

COMP 4736

Introduction to Operating Systems

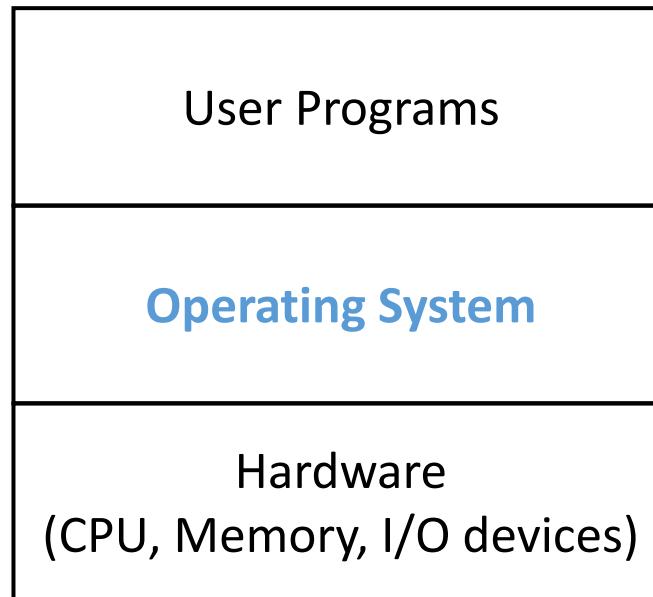
01. Computer Architecture and Overview
(OSTEP Ch. 2 & 4)

Ch. 02. Introduction to Operating Systems

(OSTEP Ch 2)

What is an Operating System?

- Middleware between **user programs** and **system hardware**.
- Manages hardware: CPU, main memory, I/O devices (disk, network card, mouse, keyboard, etc.)



What happens when a program runs?

(Recall: a program is a sequence of instructions and data.)

- The processor **fetches** an instruction from memory, ...
- **decodes** it (i.e., figures out which instruction this is), ...
- and **executes** it (e.g., add two numbers, access memory, check a condition, jump to a function, etc.).
- The processor then moves to the **next instructions**. It repeats until program completes.

Operating Systems (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, etc.
- 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 interfaces (**APIs, standard library**).
- A typical OS exports a few hundred **system calls**.
 - Run programs
 - Access memory
 - Access devices

OS as a Resource Manager

- The OS **manages 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 (cpu.c)

```
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);
17         printf("%s\n", str);
18     }
19     return 0;
20 }
```

Virtualizing the CPU (Result 1)

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

- Runs forever.
- We halt the program by pressing Ctrl+C.

Virtualizing the CPU (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
...
...
```

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

Virtualizing the CPU

- Running multiple programs at once raises new questions.
 - E.g., if two programs want to run at the same time, which **should** run?
- It is handled by **policies** of the OS.
- We will study them as we learn about the basic **mechanisms** that OS implement, as a **resource manager**.

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 (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
10    assert(p != NULL);
11    printf("(%d) address pointed to by p: %p\n",
12          getpid(), p);                  // a2
13    *p = 0;                            // a3
14    while (1) {
15        Spin(1);
16        *p = *p + 1;
17        printf("(%d) p: %d\n", getpid(), *p); // a4
18    }
19    return 0;
20 }
```

Virtualizing Memory (Result 1)

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

- The newly allocated memory is at address 0x200000.
- It updates the value and prints out the result.

Virtualizing Memory (Result 2)

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

- 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 0x200000 independently.

Virtualizing Memory

- 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.

Problems 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 (thread.c)

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

Concurrency (thread.c)

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

- 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 (Result)

- 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** can 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 (io.c)

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

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

Persistence

- 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 overheads** of the OS.
 - OS must strive to provide virtualization without excessive overheads.
- **Protection** between applications
 - **Isolation**: Bad behavior of one does not harm other and the OS itself.

Design Goals

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

Some History

- IBM 701 – Electronic Data Processing Machine (1952)
- No OS. Operated manually.



Batch Processing

- IBM 709 —Data Processing System (1952)
- SHARE Operation System (SOS). Manages buffers and I/O devices.



Multiprogramming

- PDP-11 — minicomputer (1970)
- Unix. Supports multiprogramming.



Personal Computer (PC)

- Apple II (1977) and IBM PC (1981)
- Disk Operating System (DOS) and Mac OS



Modern Era (and Error)



Mobile and IoT



Summary

- We have briefly introduced what an **operating system (OS)** is.
- Included in this course:
 - **Virtualization** of CPU and memory
 - Resource management. Allows multiple programs to run.
 - **Concurrency**
 - Data consistency. Data protection.
- Not included (but important topics nonetheless):
 - **Persistence** via devices and file systems.
 - **Networking**
 - **Graphics devices**
 - **Security**

References

- P. Brinch Hansen, “The Evolution of Operating Systems”, *Classic Operating Systems: From Batch Processing to Distributed Systems*, Springer-Verlag, New York, 2000.
- IBM Archives: IBM 701.
https://www.ibm.com/ibm/history/exhibits/701/701_intro.html
- IBM Archives: 709 Data Processing System
https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP709.html
- A, Hudson, “A brief tour of the PDP-11, the most influential minicomputer of all time”, Ars Technica, Mar 2022.
<https://arstechnica.com/gadgets/2022/03/a-brief-tour-of-the-pdp-11-the-most-influential-minicomputer-of-all-time/7/>

Ch. 04. The Process

(OSTEP Ch. 4)

How to Provide the Illusion of Many CPUs?

- **CPU virtualization**
 - 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**.

Implementing Virtualization

- OS will need low-level machinery and high-level intelligence.
- **Mechanisms**: How to “context switch” between processes.
- **Policies**: Decides which process to run, e.g., a **scheduling policy**.

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

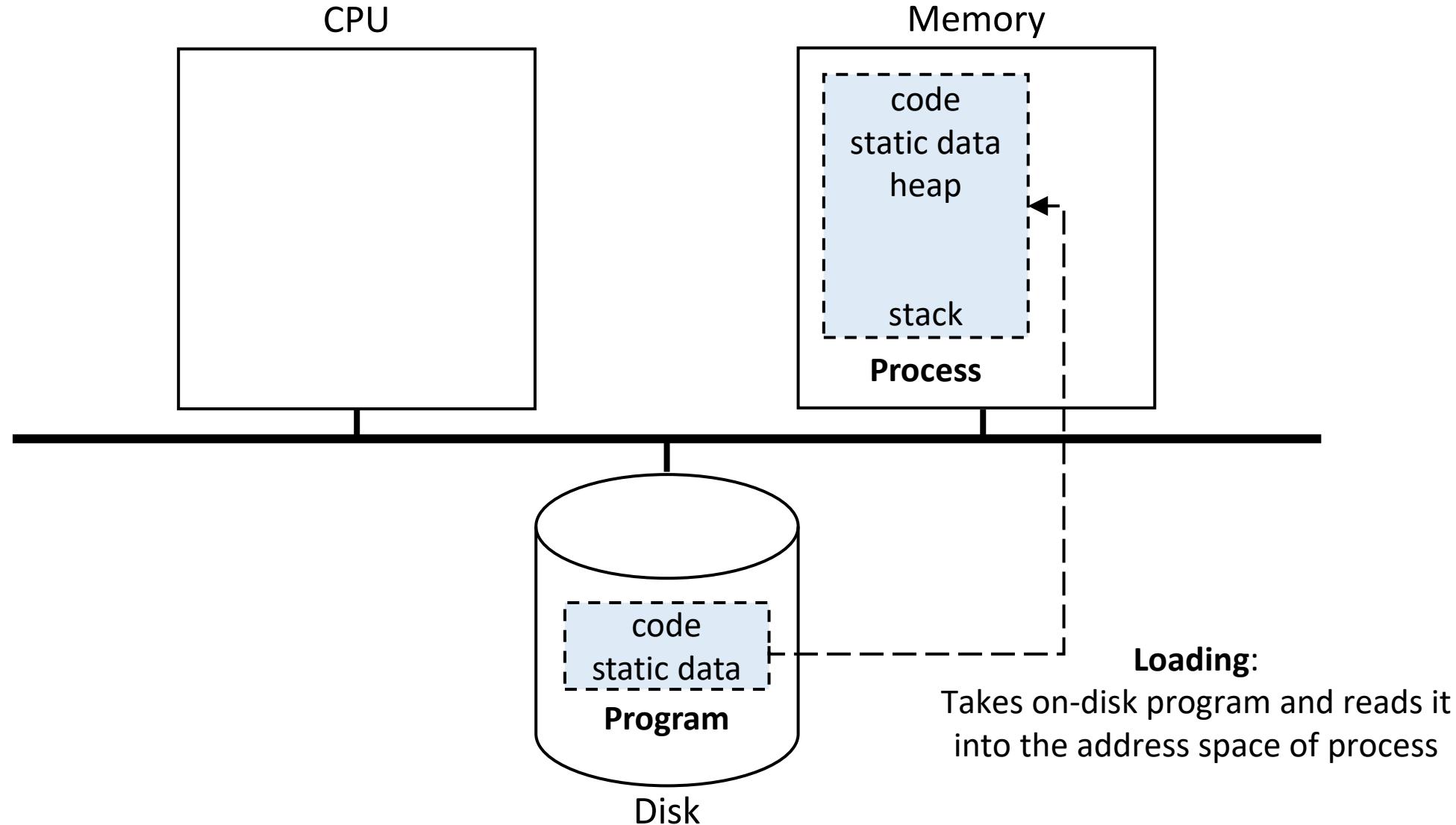
Process Creation

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

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

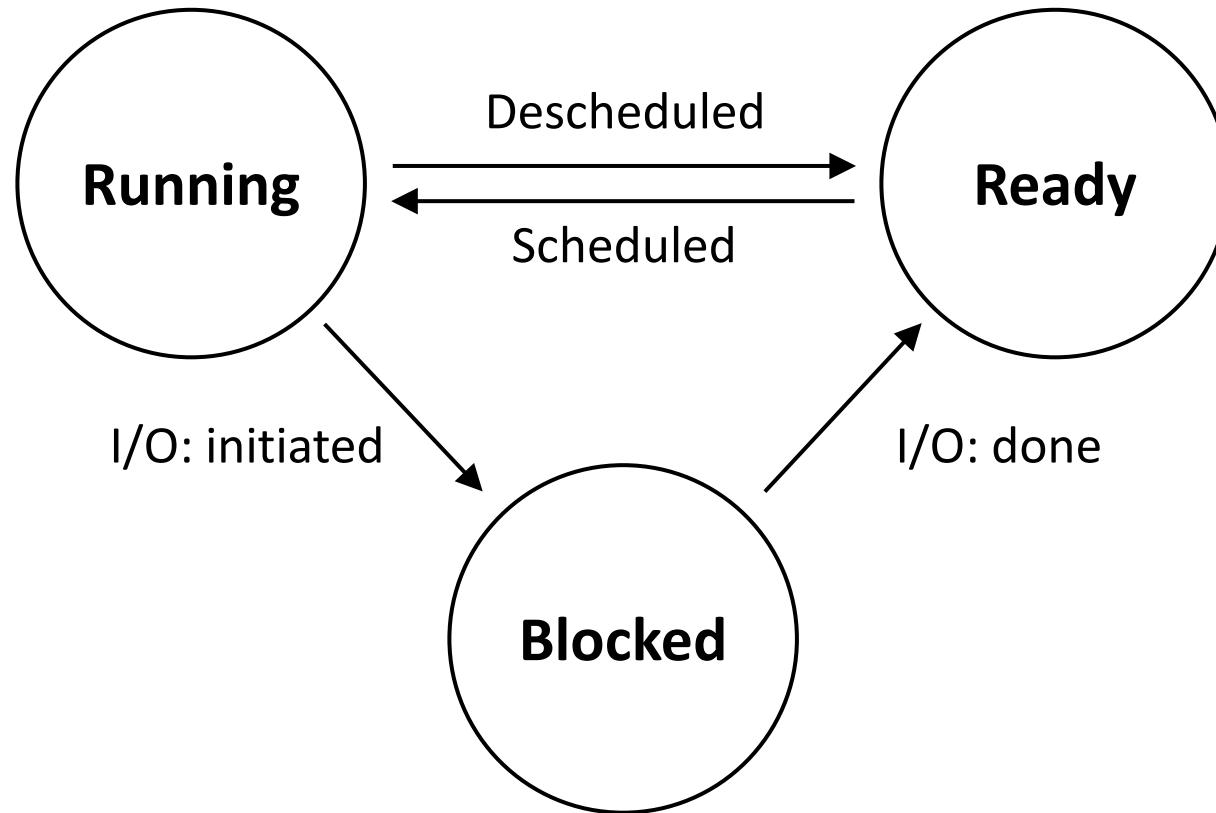
Loading: From Program To Process



Process States

- A process can be one of three states.
- **Running**
 - A process is running on a processor.
- **Ready**
 - A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- **Blocked**
 - A process has performed some kind of operation.
 - When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process: State Transitions



Tracing Process State: CPU Only

Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process ₀ is now done
5	—	Running	
6	—	Running	
7	—	Running	
8	—	Running	Process ₁ is now done

Tracing Process State: CPU and I/O

Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process ₀ initiates I/O
4	Blocked	Running	Process ₀ is blocked, so Process ₁ runs
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process ₁ is now done
9	Running	—	
10	Running	—	Process ₀ is now done

Data Structures

- OS maintains a data structure (e.g., **process list**) of all ready, running and blocked processes.
- **Process Control Block (PCB)**
 - A C-structure that contains information about each process.
 - **Register context**: a set of registers that define the state of a process

The xv6 Proc Structure

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

The xv6 Proc Structure

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

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

Summary

- We have introduced the most basic abstraction of the OS: the **process**.
- Low-level **mechanisms** are needed to switch between processes.
- High-level **policies** are required to schedule the processes in an intelligent way.

