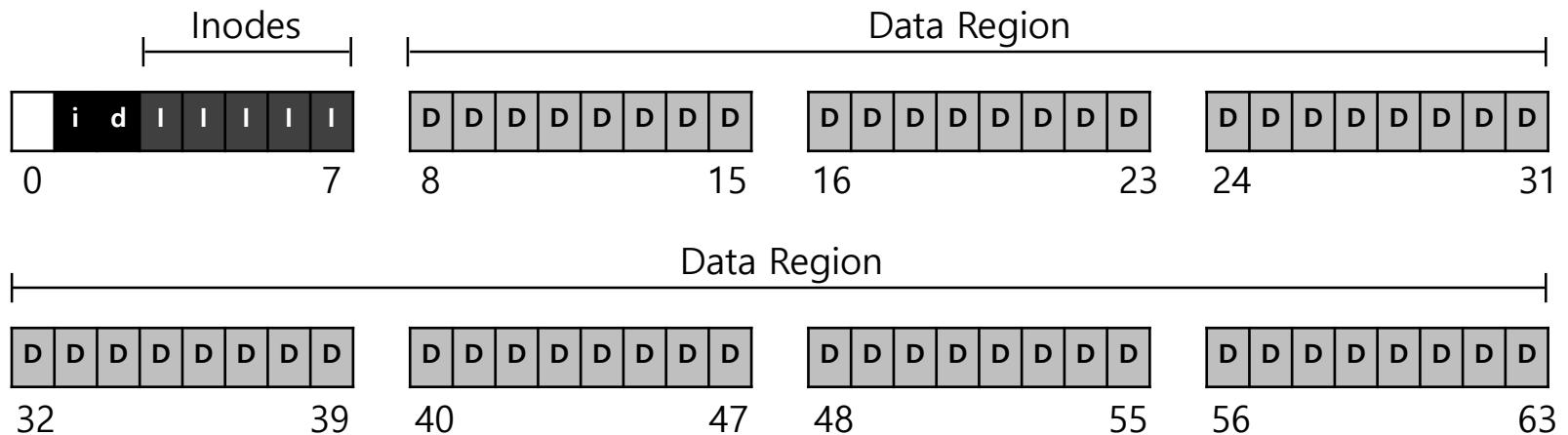
- Reserve some space for **inode table**

- This holds an array of on-disk inodes.
- Ex) inode tables : 3 ~ 7, inode size : 256 bytes
 - 4-KB block can hold 16 inodes.
 - The file system contains 80 inodes. (maximum number of files)



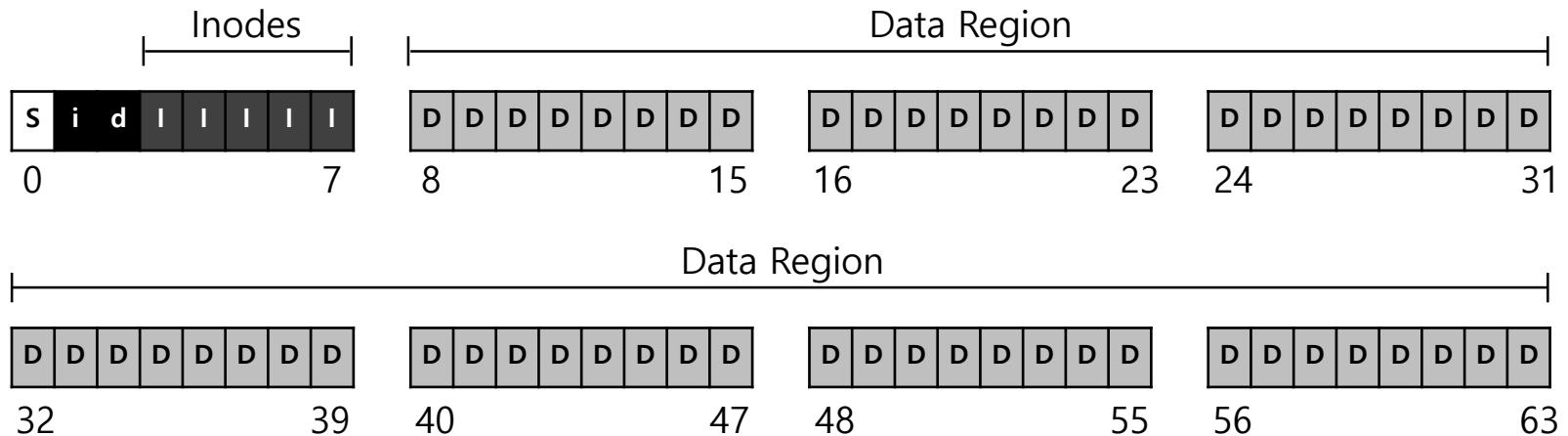
allocation structures

- ▣ This is to track whether inodes or data blocks are free or allocated.
- ▣ Use **bitmap**, each bit indicates free(0) or in-use(1)
 - ◆ data bitmap: for data region for data region
 - ◆ inode bitmap: for inode table



super block

- Super block contains this **information** for **particular file system**
 - Ex) The number of inodes, begin location of inode table. etc



- Thus, when mounting a file system, OS will read the superblock first, to initialize various information.

File Organization: The inode

- Each inode is referred to by inode number.
 - by inode number, File system calculate where the inode is on the disk.
 - Ex) inode number: 32
 - Calculate the offset into the inode region ($32 \times \text{sizeof(inode)}$) (256 bytes) = 8192
 - Add start address of the inode table(12 KB) + inode region(8 KB) = 20 KB

The Inode table

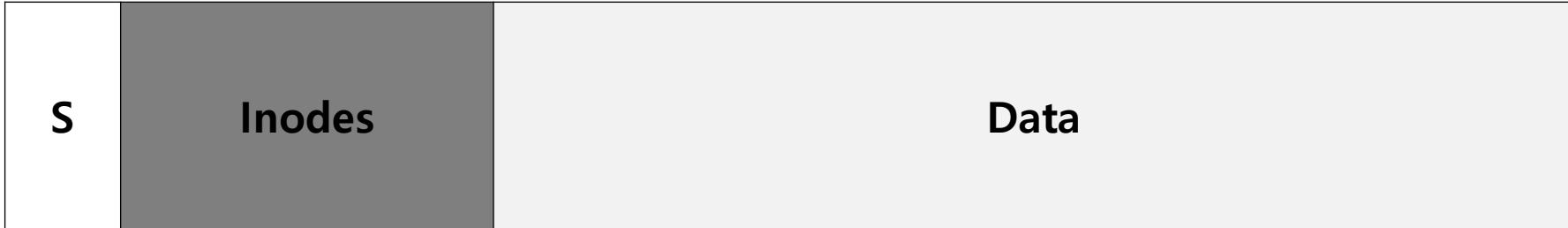
			iblock 0				iblock 1				iblock 2				iblock 3				iblock 4			
Super	i-bmap	d-bmap	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67
			4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71
			8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75
			12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79
0KB	4KB	8KB	12KB	16KB	20KB	24KB	28KB	32KB														

File Organization: The inode (Cont.)

- ▣ Disk are not byte addressable, sector addressable.
 - ◆ blk : (inumber * sizeof(inode)) / blocksize
 - ◆ sector : (blk * blocksize) + inodeStratAddr) /sectorsize

41. Locality and The Fast File System

Unix operating system



Data structures

- Simple and supports the basic abstractions.
- Easy to use file system.

However, Poor Performance !

Problem of Unix operating system

- Unix file system treated the disk as a **random-access memory**.
- Example of random-access blocks with Four files.
 - ◆ Data blocks for each file can accessed by going back and forth the disk, because they are are **contiguous**.

A1	A2	B1	B2	C1	C2	D1	D2
----	----	----	----	----	----	----	----

- ◆ File b and d is deleted.