COMP 4736

Introduction to Operating Systems

02. System Calls

(OSTEP Ch. 5 & 6)

Ch. 05. Process API

(OSTEP Ch. 5)

Process API

- `fork()`
- `exec()`
- `wait()`
- Separation of `fork()` and `exec()`
 - IO redirection
 - pipe

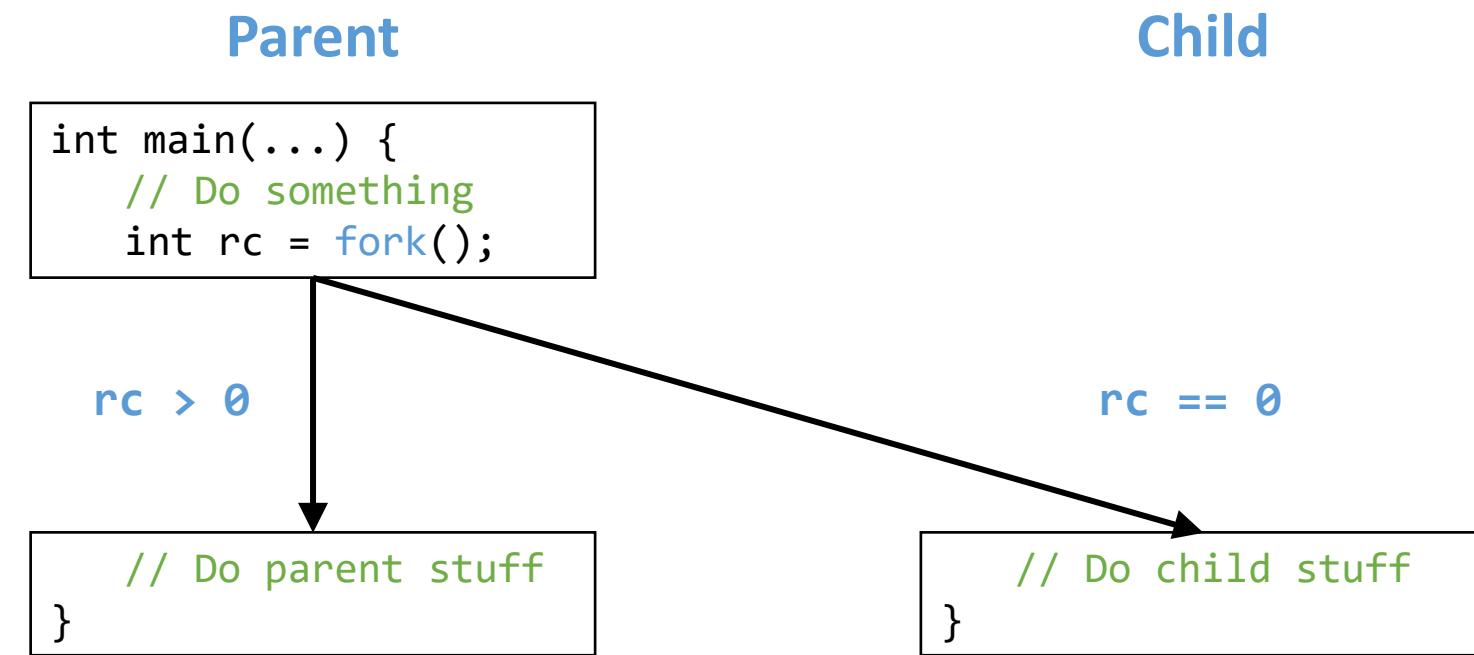
fork()

- Create a **child** process (in a strange way).
- The created process is an (almost) **exact copy** of the calling process.
- The **child** process does not start running at `main()`. It comes into life as it had **called** `fork()` itself.
- The child has its **own copy** of the address space, register, and PC.
- The newly created process becomes **independent** after it is created.
- `fork()` returns:
 - **PID** of child process for **parent**
 - **0** for **child**

Calling fork() (p1.c)

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

fork()



Calling fork() (Result)

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

or

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

- Not deterministic

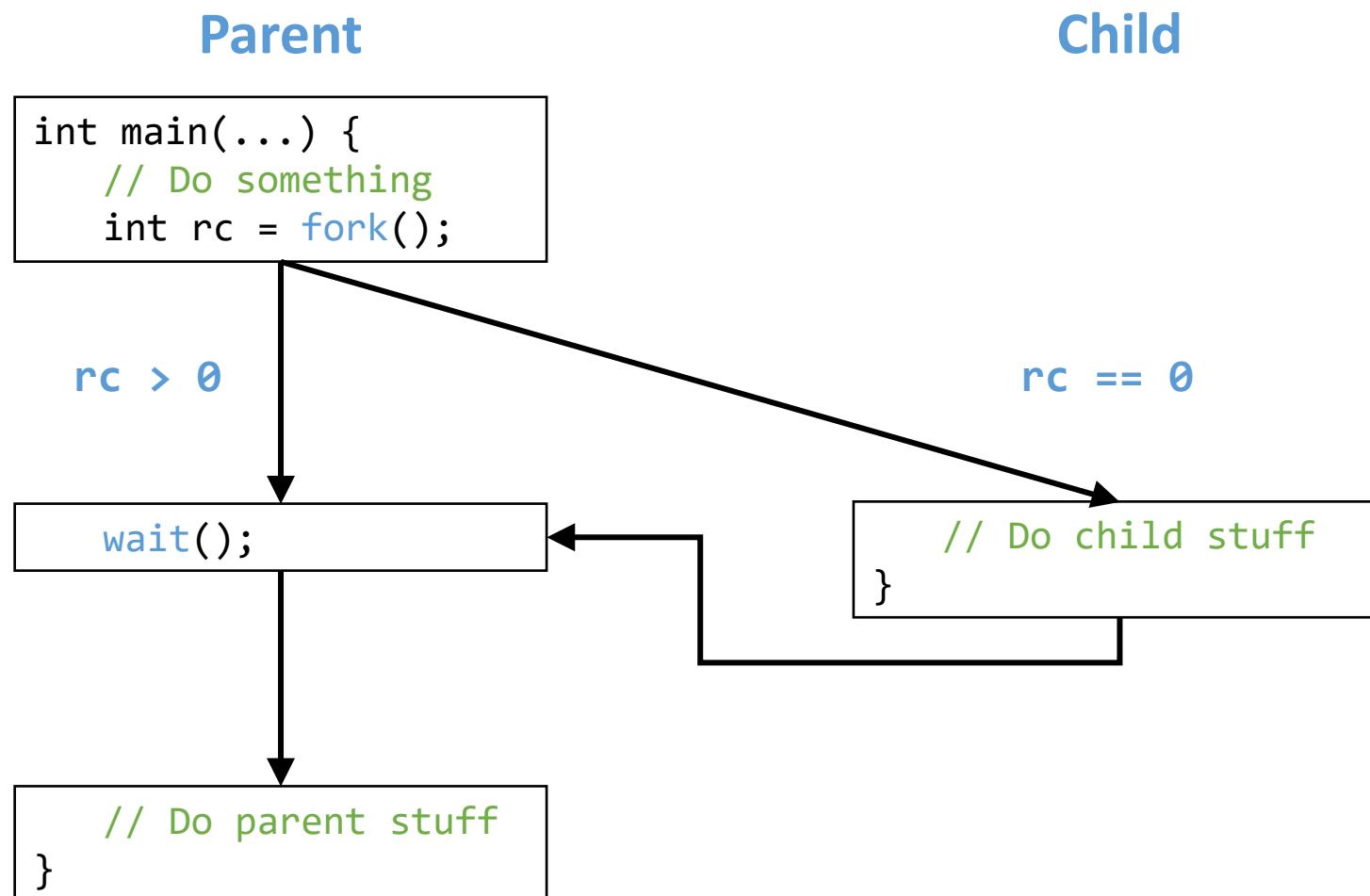
`wait()`

- When the child process is created, `wait()` in the parent process won't return until the (first) child has run and exited.
- The parent and the child **does not** have any **dependency**.
- In some cases, the application wants to enforce the order in which they are executed, e.g., the parent exits only after the child finishes.

Calling fork() And wait() (p2.c)

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

wait()



Calling fork() And wait() (Result)

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

- **Deterministic** output
- Either
 - Child runs first, prints before parent, or
 - Parent runs first, waits for child to finish, then prints.

exec()

- The caller wants to run a program that is **different** from the caller itself. E.g.,
 - Launch an editor
 - % ls -l
- OS needs to load a new binary image, initialize a new stack, initialize a new heap for the new program.
- Requires two parameters
 - The **name** of the **binary** file
 - The **array of arguments**

Calling fork(), wait(), And exec() (p3.c)

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

Calling fork(), wait(), And exec() (p3.c)

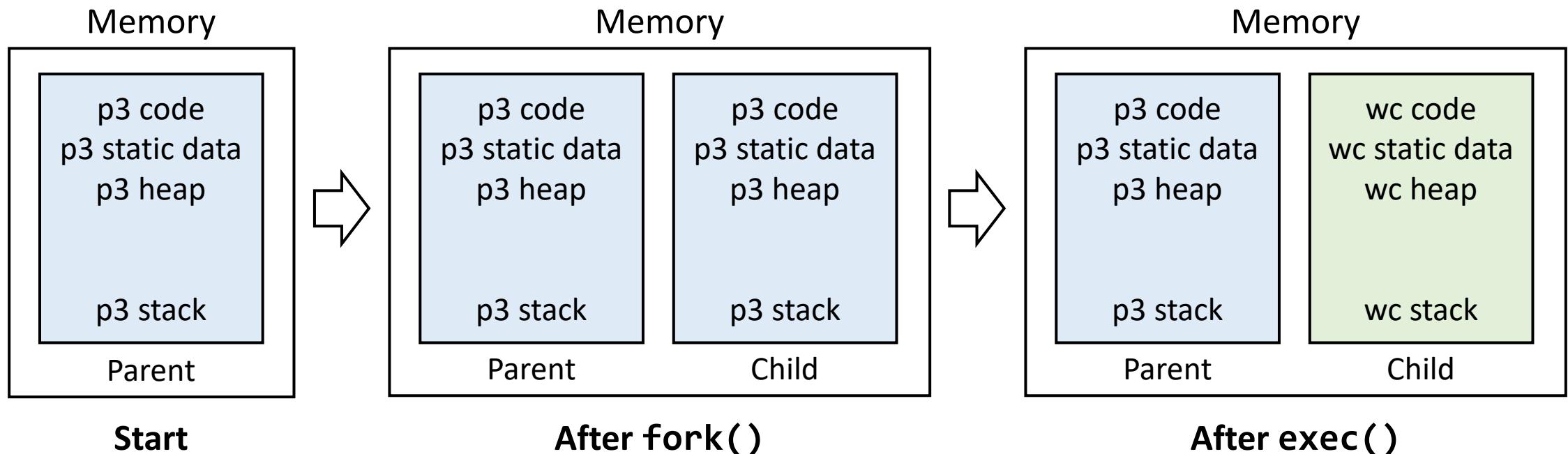
```
21 } else {                                // parent goes down this path (main)
22     int rc_wait = wait(NULL);
23     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
24         rc, rc_wait, (int) getpid());
25 }
26 return 0;
27 }
```

- Output

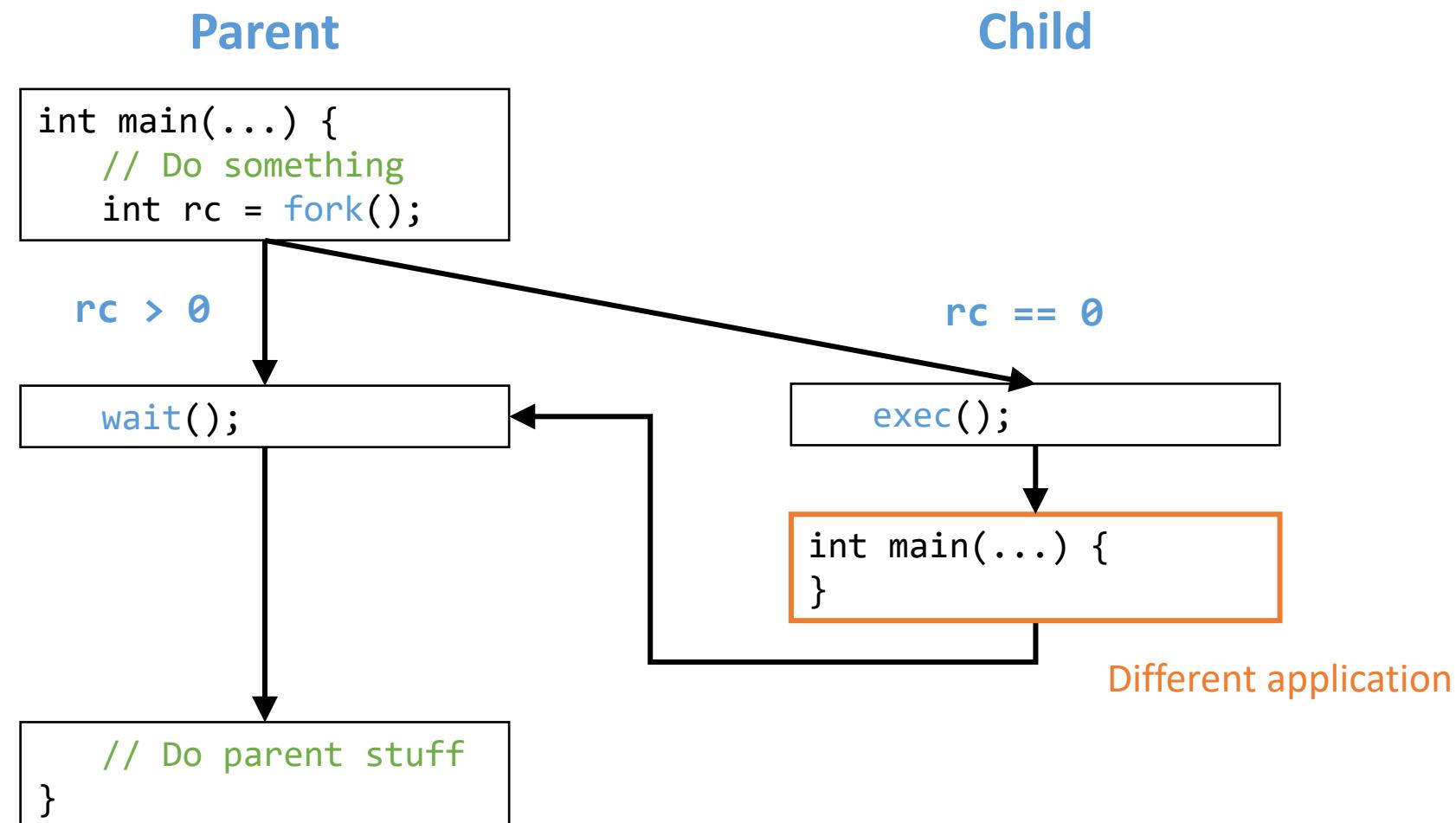
```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
    29      107      1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

When exec() is called

- It **loads** code (and data), and **overwrites** its current code segment (and data). It **does not** create a new process!
- exec() **does not return**!



exec()



Case Study: How does a shell work?

- In a basic OS, after initialization of hardware, the `init` process spawns a shell such as `bash`.
- Shell **reads** user command, **forks** a child, **execs** the command executable, **waits** for it to finish, and reads next command.
- Common commands like `ls` are all executables that are simply exec'ed by the shell.

IO Redirection (>)

```
prompt> wc p3.c > newfile.txt
```

- Shell calls `fork()` and `exec()` to accomplish redirection.
- After `fork()`, shell closes standard output, and opens the file `newfile.txt`.
- Any output will be sent to the file instead of the screen.

All of the Above With Redirection (p4.c)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <fcntl.h>
6 #include <sys/wait.h>
7
8 int main(int argc, char *argv[]) {
9     int rc = fork();
10    if (rc < 0) {
11        // fork failed
12        fprintf(stderr, "fork failed\n");
13        exit(1);
14    } else if (rc == 0) {
15        // child: redirect standard output to a file
16        close(STDOUT_FILENO);
17        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18    }
```

All of the Above With Redirection (p4.c)

```
19     // now exec "wc"...
20     char *myargs[3];
21     myargs[0] = strdup("wc");           // program: wc (word count)
22     myargs[1] = strdup("p4.c");         // arg: file to count
23     myargs[2] = NULL;                  // mark end of array
24     execvp(myargs[0], myargs);         // runs word count
25 } else {
26     // parent goes down this path (main)
27     int rc_wait = wait(NULL);
28 }
29 return 0;
30 }
```

All of the Above With Redirection (Output)

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

- **Nothing** seems to be happening when p4 is run.
 - Actually, p4 did call fork(), then wc is run via execvp(), and the output is redirected to p4.output.
 - When the output file is cat'ed, all the expected output from running wc is found.

pipe()

- Using `pipe()`, the output of one process is connected to an in-kernel pipe (i.e., queue), and the input of another process is connected to the same pipe.
- E.g., to count the occurrence of a word in a file using `grep` and `wc`:

```
grep -o foo file | wc -l
```

- Try

```
echo hello world | wc
```

Difference Between IO Redirection and Pipes

- Pipes automatically **clean** themselves up. When using IO redirection, the user has to explicitly **delete** the temporary file.
- Pipes can pass arbitrarily **long data** while file redirection requires sufficient available **disk space**.
- In pipe, reader and writer can proceed in **parallel**, while one has to finish before the others to start in redirection.

Summary

- We have introduced some **APIs** for process creation in Unix.
 - `fork()`
 - `wait()`
 - `exec()`
- We have also examined **IO redirection** and **pipes**.

References

- Linux man pages online.
<https://man7.org/linux/man-pages/index.html>

Section 2: System calls

Section 3: C standard library functions

Ch. 06. Limited Direction Execution

(OSTEP Ch. 6)

How to Efficiently Virtualize the CPU with Control?

- The OS needs to share the physical CPU by **time sharing**.
- Issues:
 - **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
1. Create entry for process list	
2. Allocate memory for program	
3. Load program into memory	
4. Set up stack with argc / argv	
5. Clear registers	
6. Execute <code>call main()</code>	
	7. Run <code>main()</code>
	8. Execute <code>return</code> from <code>main()</code>
9. Free memory of process	
10. Remove from process list	

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

Recap: Process Creation

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

Recap: Process Creation

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.

Problems

- Issue #1: User process can perform illegal operation.
- What if?

```
int *i;  
i = 0;  
*i = 1;
```

- Issue #2: Regaining the control of CPU is not easy.
- What if?

```
i = -1;  
while (i < 0) {  
    // do something  
}
```

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:** Use 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

Example: write() on Pintos

- User program

```
int main() {
    write(...);
}
```

- pintos/src/lib/user/syscall.c

```
int write (int fd, const void *buffer, unsigned size)
{
    return syscall3 (SYS_WRITE, fd, buffer, size);
}
```

- pintos/src/lib/syscall-nr.h

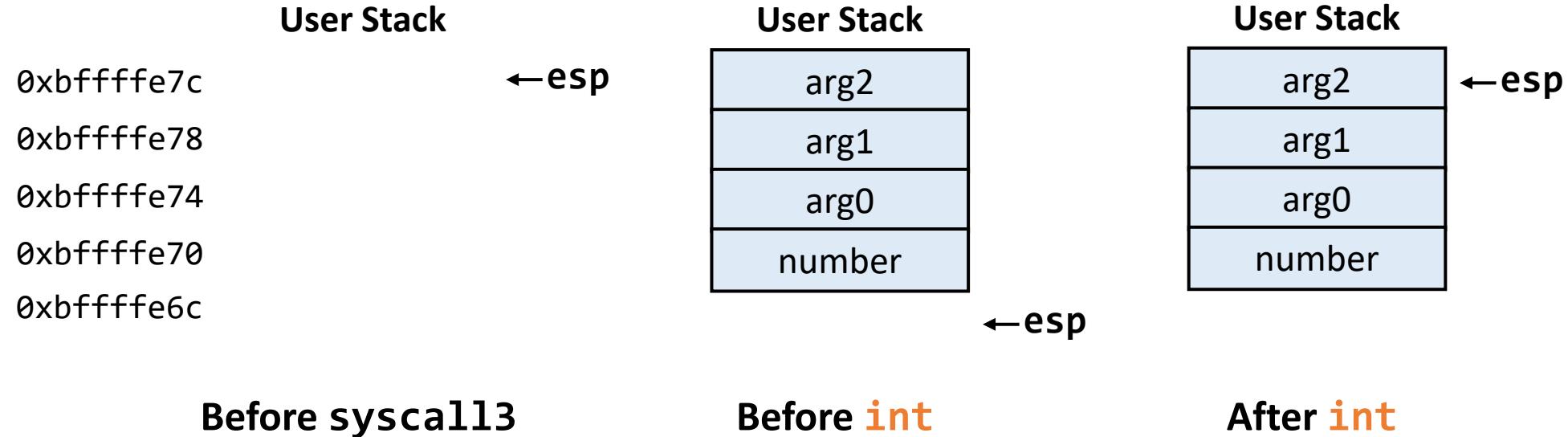
```
/* System call numbers. */
enum {
    ...
    SYS_WRITE,           /* Write to a file. */
    ...
}
```

Example: write() on Pintos

- `pintos/src/lib/user/syscall.c`

```
/* Invokes syscall NUMBER, passing arguments ARG0, ARG1, and
   ARG2, and returns the return value as an `int'. */
#define syscall3(NUMBER, ARG0, ARG1, ARG2)
    ({
        int retval;
        asm volatile
            ("pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; "
             "pushl %[number]; int $0x30; addl $16, %%esp"
             : "=a" (retval)
             : [number] "i" (NUMBER),
               [arg0] "r" (ARG0),
               [arg1] "r" (ARG1),
               [arg2] "r" (ARG2)
             : "memory");
        retval;
    })
```

Example: write() on Pintos



Definition of System Calls (Pintos)

- `pintos/src/lib/user/syscall.c`

```
int open (const char *file) {
    return syscall1 (SYS_OPEN, file);
}

int filesize (int fd) {
    return syscall1 (SYS_FILESIZE, fd);
}

int read (int fd, void *buffer, unsigned size) {
    return syscall3 (SYS_READ, fd, buffer, size);
}

int write (int fd, const void *buffer, unsigned size) {
    return syscall3 (SYS_WRITE, fd, buffer, size);
}

void seek (int fd, unsigned position) {
    syscall2 (SYS_SEEK, fd, position);
}
```

Definition of System Calls (Pintos)