A1	A2			C1	C2		
----	----	--	--	----	----	--	--

- ◆ File E is created with free blocks. (**spread across** the block)

A1	A2	E1	E2	C1	C2	E3	E4
----	----	----	----	----	----	----	----

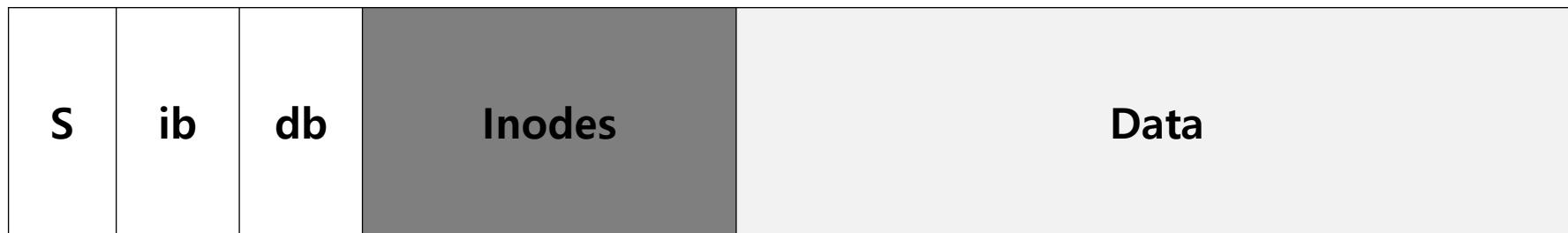
FFS: Disk Awareness is the solution

- ▣ FFS is **Fast File system** designed by a group at Berkeley.
- ▣ The design of FFS is that file system structures and allocation policies to be “disk aware” and improve performance.
 - ◆ Keep same API with file system. (`open()`, `read()`, `write()`, etc)
 - ◆ Changing the internal implementation.

Organizing Structure: The Cylinder Group

- ▣ FFS divides the disk into a bunch of groups. (**Cylinder Group**)
 - ◆ Modern file system call cylinder group as block group.
- ▣ These groups are used to improve seek performance.
 - ◆ By placing two files within the same group.
 - ◆ Accessing one after the other **will not be long seeks** across the disk.
 - ◆ FFS needs to allocate files and directories within each of these groups.

Organizing Structure: The Cylinder Group (Cont.)



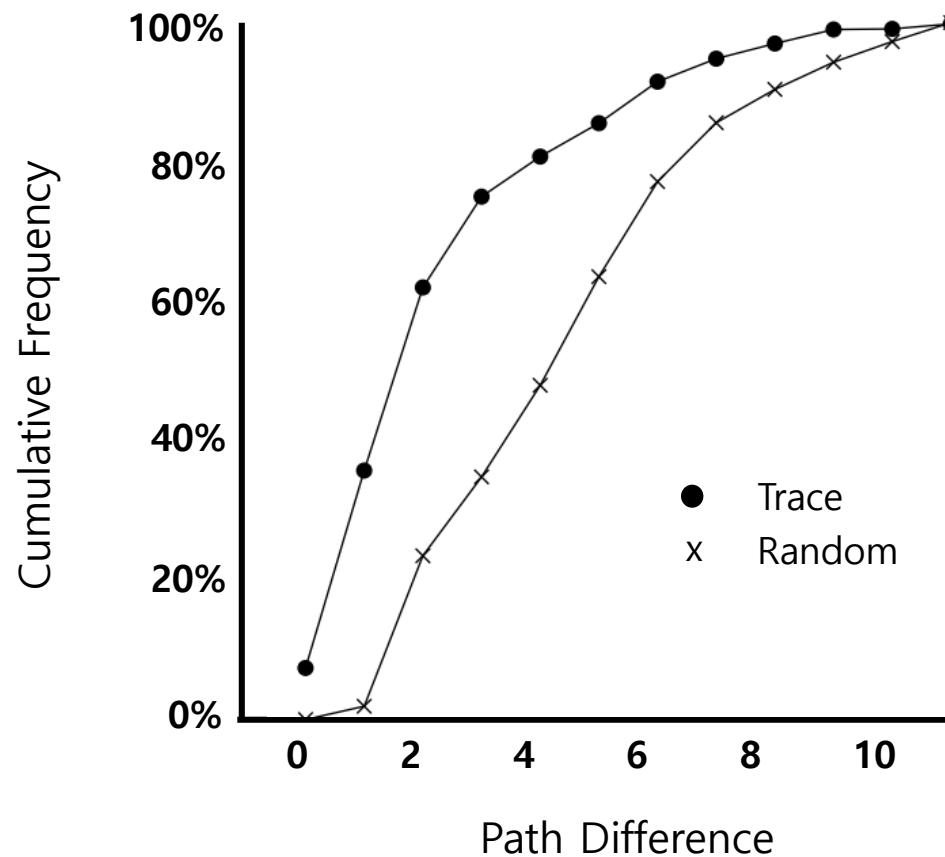
- ▣ Data structure for each cylinder group.
 - ◆ A copy of the super block for reliability reason.
 - ◆ inode bitmap and data bitmap to track free inode and data block.
 - ◆ inodes and data block

How To Allocate Files and Directories?

- ▣ Policy is “**keep related stuff together**”
- ▣ The placement of directories
 - ◆ Find the cylinder group with a low number of allocated directories and a high number of free inodes.
 - ◆ Put the directory data and inode in that group.
- ▣ The placement of files.
 - ◆ Allocate data blocks of a file in the same group as its inode
 - ◆ It places all files in the same group as their directory

FFS Locality for SEER Traces.

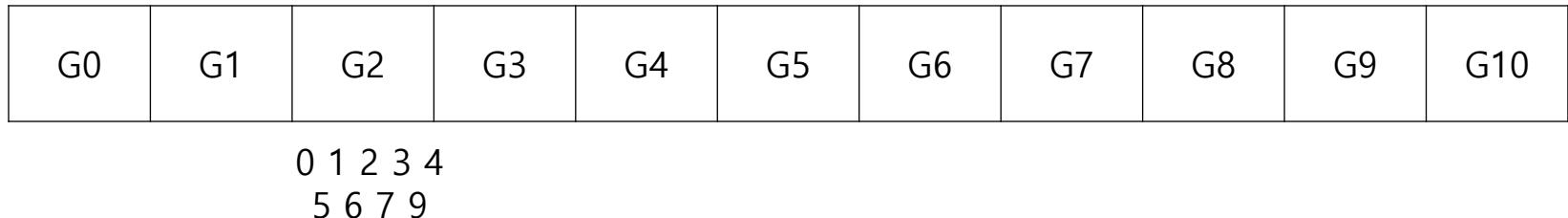
- SEER traces is analysis about how “far away” file accesses were from one another in the directory tree.



The Large-File Exception

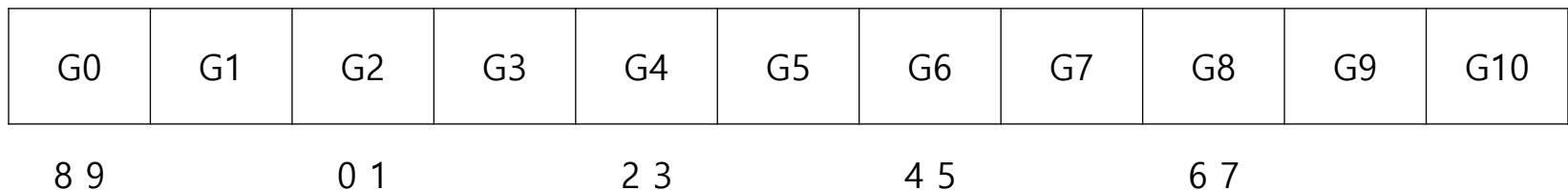
- FFS without the large file exception.