- `pintos/src/lib/user/syscall.c`

```
#define syscall0(NUMBER) \
    ... \
    ("pushl %[number]; int $0x30; addl $4, %%esp" \
     ... \
     \
#define syscall1(NUMBER, ARG0) \
    ... \
    ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
     ... \
     \
#define syscall2(NUMBER, ARG0, ARG1) \
    ... \
    ("pushl %[arg1]; pushl %[arg0]; " \
     "pushl %[number]; int $0x30; addl $12, %%esp" \
     ... \
     \
#define syscall3(NUMBER, ARG0, ARG1, ARG2) \
    ... \
    ("pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; " \
     "pushl %[number]; int $0x30; addl $16, %%esp" \
     ... \
     \

```

System Call Numbers (Pintos)

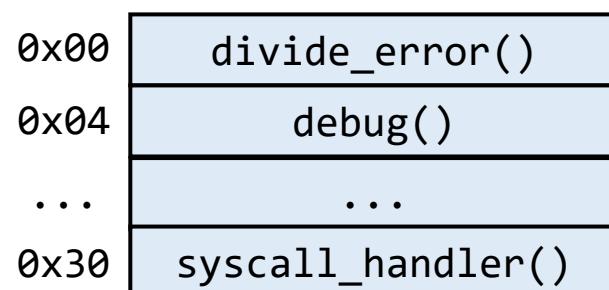
- `pintos/src/lib/syscall-nr.h`

```
/* System call numbers. */
enum
{
    /* Projects 2 and later. */
    SYS_HALT,                      /* Halt the operating system. */
    SYS_EXIT,                       /* Terminate this process. */
    SYS_FORK,                       /* Fork a new process */
    SYS_EXEC,                        /* Execute a program by filename */
    SYS_DUP2,                        /* Duplicate a file descriptor */
    SYS_PIPE,                        /* Create a pipe */
    SYS_WAIT,                        /* Wait for a child process to die. */
    SYS_CREATE,                      /* Create a file. */
    SYS_REMOVE,                      /* Delete a file. */
    SYS_OPEN,                         /* Open a file. */
    SYS_FILESIZE,                    /* Obtain a file's size. */
    SYS_READ,                        /* Read from a file. */
    SYS_WRITE,                        /* Write to a file. */
    SYS_SEEK,                         /* Change position in a file. */
    SYS_TELL,                         /* Report current position in a file. */
    SYS_CLOSE,                        /* Close a file. */
};
```

Registering System Call Handler in IDT

- Kernel sets **up interrupt descriptor table** (IDT) or **trap table** at **boot time**.
- Table of the locations of the interrupt handlers for the associated interrupt number.
- Register the syscall handler at interrupt **0x30** for Pintos.

```
void syscall_init (void)
{
    intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
    lock_init(&filesys_lock);
}
```



Interrupt Descriptor Table (IDT) / Trap Table

Executing a System Call

- Trap (interrupt) instruction
 - Jumps into the kernel and raises the privilege level to kernel mode.
 - Saves registers at the kernel stack.
 - Jumps to trap handler according to the trap table (e.g., IDT).
- Return-from-trap instruction
 - Restores registers from the kernel stack.
 - Returns into the calling user program.
 - Reduces the privilege level back to user mode.

Trap Table

- Sources of trap
 - Hard disk operation completed
 - Keyboard interrupt
 - System call
- Trap handler
 - Codes to run when certain trap (interrupt) takes place.
- Trap table
 - The address of the trap handlers.
 - Privileged (kernel) mode is needed for setting up the trap table.

Limited Direct Execution Protocol

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of ... syscall handler	
OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC return-from-trap		
	restore regs (from kernel stack) move to user mode jump to main	
		Run main() ... Call system call trap into OS

Limited Direct Execution Protocol

(Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	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 <code>exit()</code>)
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**.
 - Opens and reads a file.
 - Sends a message to another machine
 - **yield**
- Applications also transfer control to the OS when they do something **illegal**. **Trap** will be generated.
 - Divides by zero
 - Tries to access memory that it shouldn't be able to access

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

A Non-Cooperative Approach: The OS Takes Control

- A **timer interrupt**
 - During the boot sequence, the OS starts the timer.
 - The timer raises an interrupt every few milliseconds.
- When the interrupt is raised:
 - The currently running process is suspended.
 - Save enough of the state of the process.
 - A pre-configured **interrupt handler** in the OS runs.
 - OS can even stop the current process, and start a different one.

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

Saving and Restoring Context

- Scheduler makes decision when OS regains control.
 - 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
 - Program counter (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 Direct Execution Protocol (Timer Interrupt)

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of ... syscall handler timer handler	
start interrupt timer		
	start timer Interrupt CPU in X ms	
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A ...
	timer interrupt save regs(A) → k-stack(A) move to kernel mode jump to trap handler	

Limited Direct Execution Protocol (Timer Interrupt)

(Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
Handle the trap Call switch() routine save regs(A) → proc t(A) restore regs(B) ← proc t(B) switch to k-stack(B) return-from-trap (into B)		
	restore regs(B) ← 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,
2     struct context *new);
3
4 # Save current register context in old
5 # and then load register context from new.
6 .globl swtch
7 swtch:
8     # Save old registers
9     movl 4(%esp), %eax      # put old ptr into eax
10    popl 0(%eax)          # save the old IP
11    movl %esp, 4(%eax)      # and stack
12    movl %ebx, 8(%eax)      # and other registers
13    movl %ecx, 12(%eax)
14    movl %edx, 16(%eax)
15    movl %esi, 20(%eax)
16    movl %edi, 24(%eax)
17    movl %ebp, 28(%eax)
```