G: block group



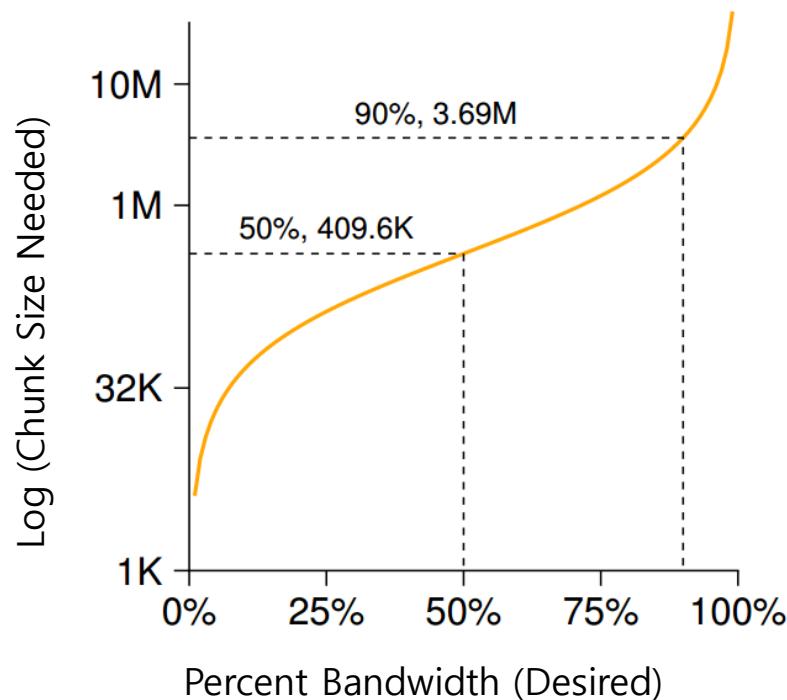
- FFS with the large file exception handling.

- Spread blocks of a file across the disk.
- Consider the time transferring big data from disk and block seek time.



The Large-File Exception (Cont.)

- ❑ How big do chunks have to be?
 - ◆ First twelve direct blocks places in same group as the inode
 - ◆ Indirect block, and all blocks it point places in a different group.



A few other Things about FFS

- ▣ Internal fragmentation.
- ▣ Sub-blocks.
 - ◆ Ex) Create a file with 1 KB : use two sub-blocks, not an entire 4-KB blocks
- ▣ Parameterization.
- ▣ Track buffer.
- ▣ Long file names.
 - ◆ Enabling more expressive names in the file system
- ▣ Symbolic link.

42. Crash Consistency: FSCK and Journaling

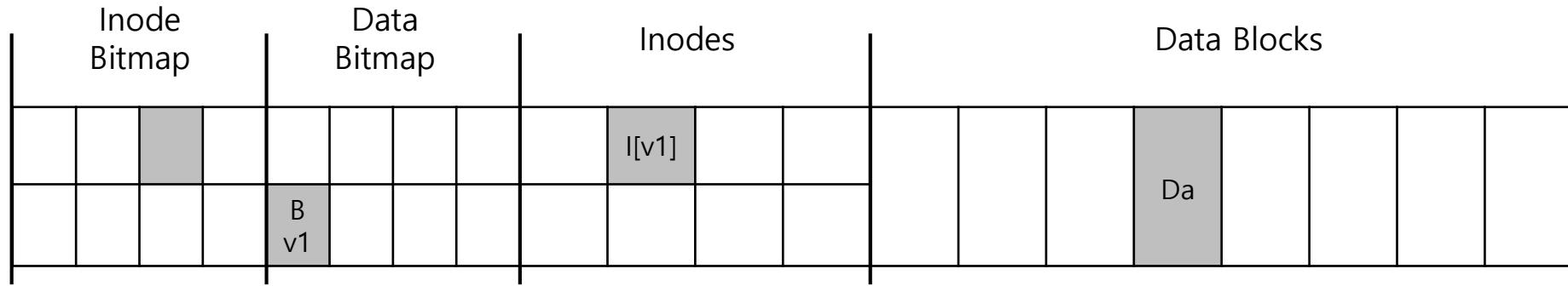
Crash Consistency

- ▣ File system data structures must **persist**
 - ◆ files, directories, all of the other metadata ,etc
- ▣ How to update persistent data structure?
 - ◆ If the system crashes or loses power, on-disk structure will be in **inconsistent** state.

An Example of Crash Consistency

❑ Scenario

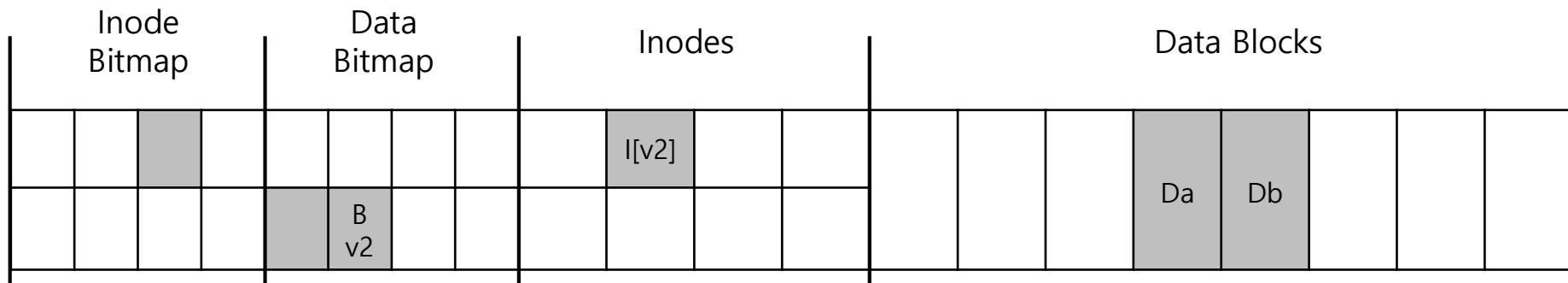
- ◆ Append of a single data block to an existing file.



Before Append a single data block

An Example of Crash Consistency (Cont.)

- ▀ Fist system perform three writes to the disk.
 - ◆ Each of inode I[v2]
 - ◆ Data bitmap B[v2]
 - ◆ Data block (Db)



After Append a single data block

Crash Scenario

- ▣ Only the blow block is written to disk.
 - ◆ Data block (Db)
 - ◆ Update inode (I[v2]) block
 - ◆ Updated bitmap (B[v2])
- ▣ Two writes succeed and the last one fails.
 - ◆ The inode(I[v2]) and bitmap (B[v2]), but not data (Db).
 - ◆ The inode(I[v2]) and data block (Db), but not bitmap(B[v2])
 - ◆ The bitmap(B[v2]) and data block (Db), but not the inode(I[v2])

Crash-consistency problem (consistent- update problem)

Solution

▣ The File System Checker (**fsck**)

- ◆ **fsck** is a Unix tool for finding inconsistencies and repairing them.
- ◆ **fsck** check super block, Free block, Inode state, Inode links, etc.

▣ **Journaling** (or Write-Ahead Logging)

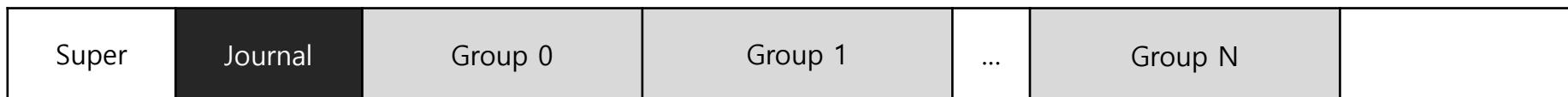
- ◆ Before overwriting the structures in place, write down a little note on the disk, describing what you are to do.
- ◆ Writing this note is the “write ahead”, writing to a structure is log. hence, This is Write-Ahead Logging.

Journaling

- File system reserve some small amount of space within the partition or on another device.



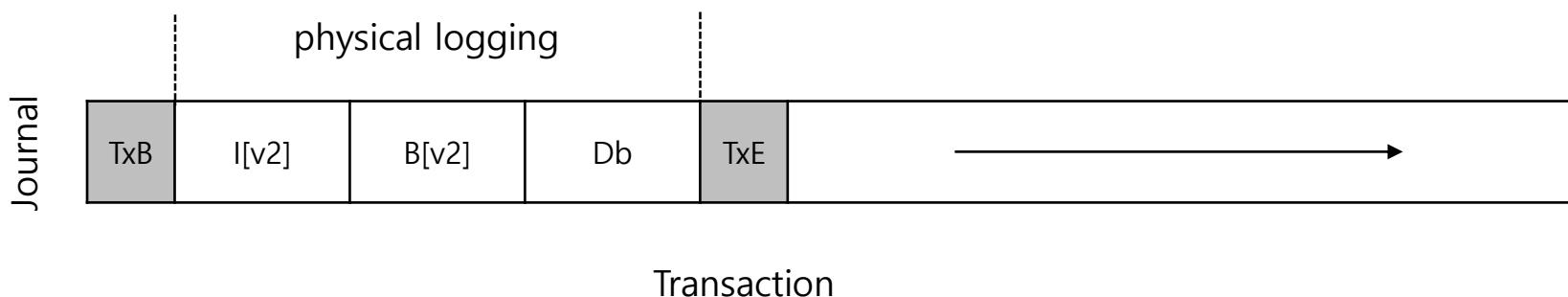
without journaling



with journaling

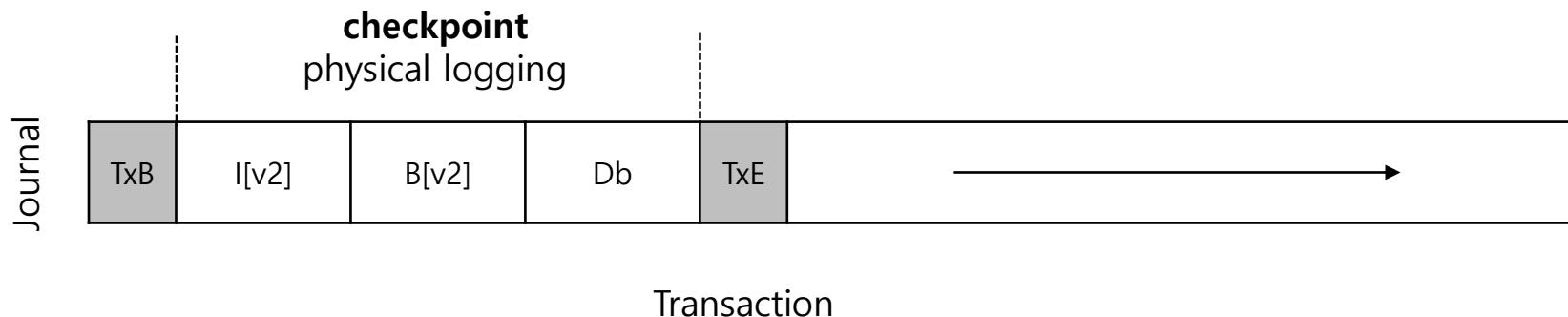
Data Journaling

- ▣ Lets' update a file, following structures is updated.
 - ◆ inode (I[v2]), bitmap (B[v2]), and data block (Db)
- ▣ First, **Journal write**: write the transaction as below.
 - ◆ TxB: Transaction begin block (including transaction identifier)
 - ◆ TxE: Transaction end block
 - ◆ others: contain the exact contents of the blocks



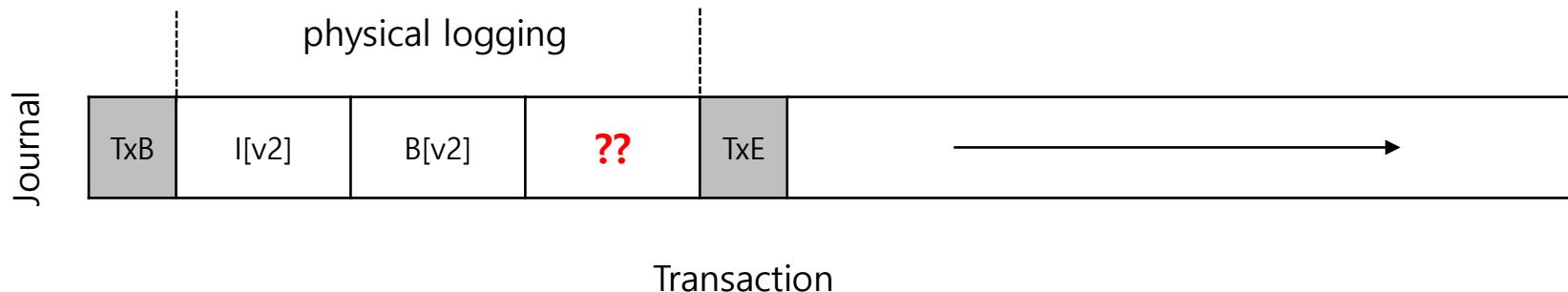
Data Journaling (Cont.)

- Second, **Checkpoint**: Write physical logging to their disk locations



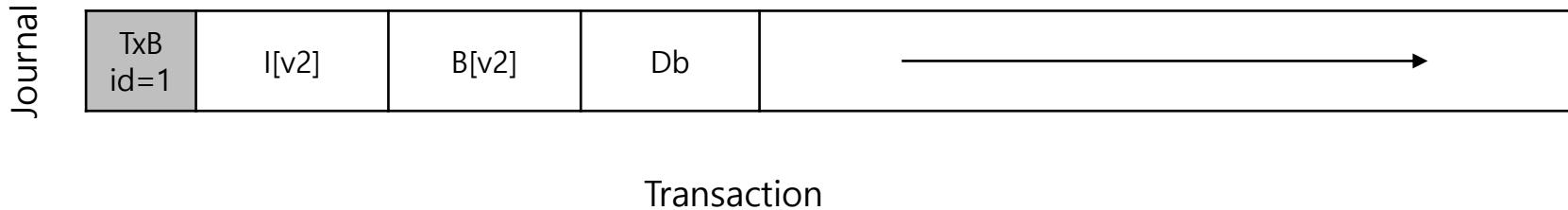
Crash during Data Journaling

- However, If a *crash occurs* during the writes to the journal?

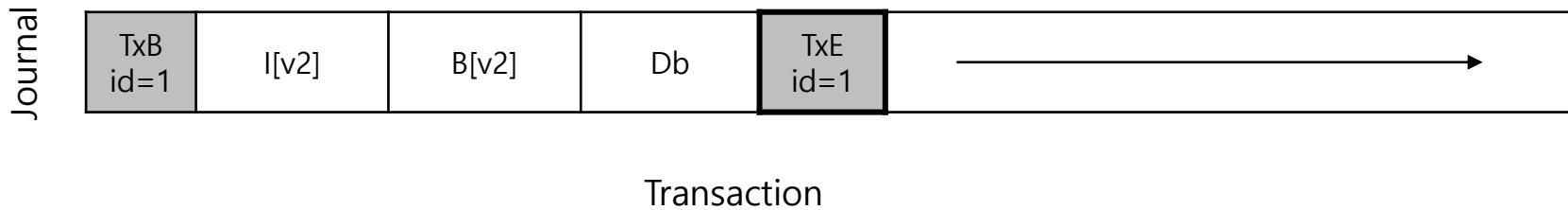


To avoid data being inconsistent

- First, writes all blokes **except the TxE block** to journal.



- Second, The file system issues the write of the TxE.



To avoid data being inconsistent (Cont.)

- **Journal write:** write the contents of the transaction to the log
- **Journal commit:** write the transaction commit block
- **Checkpoint:** write the contents of the update to their locations.

Recovery

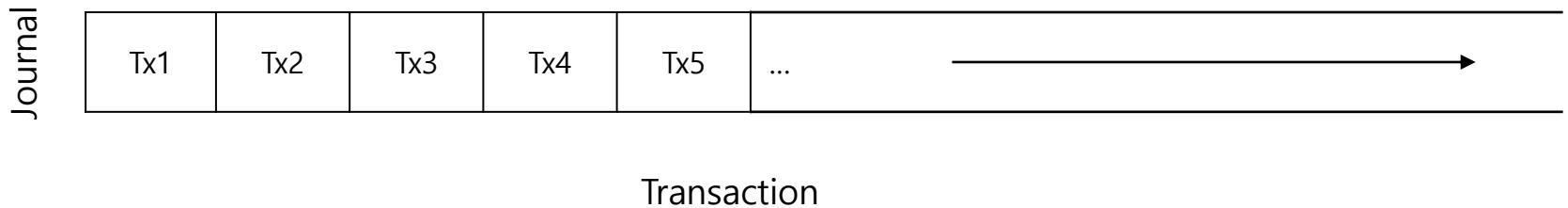
- ▣ If the crash happens, before the transactions is written to the log
 - ◆ The pending update is skipped
- ▣ If the crash happens, after the transactions is written to the log, but **before the checkpoint**
 - ◆ **Recover** the update as follow:
 - Scan the log and **lock for transactions** that have committed to the disk
 - Transactions are **replayed**

Batching Log Updates

- ▣ If we create two files in same directory, same inode, directory entry block is to the log and committed twice.
- ▣ To reduce excessive write traffic to disk, journaling manage the **global transaction**.
 - ◆ Write the content of the global transaction forced by synchronous request.
 - ◆ Write the content of the global transaction after timeout of 5 seconds.