```
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 ctxtt
```

Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations by:
 - **Disabling interrupts** during interrupt processing
 - Using a number of sophisticated **locking** schemes to protect concurrent access to internal data structures.

Summary

- Some key low-level mechanisms for CPU virtualization have been described, namely **limited direction execution**.
 - First, make sure to set up the **hardware** to limit what processes can do without OS assistance.
 - Then, just run the desired program on the **CPU**.
- The OS “baby proofs” the CPU by
 - Setting up **trap handlers** and starting an **interrupt timer** at **boot time**.
 - Then only running processes in a **restricted mode**.
- Then OS intervention is only required when
 - **Privileged** operations are performed, or
 - Processes have **monopolized** the CPU for too long and need to be switched out.

References

- Pintos Projects: Table of Contents
<https://web.stanford.edu/class/cs140/projects/pintos/pintos.html>
- Pintos source codes
<https://github.com/EddieCarlson/pintos>

COMP 4736

Introduction to Operating Systems

03. Operating System Structures and Clocks

(OSTEP Ch. 7 & 8)

Ch. 07. Scheduling: Introduction

(OSTEP Ch. 7)

Mechanisms and Policies

- We have discussed the low-level **mechanisms** of running processes (e.g., context switching).
- Now we will introduce the high-level **policies** that OS scheduler employs.

Scheduling: Introduction

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

Scheduling Metrics

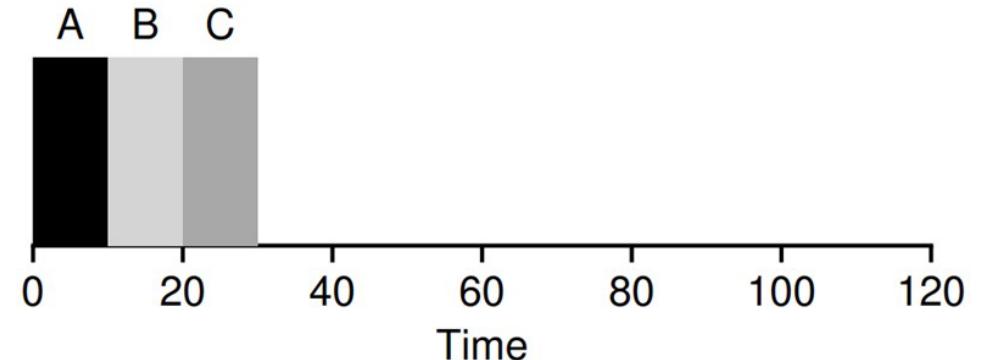
- **Performance** metric: **Turnaround time**
- The time at which the job **completes** minus the time at which the job **arrived** in the system.

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

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

First In, First Out (FIFO)

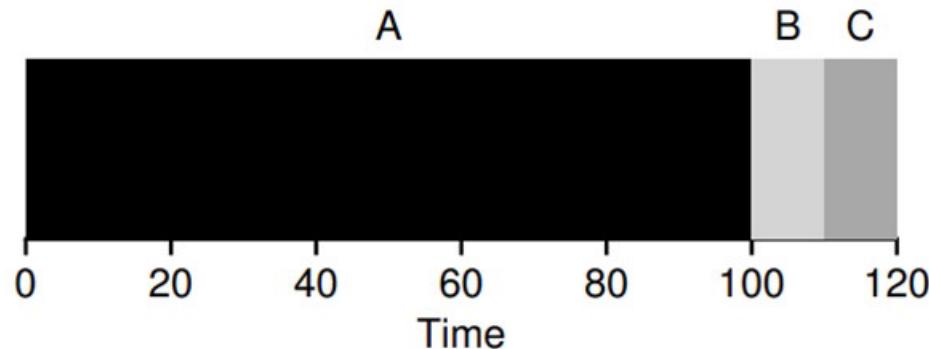
- **First In, First Out (FIFO)**
 - Sometimes called **First Come, First Served (FCFS)**
 - Very simple and easy to implement.
- Example:
 - A arrived just before B which arrived just before C.
 - Each job runs for 10 seconds.



$$\text{Average Turnaround Time} = \frac{10 + 20 + 30}{3} = 20 \text{ seconds}$$

Why FIFO is not that great?

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

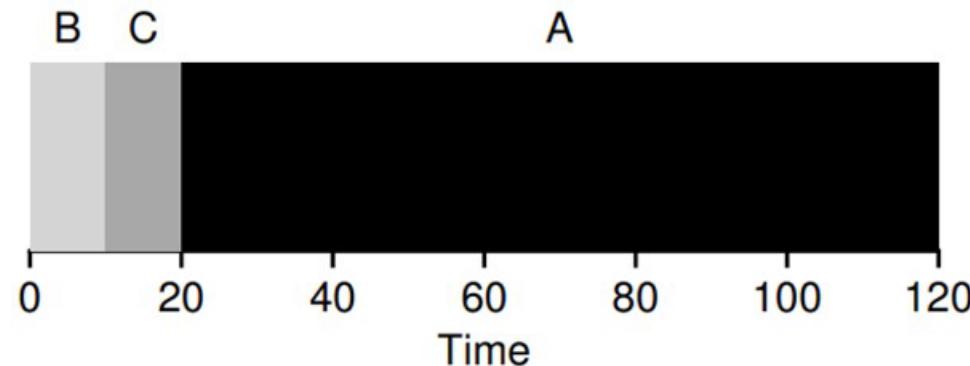


Convoy Effect

$$\text{Average Turnaround Time} = \frac{100 + 110 + 120}{3} = 110 \text{ seconds}$$

Shortest Job First (SJF)

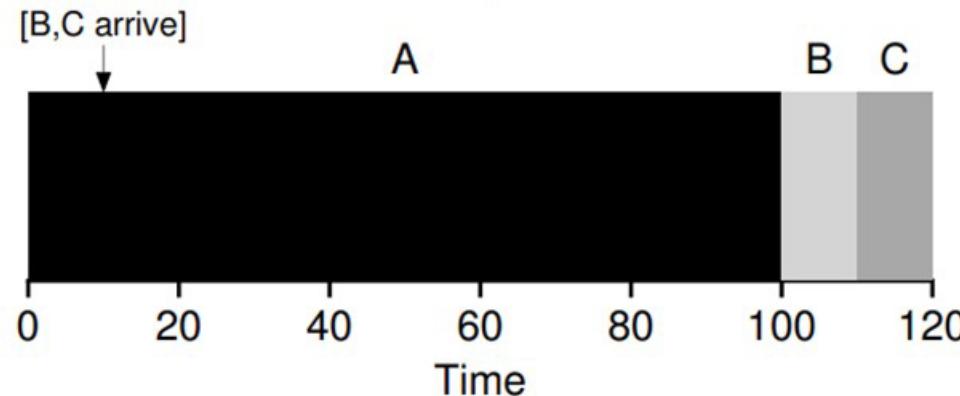
- Run the shortest job first, then the next shortest, and so on.
 - Non-preemptive scheduler
- Example:
 - A arrived just before B which arrived just before C.
 - A runs for 100 seconds, B and C run for 10 each.



$$\text{Average Turnaround Time} = \frac{10 + 20 + 120}{3} = 50 \text{ seconds}$$

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.



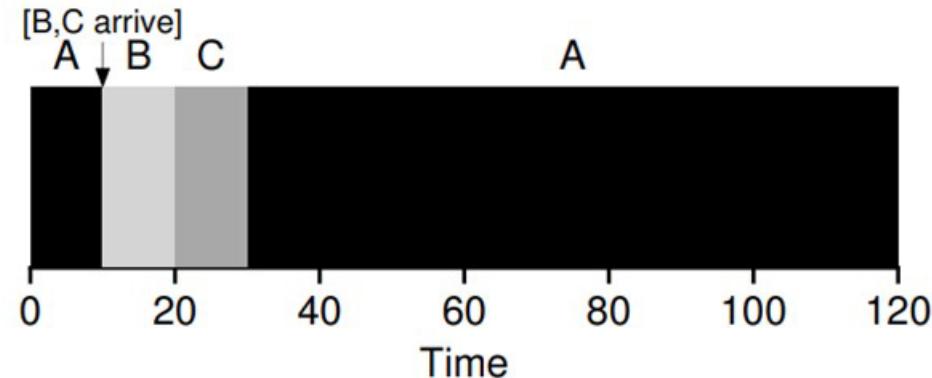
$$\text{Average Turnaround Time} = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33 \text{ seconds}$$

Shortest Time-to-Completion First (STCF)

- Let's relax assumption 3: Jobs no longer run to completion in one go.
- Add **preemption** to SJF.
 - Also known as **Preemptive Shortest Job First (PSJF)**.
- A new job enters the system:
 - STCF scheduler examines 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.



$$\text{Average Turnaround Time} = \frac{(120 - 0) + (20 - 10) + (30 - 10)}{3} = 50 \text{ seconds}$$

A New Metric: Response Time

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

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

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

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

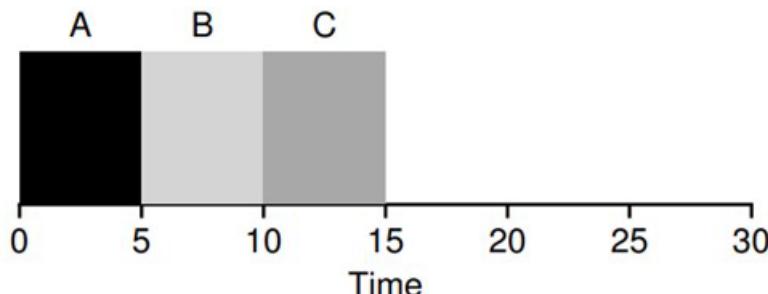
Round-Robin (RR) Scheduling

- 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.

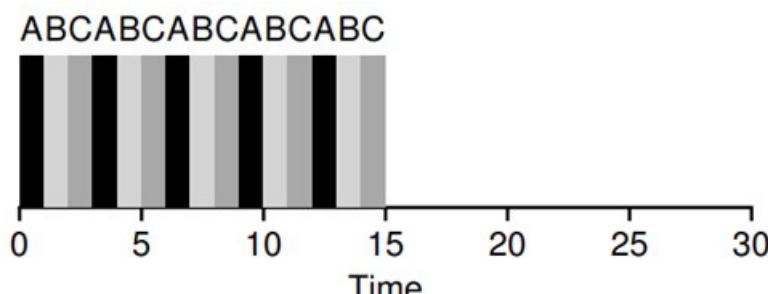
RR is fair, but performs **poorly** on metrics such as **turnaround time**.

RR Scheduling Example

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



SFJ (Bad for Response Time)



RR with 1-s Time Slice (Good for Response Time)

$$\text{Average Response Time} = \frac{0 + 5 + 10}{3} = 5 \text{ seconds}$$

$$\text{Average Response Time} = \frac{0 + 1 + 2}{3} = 1 \text{ second}$$

Time-Slice Length

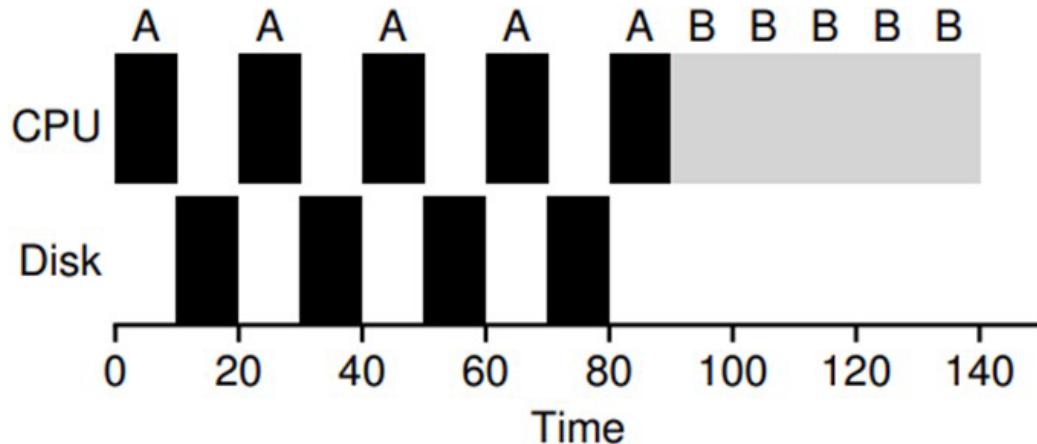
- The length of the time slice is **critical**.
- The **shorter** time slice...
 - Better response time
 - The **cost of context switching** will dominate overall performance.
- The **longer** time slice...
 - **Amortize** the cost of switching.
 - Worse response time

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

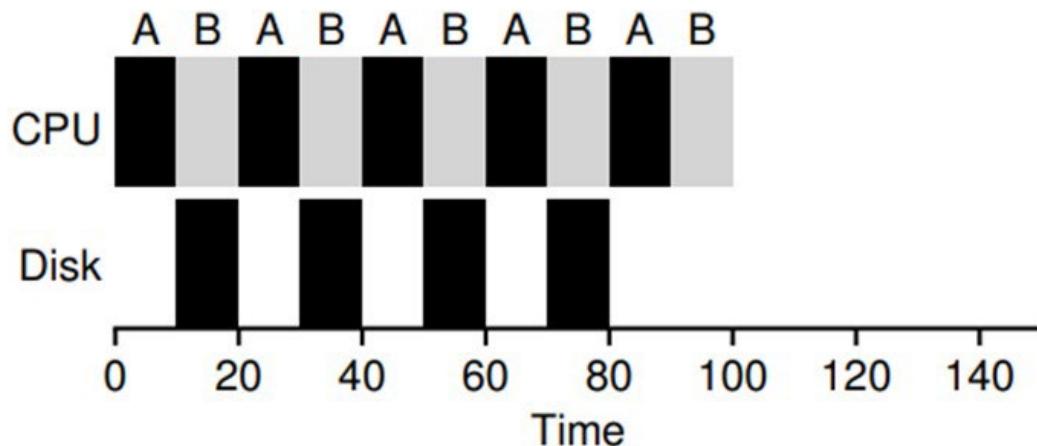
Incorporating I/O

- Let's relax assumption 4: All programs perform I/O
- Example:
 - A and B need 50 ms of CPU time each.
 - A runs for 10 ms and then issues an I/O request
 - I/Os each take 10 ms
 - B simply uses the CPU for 50 ms and performs no I/O
 - The scheduler runs A first, then B after

Incorporating I/O



Poor Use of Resources



Overlap Allows Better Use of Resources

Maximize the
CPU utilization

Incorporating I/O

- 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.

Summary

- The basic ideas behind **scheduling** have been introduced.
- **Two families** of approaches have been developed.
 - The first runs the **shortest job remaining** and optimizes **turnaround time**.
 - The second **alternates** between all jobs and optimizes **response time**.
- No approach is always the best. **Trade-off** exists inherently.
- I/O can also be incorporated into the scheduling.
- However, the OS still cannot see into the future.

Ch. 08. Scheduling: The Multi-Level Feedback Queue

(OSTEP Ch. 8)

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.

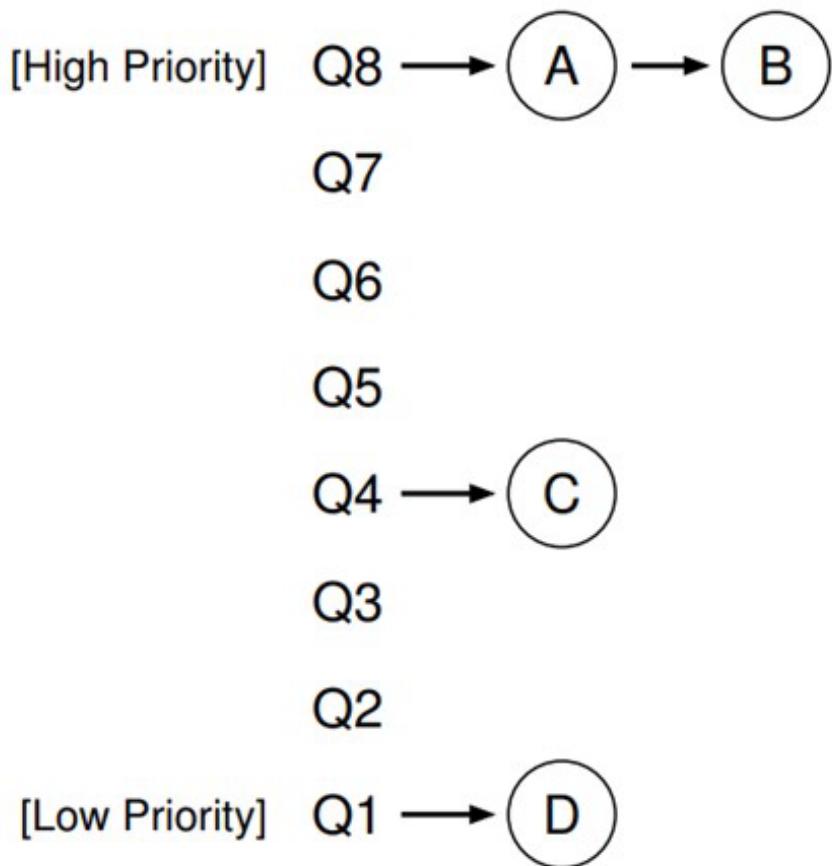
Rule 1: If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).

Rule 2: If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR.

MLFQ: Basic Rules

- 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



Jobs C and D would never get to run!

But it is only a static snapshot.

MLFQ: How to Change Priority

- MLFQ priority adjustment algorithm:

Rule 3: When a job enters the system, it is placed at the **highest** priority

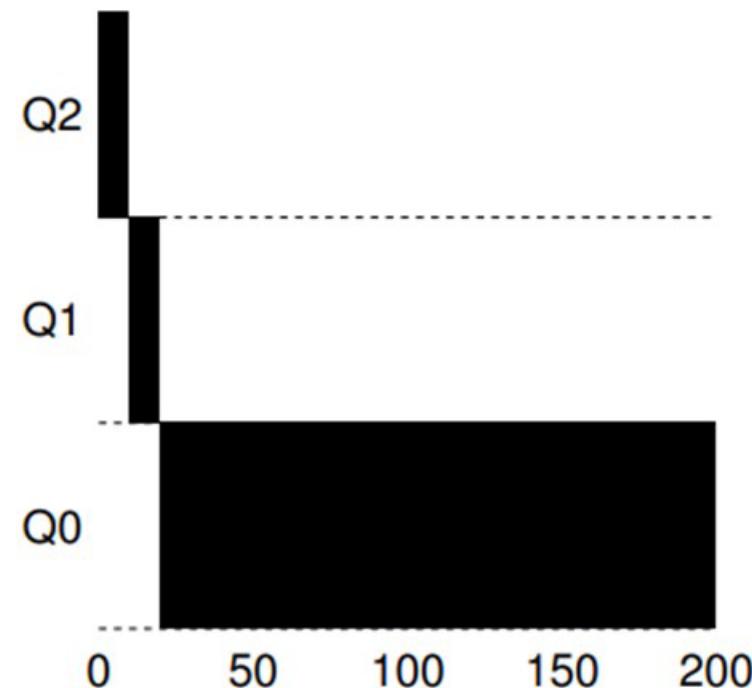
Rule 4a: If a job uses up an **entire** time slice while running, its priority is **reduced** (i.e., it moves down on queue).

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

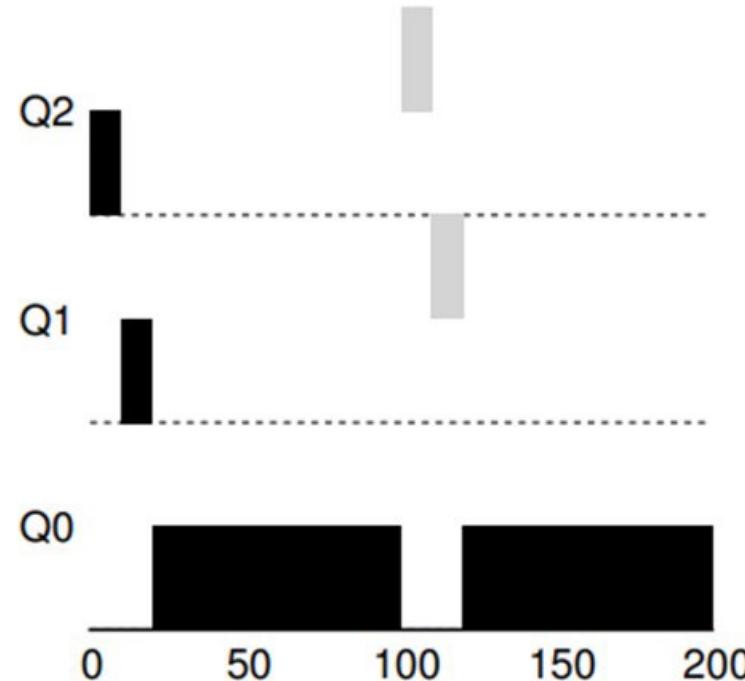
Example 1: A Single Long-Running Job

- A three-queue scheduler with time slices of 10 ms.
- A long-running job over time:



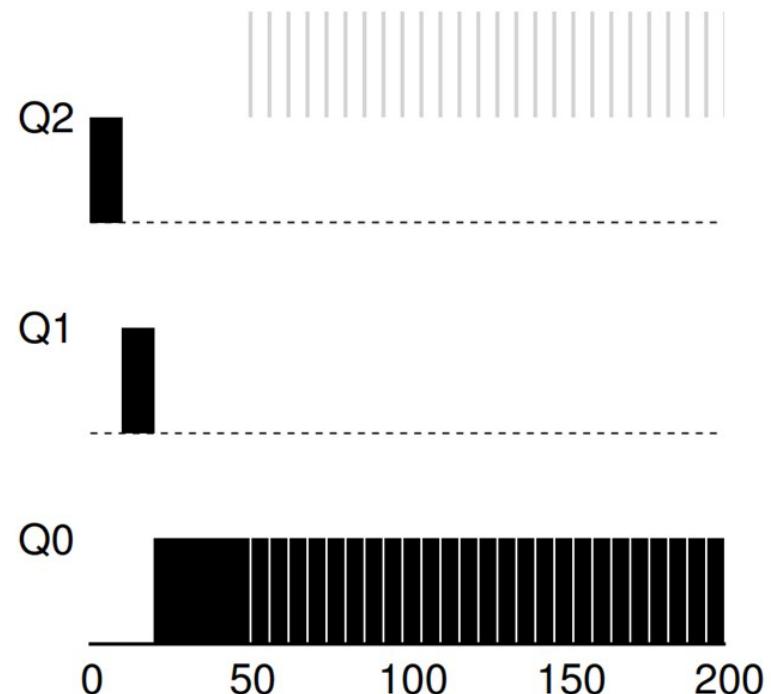
Example 2: Along Came A Short Job

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



Example 3: What About I/O?

- **Job A** (black): A long-running CPU-intensive job
- **Job B** (gray): An interactive job that needs the CPU only for 1 ms before performing an I/O operation.



MLFQ keeps an interactive job at the highest priority.

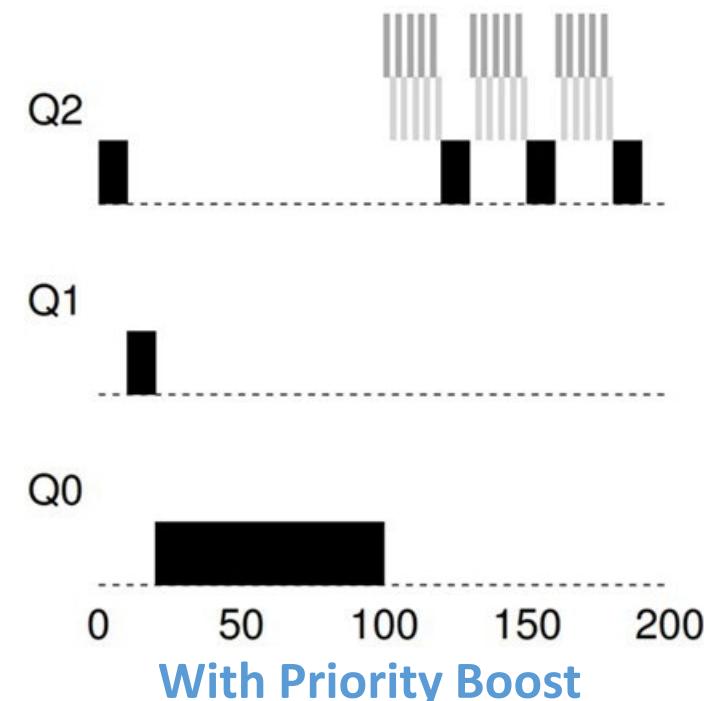
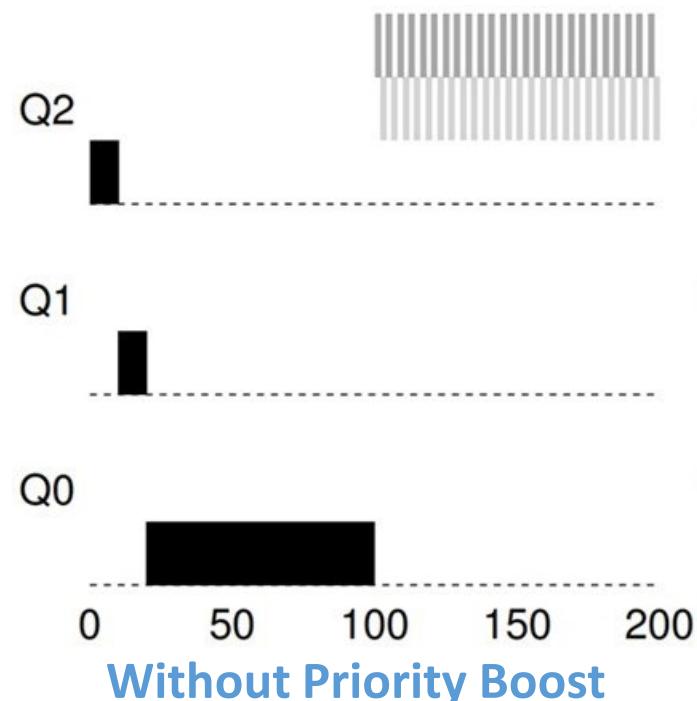
Problems with the Basic MLFQ

- **Starvation**
 - If there are “too many” interactive jobs in the system, long-running jobs will **never** receive any CPU time.
- **Gaming the scheduler**
 - After running 99% of a time slice, the job issues an I/O operation.
 - The job gains a higher percentage of CPU time.
- A program may **change its behavior** over time.
 - CPU-bound process may transition to a phase of interactivity.

The Priority Boost

Rule 5: After some time period S , move all the jobs in the system to the topmost queue.

- A long-running job (A) with two short-running interactive jobs (B, C).



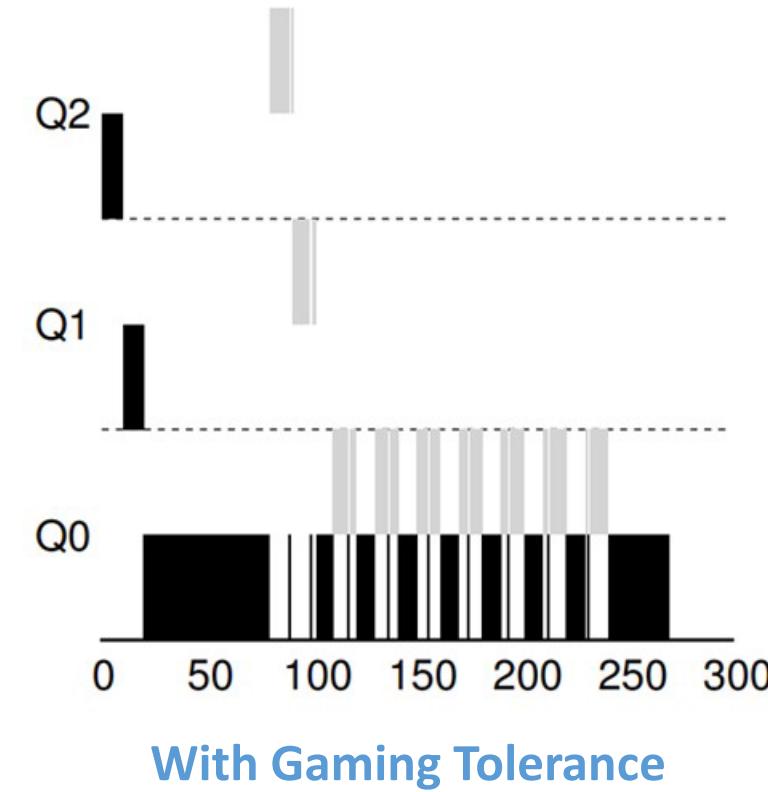
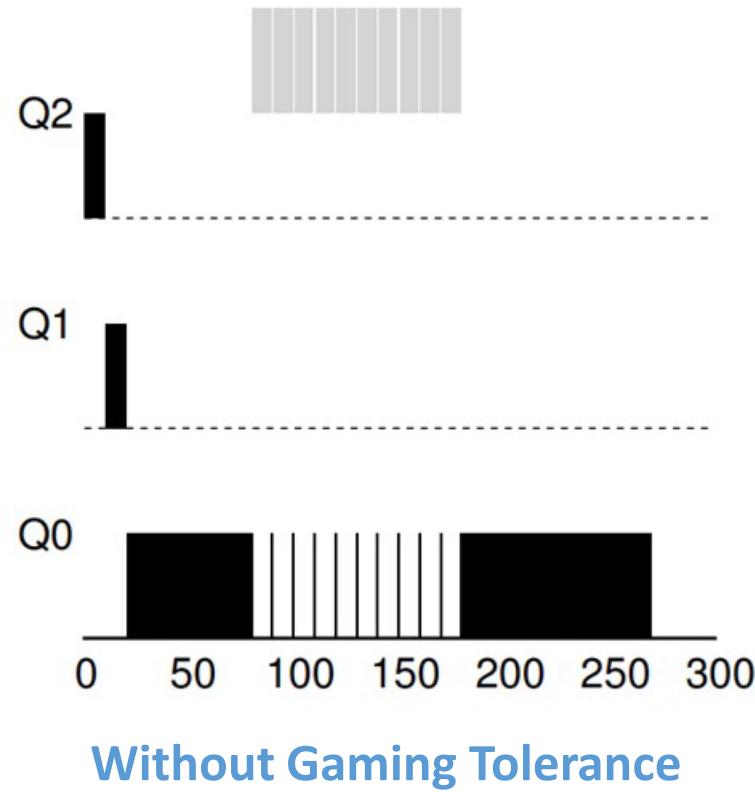
Better Accounting

- How to prevent gaming of our scheduler?
- Solution:
 - Rewrite Rules 4a and 4b to the following rule:

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 one queue).

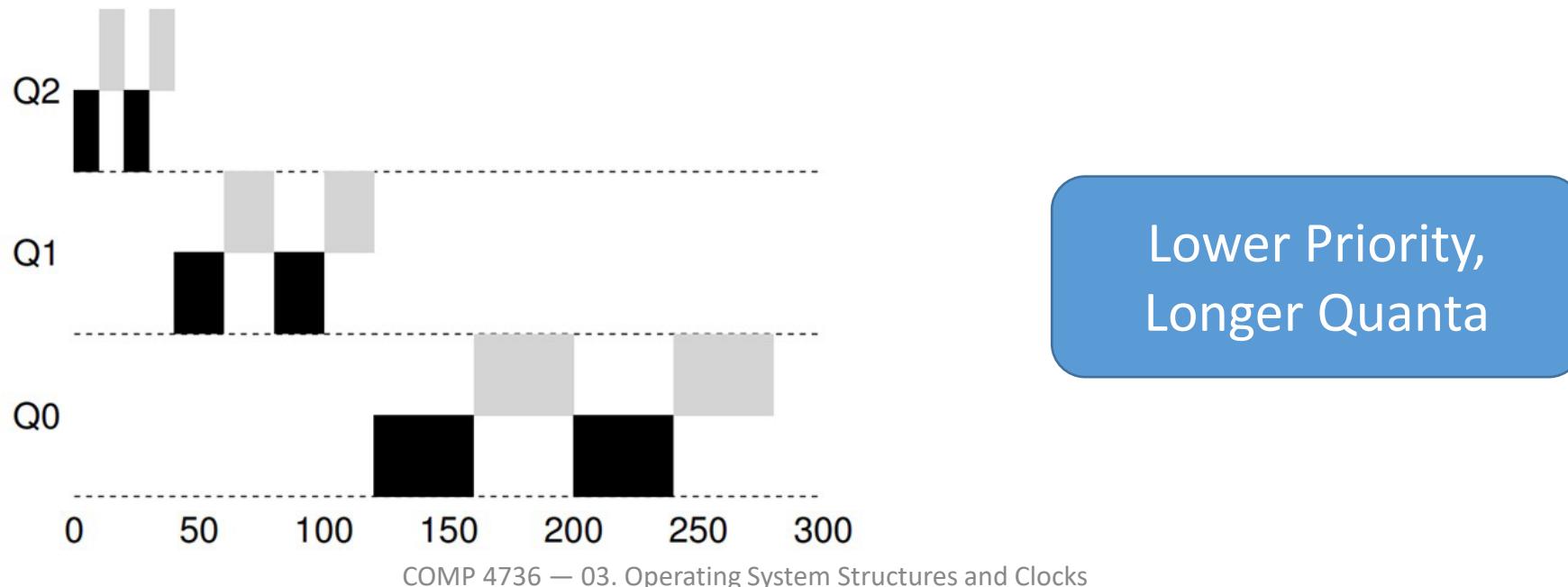
Better Accounting

- Example:



Tuning MLFQ and Other Issues

- High-priority queues → Interactive jobs → Quickly alternating between them
(e.g., 10 or fewer milliseconds)
- Low-priority queue → CPU-bound jobs → Longer time slices
(e.g., 100s of milliseconds)



The Solaris MLFQ implementation

- For the Time-Sharing scheduling class (TS):
- 60 Queues
- Slowly increasing time-slice lengths
 - Highest priority: 20 milliseconds
 - Lowest priority: A few hundred milliseconds
- Priorities are boosted around every 1 second or so.

FreeBSD Scheduler (4.3)

- MLFQ without queue.
- Instead, it uses mathematical formula.
- Compute the priority of a process based upon...
 - How much CPU the process has used.
 - Boost priority by usage decay.
 - Take the **advice** from the user (nice command).
- For efficiency, use queue.

MLFQ: Summary

- The refined set of MLFQ rules:

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 round-robin fashion using the time slice (quantum length) of the given queue.

Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).

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 one queue).

Rule 5: After some time period S , move all the jobs in the system to the topmost queue.

- It does not require prior knowledge on the CPU usage of a process.

COMP 4736

Introduction to Operating Systems

04. Process Management I

(OSTEP Ch. 9 & 10)

Ch. 09. Scheduling: Proportional Share

(OSTEP Ch. 9)

Proportional-Share Scheduler

- Fair-share scheduler
 - It tries to guarantee that each job obtain a certain percentage of CPU time.
 - Instead of optimizing for turnaround or response time

Basic Concept

- **Tickets** represent the share.
- 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 (out of 100 tickets) → receive 75% of the CPU
 - Process B has 25 tickets (out of 100 tickets) → receive 25% of the CPU

Lottery Scheduling

- The scheduler picks a **winning ticket**.
 - Loads the state of that winning process and runs it.
- Example (100 tickets in total):
 - 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	62	43	0	49	12
Resulting schedule																			
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	16/20 = 80%	
B		B				B			B		B							4/20 = 20%	

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:
 - Users **A** and **B** are given 100 tickets each (global currency).
 - A is running two jobs, **A1** and **A2**. Each is given 500 tickets (A's currency).
 - B is running one job, **B1**, and it is given 10 tickets (B's currency).

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

- **Ticket transfer**
 - A process can **temporarily** hand off its tickets to another process.
 - E.g., Client requests Server to perform some work
 - ➔ Client can transfer tickets to Server
 - ➔ Server has more CPU time to finish the task
 - ➔ Server transfer the tickets back to Client at the end
- **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**.