Making The log Finite

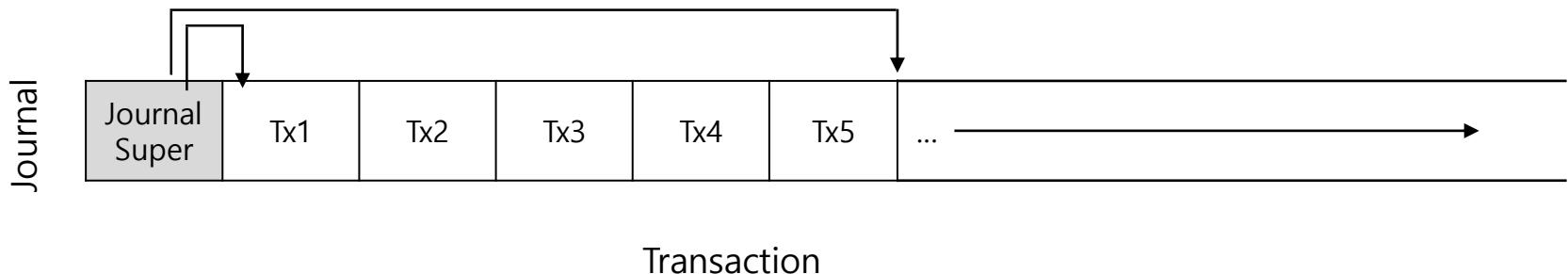
- The log is of a finite size (**circular log**).
 - ◆ To re-using it over and over



Making The log Finite (Cont.)

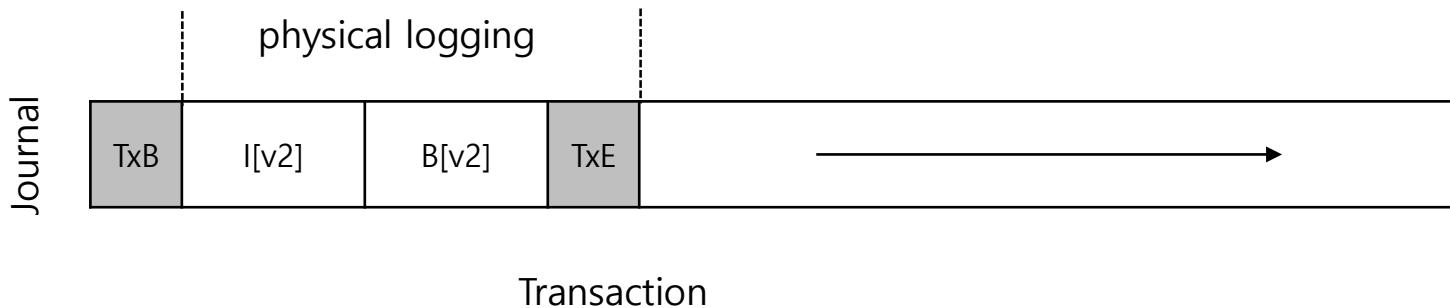
- journal super block

- ◆ Mark the oldest and newest transactions in the log.
- ◆ The journaling system records which transactions have not been check pointed.



Metadata Journaling

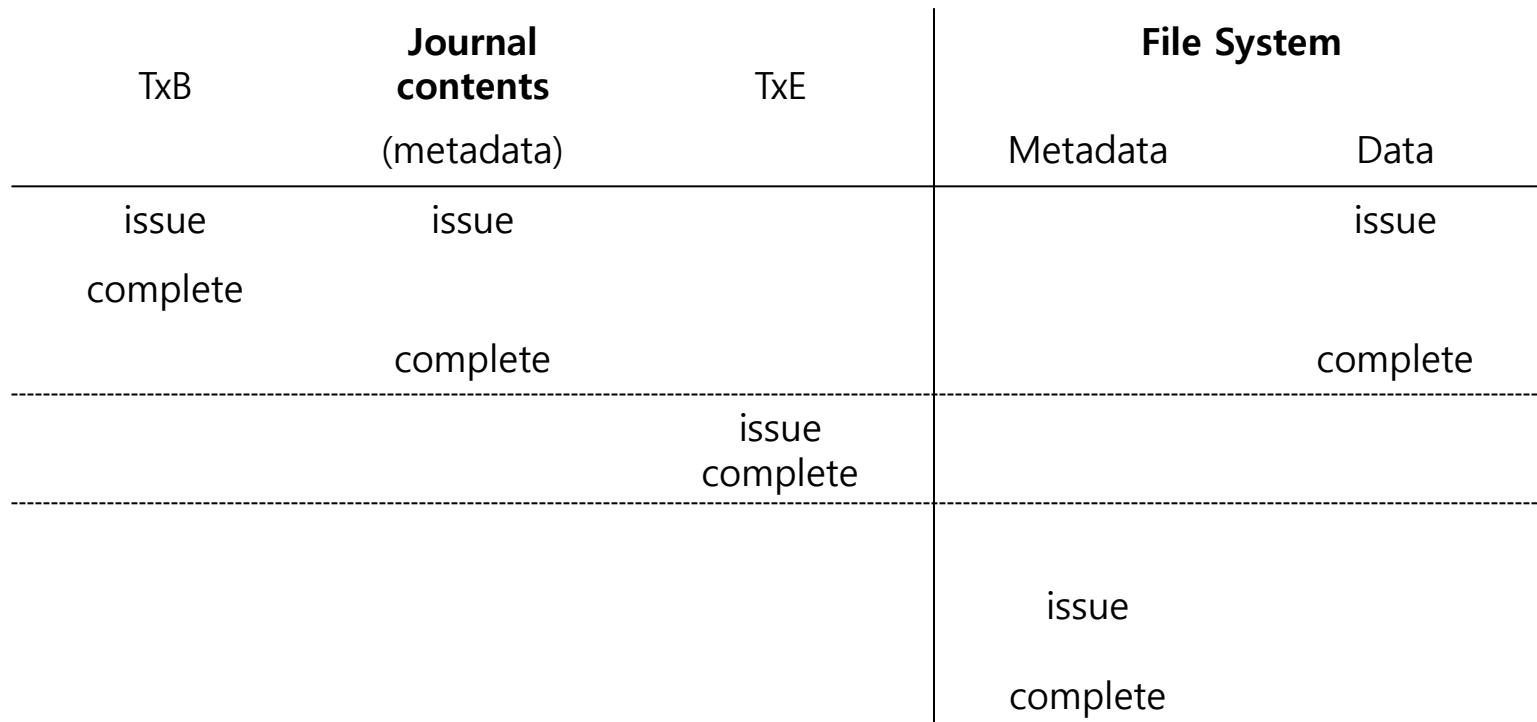
- Because of the high cost of writing every data block to disk twice
 - commit to log (journal)
 - checkpoint to on-disk location.
- File system use ordered journaling (metadata journaling)



Metadata Journaling (Cont.)

- **Data Write:** Write data to final location
- **Journal metadata write:** Write the begin and metadata to the log
- **Journal commit:** Write the transaction commit block to the log
- **Checkpoint metadata:** Write the contents of the metadata to the disk
- **Free:** Later, mark the transaction free in journal super block

Metadata Journaling Timeline



Metadata Journaling Timeline

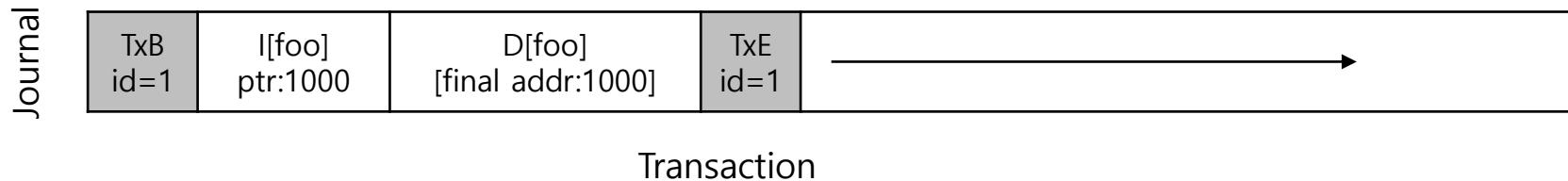
Data Journaling Timeline

Journal contents			File System	
TxB	(metadata)	(data)	TxE	Metadata
issue	issue	issue		
complete				
	complete		complete	
		complete		
			issue	issue
			complete	complete

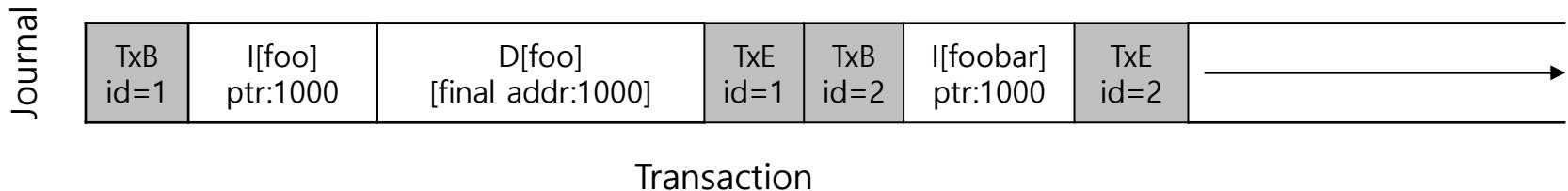
Data Journaling Timeline

Tricky case: Block Reuse

- ☐ Revoke record: some metadata should not be replayed.
- ☐ Scenario.
 - ◆ Directory "foo" is updated.



- ◆ Delete "foo" director, freeing up block 1000 for reuse
- ◆ Create a file "foobar", reusing block 1000 for data



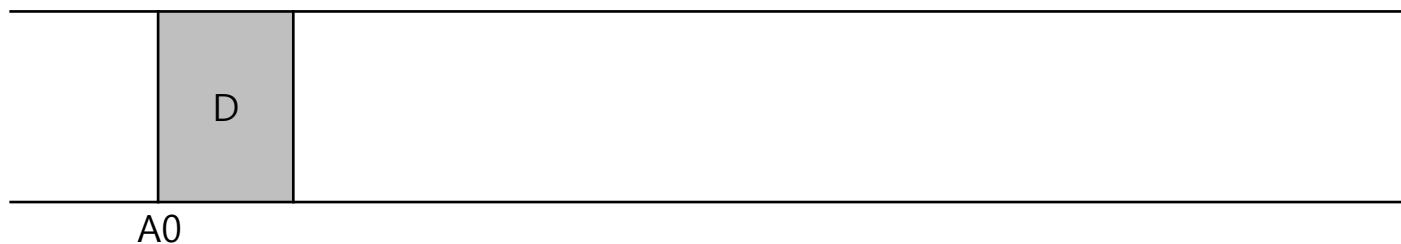
43. Log-structured File Systems

Motivation of Log-structured File System

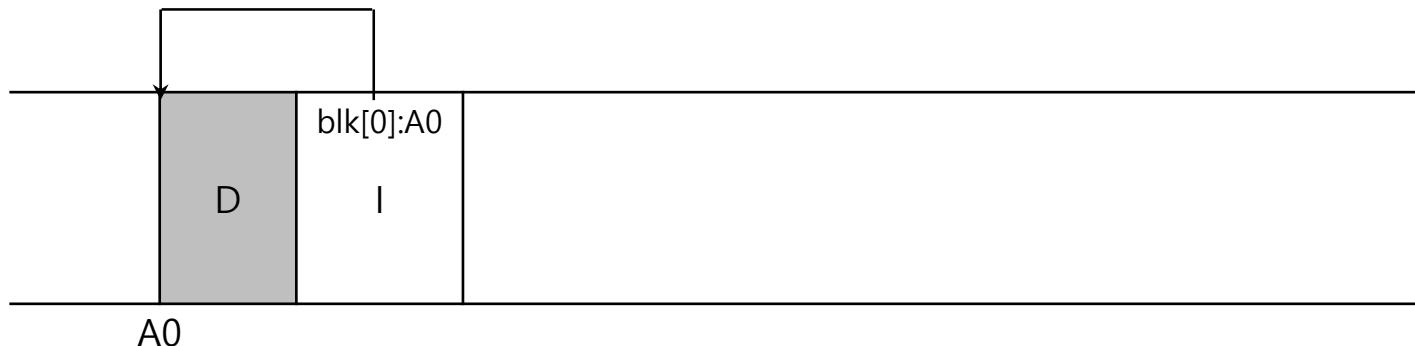
- Memory sizes were growing.
- Large gap between random IO and sequential IO performance.
- Existing File System perform poorly on common workloads.
- File System were not RAID-aware.

Writing to Disk Sequentially

- How do we transform all updates to file-system state into a series of sequential writes to disk?
 - data update

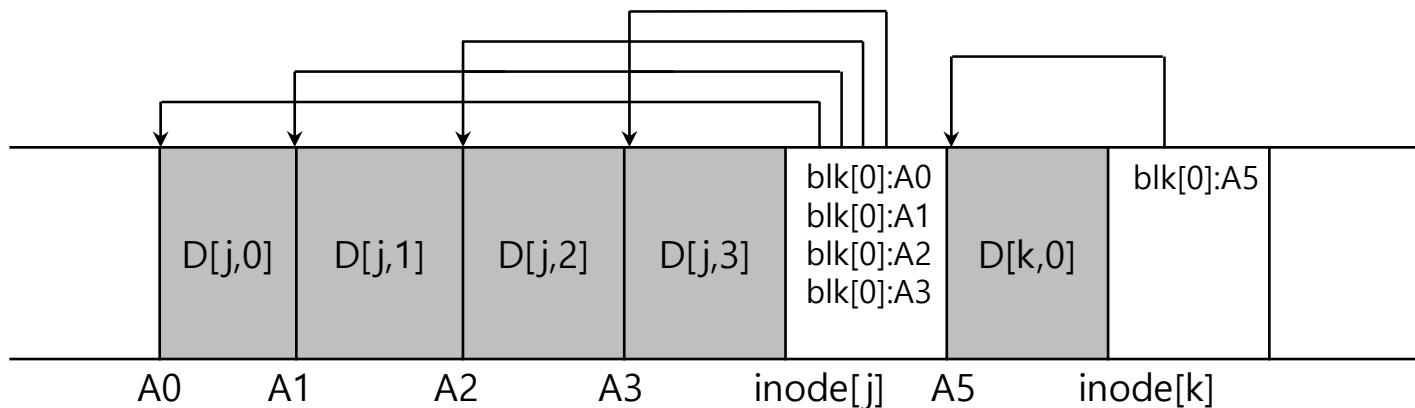


- metadata needs to be updated too. (Ex. inode)



Writing to Disk Sequentially and Effectively

- Write effectively with write buffering.
 - Keeps track of updates in **memory buffer**.
 - Writes them to disk all at once, when it has sufficient number of updates.



How Much to Buffer?