```
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, >= 0 and <= (totaltickets - 1)
6 int winner = getrandom(0, totaltickets);
7
8 // current: use this to walk through the list of jobs
9 node_t *current = head;
10 while (current) {
11     counter = counter + current->tickets;
12     if (counter > winner)
13         break; // found the winner
14     current = current->next;
15 }
16 // 'current' is the winner: schedule it...
```

Implementation

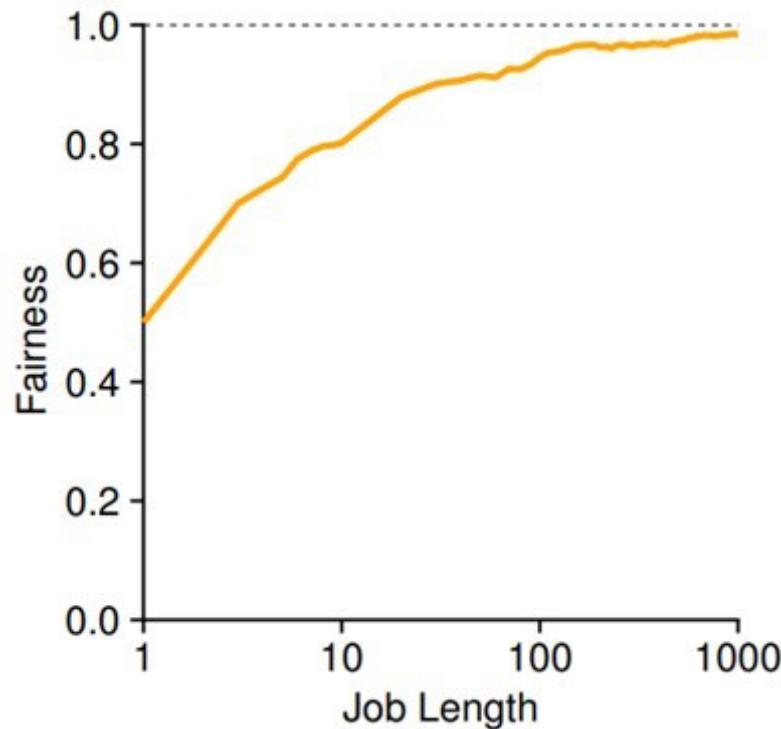
- F : Fairness metric
 - The time the first job completes divided by the time that the second job completes.
- Example:
 - There are two jobs, each job has runtime $R = 10$.
 - Both have the same number of tickets.
 - First job finishes at time 10. Second job finishes at time 20.

$$F = \frac{10}{20} = 0.5$$

- F will be close to 1 when both jobs finish at nearly the same time.

Lottery Fairness Study

- Average fairness over thirty trials is plotted against the length of the two jobs ($R = 1$ to 1000).



Only as the jobs run for a significant number of time slices does the lottery scheduler approach the desired fair outcome.

Stride Scheduling

- Stride scheduling
 - Deterministic fair-share scheduler
- Define **stride** and **pass** for each process
 - **Stride** = (A large number) / (Number of tickets of the process)
 - **Pass** = Counter of the process, increased by its stride
- Pick the process with the **lowest pass** value to run.

Stride Scheduling Example

- Pseudocode implementation