- $T_{write} = T_{position} + \frac{D}{R_{peak}}$ (43.1)

- $R_{effecitve} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}$ (43.2)

- $R_{effecitve} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak}$ (43.3)

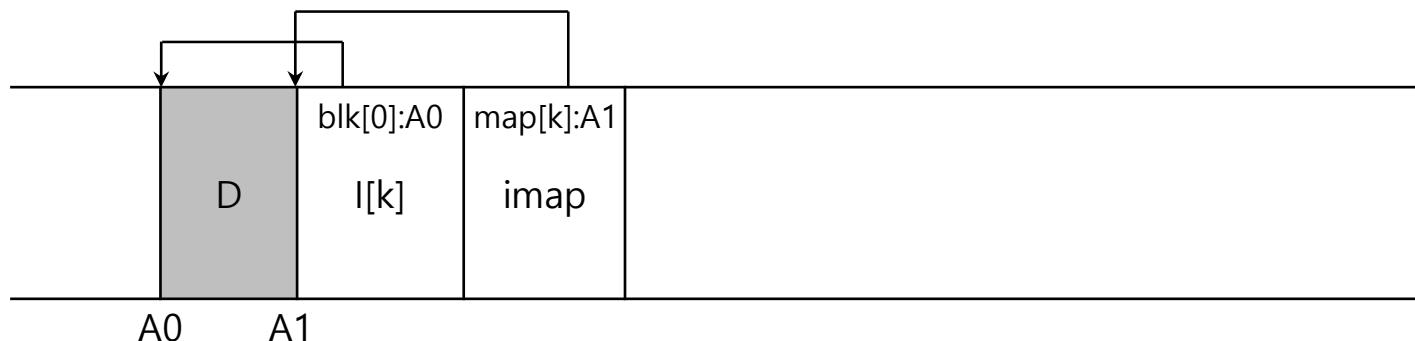
- $D = F \times R_{peak} \times (T_{position} + \frac{D}{R_{peak}})$ (43.4)

- $D = (F \times R_{peak} \times T_{position}) + (F \times R_{peak} \times \frac{D}{R_{peak}})$ (43.5)

- $D = \frac{F}{1-F} \times R_{peak} \times T_{position}$ (43.6)

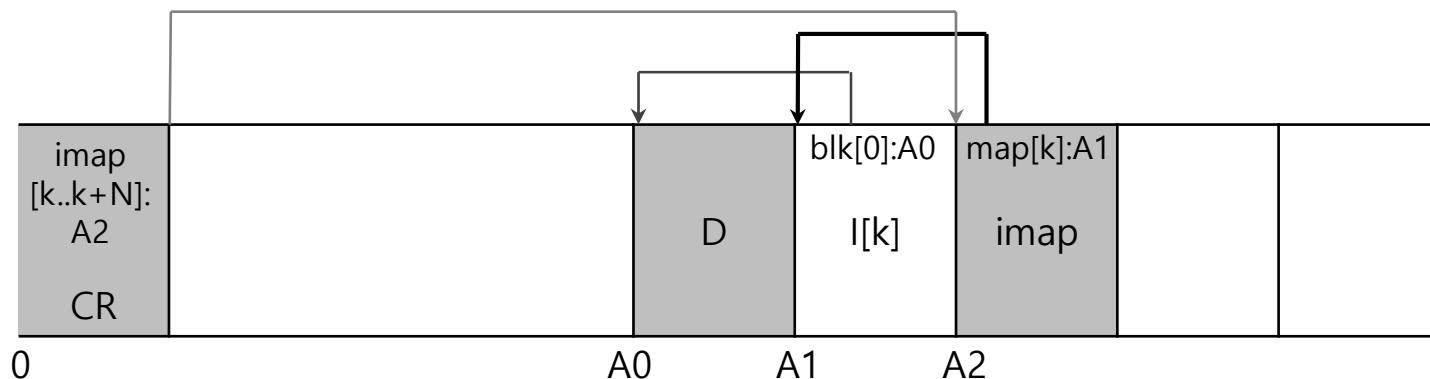
Finding Inode

- ❑ How to find inode in FFS? because FFS splits up the inode table into chunks and places a group of inodes.
- ❑ Solution is through indirection “The Inode Map”



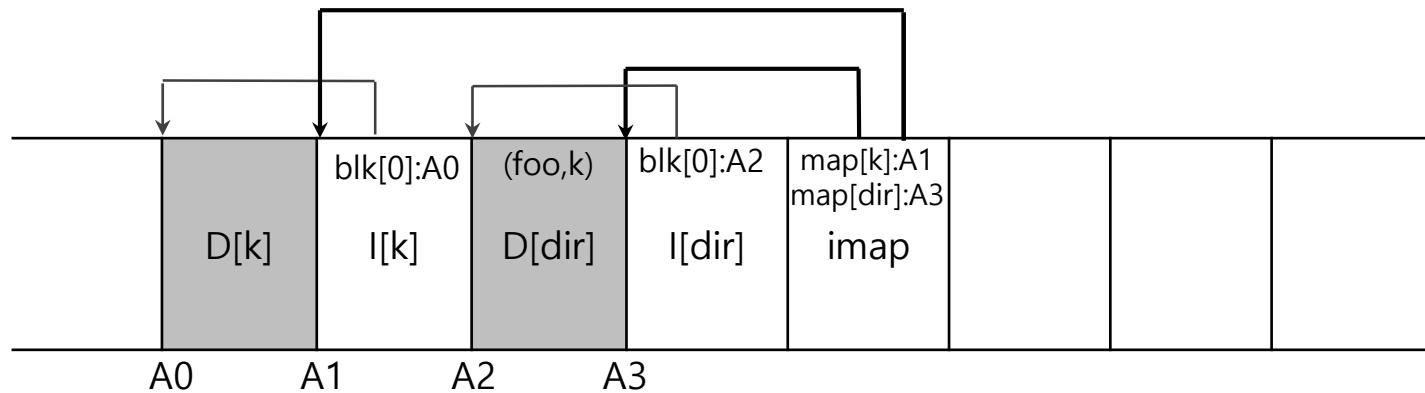
The Checkpoint Region

- How to find the inode map, spread across the disk?
 - ◆ The LFS File system must have fixed location on disk to begin a file lookup
- **Checkpoint Region** contains pointers to the latest of the inode map



What About Directories?

- How LFS store directory data ?



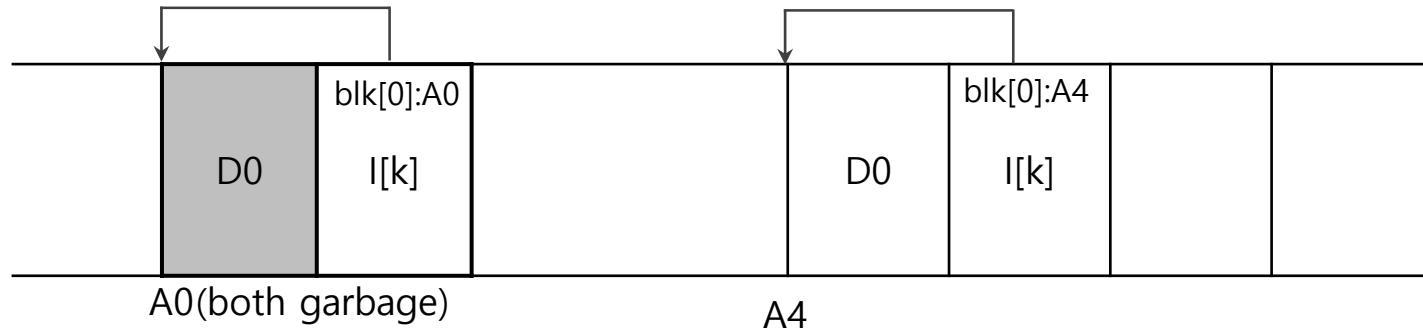
Garbage Collection

- LFS keeps writing newer version of file.
- However LFS leaves the older versions of file structures all over the disk, call as garbage.

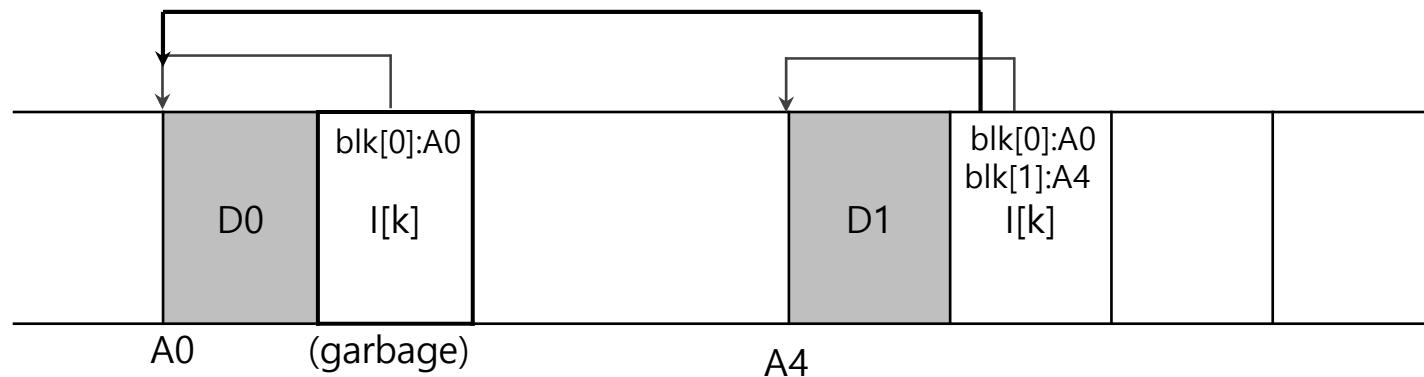
Garbage Collection (Cont.)

- An example of garbage collection.

- ◆ Overwrite the data block:

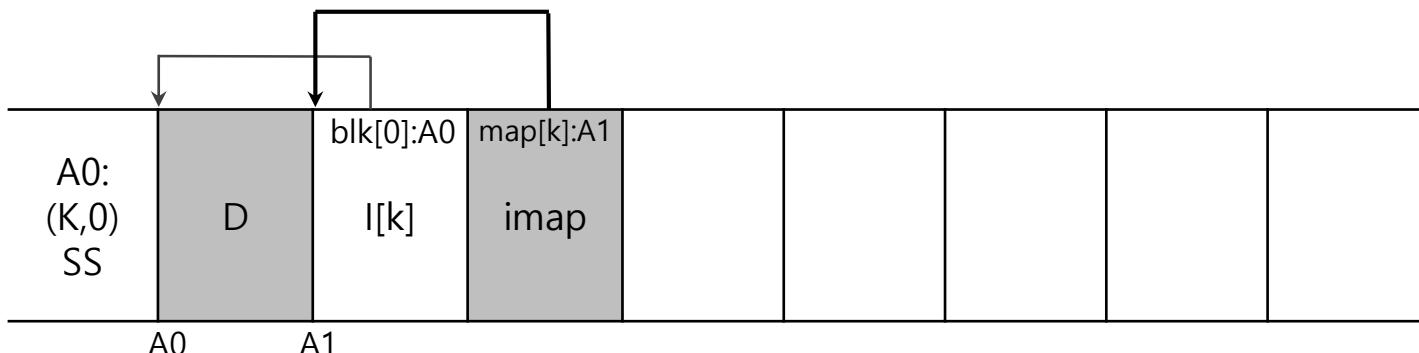


- ◆ Append a block to that original file k:



Determining Block Liveness

- ❑ LFS adds information to each segment that describes each blocks.
 - ◆ Ex: LFS includes for each data block, its inode number and its offset.
- ❑ This information is recorded in **Segment summary block**.



44. Data Integrity and Protection

Disk Failure Modes

- Common and worthy of failures are **frequency of latent-sector errors(LSEs)** and **block corruption**.

	Cheap	Costly
LSEs	9.40%	1.40%
Corruption	0.50%	0.05%

Frequency of LSEs and Block Corruption

Disk Failure Modes (Cont.)

- ▣ Frequency of latent-sector errors(LSEs)
 - ◆ Costly drives with more than one LSE are as likely to develop additional.
 - ◆ For most drives, annual error rate increases in year two
 - ◆ LSEs increase with disk size
 - ◆ Most disks with LSEs have less than 50
 - ◆ Disks with LSEs are more likely to develop additional LSEs
 - ◆ There exists a significant amount of spatial and temporal locality
 - ◆ Disk scrubbing is useful (most LSEs were found this way)

Disk Failure Modes (Cont.)

- Block corruption:

- ◆ Chance of corruption varies greatly across different drive models
- ◆ within the same drive class
- ◆ Age affects are different across models
- ◆ Workload and disk size have little impact on corruption
- ◆ Most disks with corruption only have a few corruptions
- ◆ Corruption is not independent with a disk or across disks in RAID
- ◆ There exists spatial locality, and some temporal locality
- ◆ There is a weak correlation with LSEs

Handling Latent Sector Errors

- ▣ Latent sector errors are easily detected and handled.
- ▣ Using **redundancy mechanisms**:
 - ◆ In a mirrored RAID or RAID-4 and RAID-5 system based on parity, the system should reconstruct the block from the other blocks in the parity group.

Detecting Corruption: The Checksum

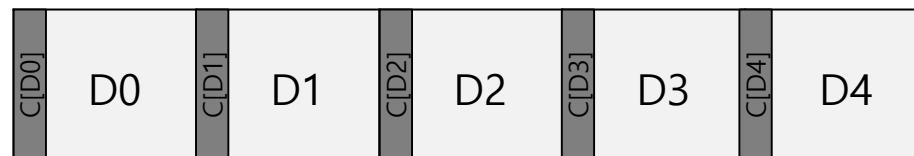
- ▣ How can a client tell that a block has gone bad?
- ▣ Using **Checksum mechanisms**:
 - ◆ This is simple the result of a function that takes a chunk of data as input and computes a function over said data, producing a small summary of the contents of the data.

Checksum Layout

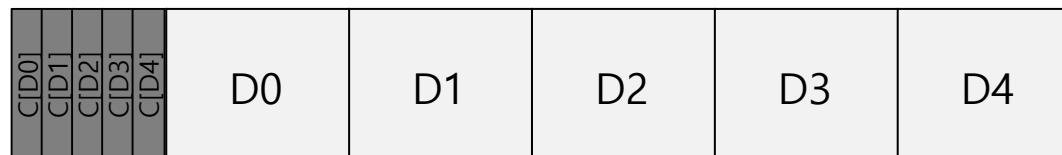
- The disk layout without checksum:



- The disk layout **with checksum**:



- ◆ Store the checksums packed into 512-byte blocks.

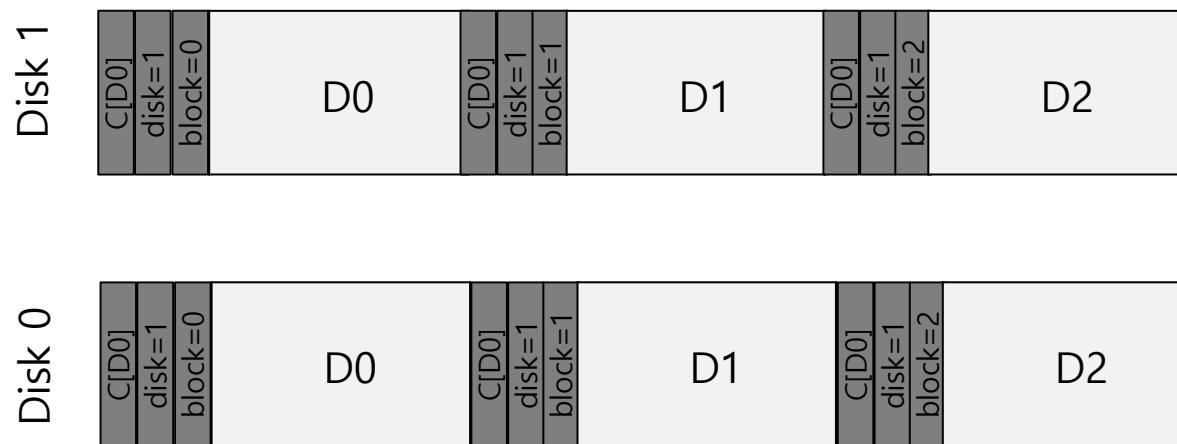


Using Checksums

- ▣ When reading a block D, the client reads its checksum from disk $Cs(D)$, **stored checksum**
- ▣ Computes the checksum over the retrieved block D, **computed checksum** $Cc(D)$.
- ▣ Compares the stored and computed checksums;
 - ◆ if they are equal ($Cs(D) == Cc(D)$), the data is in safe.
 - ◆ If they do not match ($Cs(D) != Cc(D)$), the data has changed since the time it was stored (since the stored checksum reflects the value of the data at that time).

Issues of Checksum

- Misdirected write arises in disk and RAID controllers which write data to disk correctly, except in the *wrong* location.



Issues of Checksum (Cont.)

- ❑ Lost Writes, occurs when the device informs the upper layer that a write has completed but in fact it never is persisted.
- ❑ Scribbling
- ❑ Overheads of Check summing

- Disclaimer: This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.