```
curr = remove_min(queue);      // pick client with min pass
schedule(curr);                // run for quantum
curr->pass += curr->stride;   // update pass using stride
insert(queue, curr);          // return curr to queue
```

- Example:
 - Jobs **A**, **B** and **C** have 100, 50 and 250 tickets, respectively.
 - Using a large number of 10,000, **stride** values for A, B and C would be 100, 200 and 40, respectively.
 - Each time a process runs, its **pass** value will be incremented by its stride.

Stride Scheduling Example

Pass (A) (Stride = 100)	Pass (B) (Stride = 200)	Pass (C) (Stride = 40)	Which 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	...

Stride scheduling needs to maintain the pass value **per process**.
(No global state in lottery scheduling.)

If new job enters with pass value 0, it will monopolize the CPU!

The Linux Completely Fair Scheduling (CFS)

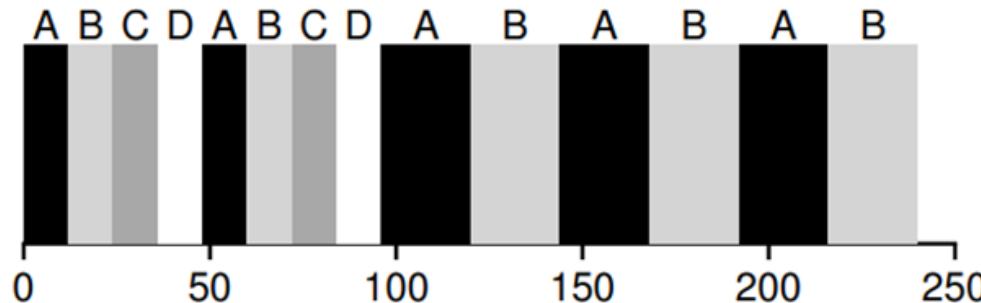
- Completely Fair Scheduling (CFS)
 - The current CPU scheduler in Linux
 - Achieves similar fair-share goals in an alternate manner.
- Non-fixed time slice (virtual runtime)
 - CFS aims to fairly divide a CPU evenly among all competing processes.
- Weighting (Niceness)
 - Enables control over process priority by using nice value.
- Efficient data structure
 - Use red-black tree for efficient searching, insertion and deletion of a process.

Basic Operation

- **Virtual runtime** (`vruntime`)
 - Denote how long the process has been executing.
 - Per-process variable
 - Increase in proportion with **physical (real) time** when it runs.
 - CFS will pick the process with the **lowest `vruntime`** to run next.
- **`sched_latency`**
 - A typical value is 48 (milliseconds)
 - Process's time slice = `sched_latency` / (the number of process)

CFS Simple Example

- Example:
 - There are four jobs (A, B, C, D) running. Per-process time slice is 12 ms.
 - Job with lowest `vruntime` runs next (round-robin).
 - C and D complete after two time slices. Per-process time slice is now 24 ms.



- **min_granularity**

- Minimum time slice (e.g., 6 ms)
- Ensure that not too much time is spent in scheduling overhead when there are too many processes running.

Weighting

- **Nice** level of a process
 - CFS enables controls over process priority.
 - Nice parameter is integer value and can be set from -20 to +19 (with a default of 0).
 - Positive nice values imply **lower** priority and negative values imply **higher** priority.
 - The nice value is mapped to a **weight**.

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

Weighting (Niceness)

- New time slice formula

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

- Example:

- Assign job **A** a nice value of -5 and job **B** a nice value of 0 .

Job	Nice value	Weight	Time slice
A	-5	3121	36 ms
B	0	1024	12 ms

vruntime With Weight

- vruntime formula
 - Takes the actual run time and scales it inversely by the weight of process.

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

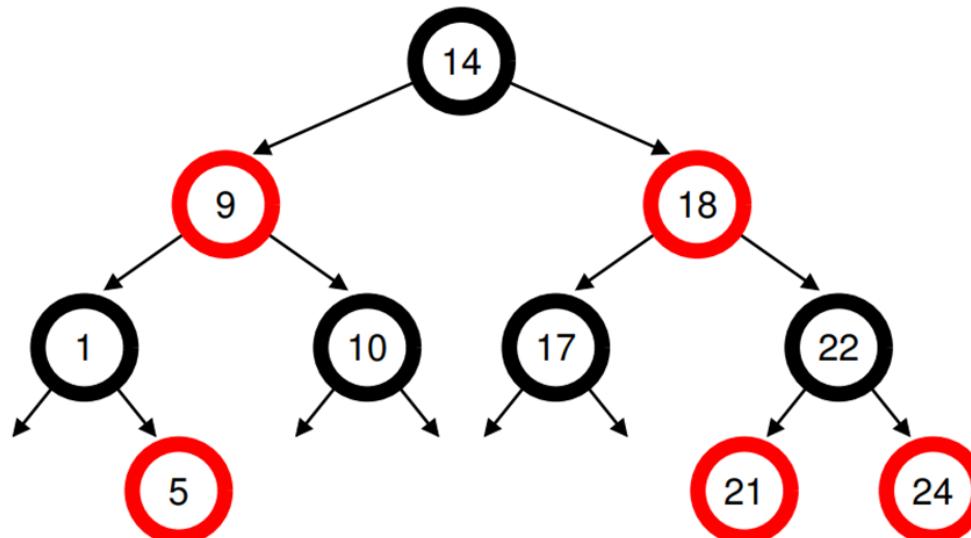
- Example:

Job	Nice value	Weight	Accumulated value
A	-5	3121	$1/3 \times \text{runtime}$
B	0	1024	$1 \times \text{runtime}$

Data Structure for Ready Queue

- **Red-Black Tree**

- Balanced binary tree (can address worst-case insertion pattern in simple binary tree).
- Ordering of **Red-Black Tree**: $O(\log n)$
- Efficiently find the process with minimum virtual runtime.
- Only running (or runnable) processes are kept therein.



Dealing With I/O and Sleeping Processes

- Example:
 - A runs continuously, and B has slept for 10 seconds.
 - When B wakes up, its vruntime will be 10 seconds less than A's, and B will **monopolize** the CPU for the next 10 seconds.
- Solution:
 - Set the vruntime of process to the **minimum value** found in tree when it wakes up. (**Avoid starvation**)
 - Processes that sleep for short periods of time frequently **do not ever** get their fair share of the CPU. (**Unfair share**)

Summary

- The concept of **proportional-share scheduling** have been introduced.
- Three approaches have been briefly discussed.
 - **Lottery scheduling**
 - Use randomness to achieve proportional share.
 - **Stride scheduling**
 - Achieve proportional share deterministically.
 - **Completely Fair Scheduler (CFS)**
 - Like weighted round-robin with dynamic time slices.
- Fair-share schedulers have their own problems. They do not particularly mesh well with I/O. They also leave open the hard problem of ticket or priority assignment.

Ch. 10. Multiprocessor Scheduling

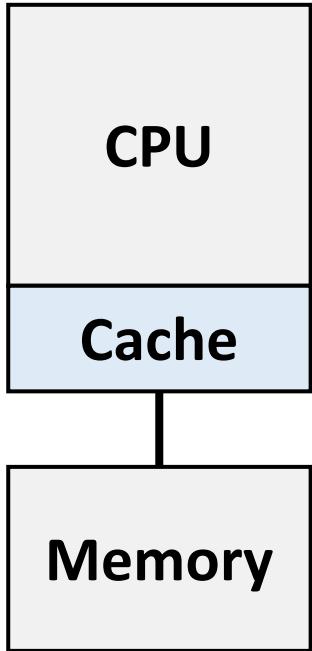
(OSTEP Ch. 10)

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

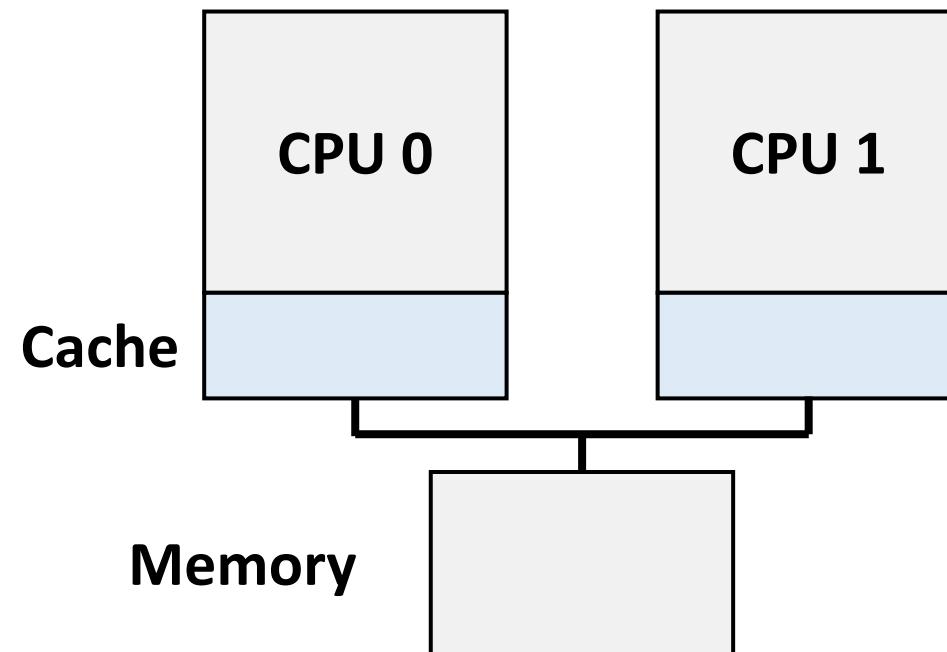


- **Cache**
 - Small, fast memories
 - Hold copies of **popular** data that is found in the main memory.
 - Utilize **temporal** and **spatial locality**
- **Main Memory**
 - Holds **all** of the data
 - Access to main memory is **slower** than cache.

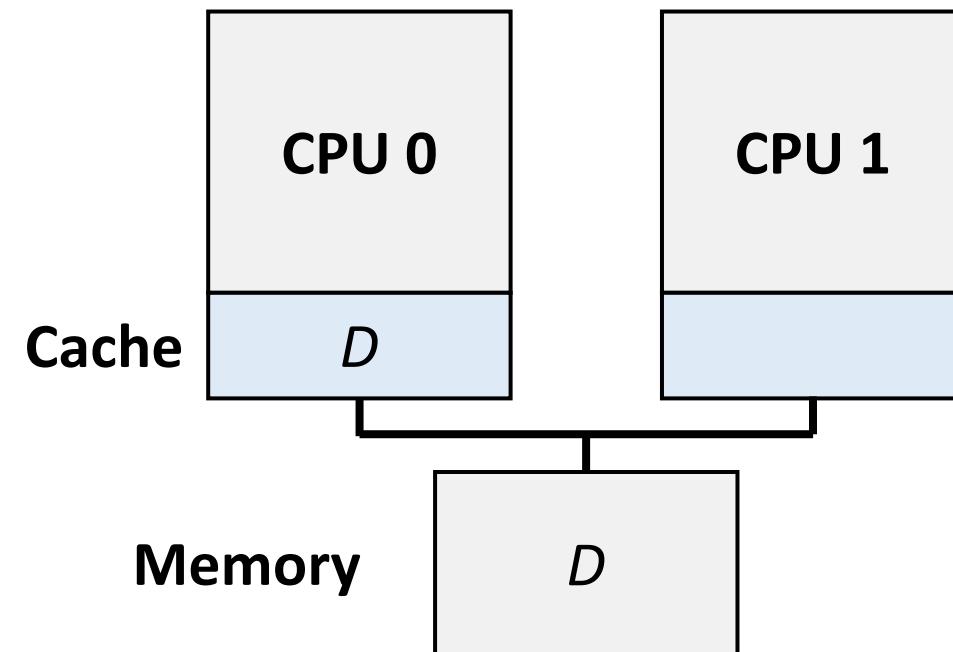
By keeping **frequently accessed** data in a cache, the system can make the **large, slow** memory appear to be a **fast** one.

Cache Coherence

- Consistency of shared resource data stored in multiple caches.



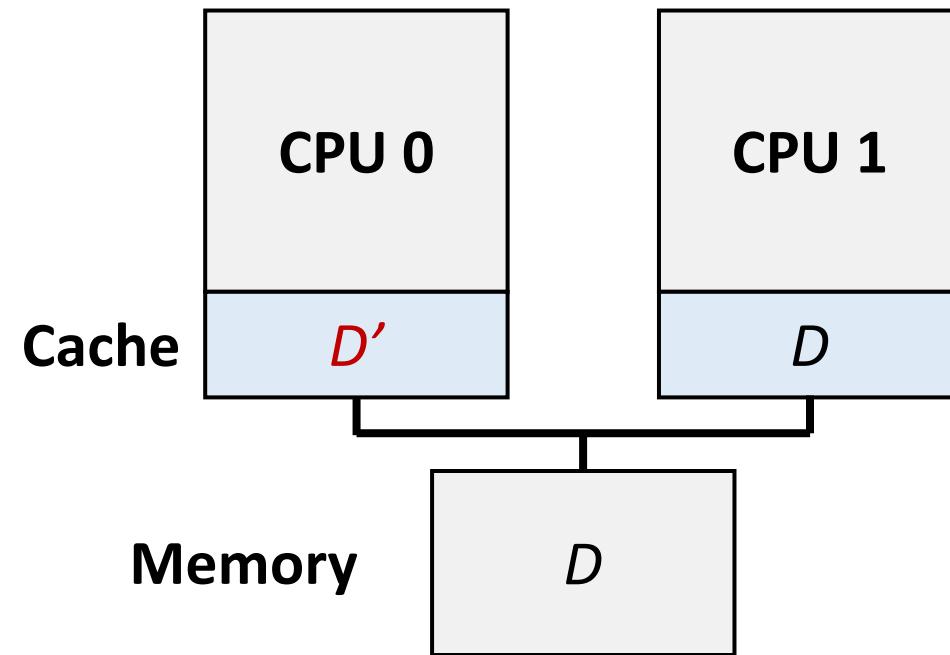
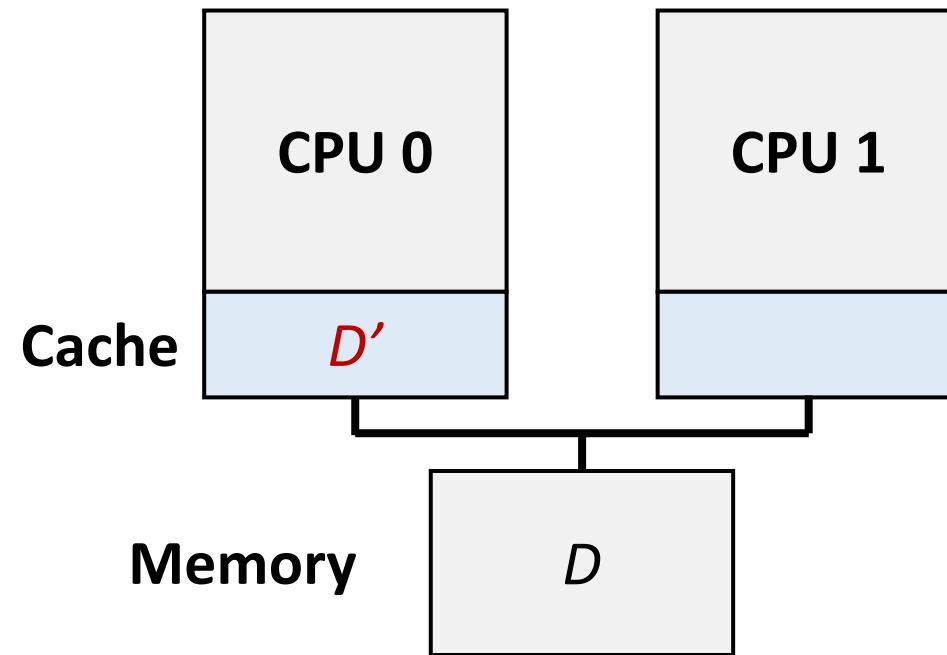
0. Two CPUs with caches share memory.



1. CPU 0 reads a data from memory.

Cache Coherence

- CPU may get the old value instead of the correct value.



2. CPU 0 updates the data (to its cache only). 3. CPU 1 re-reads the data from memory.

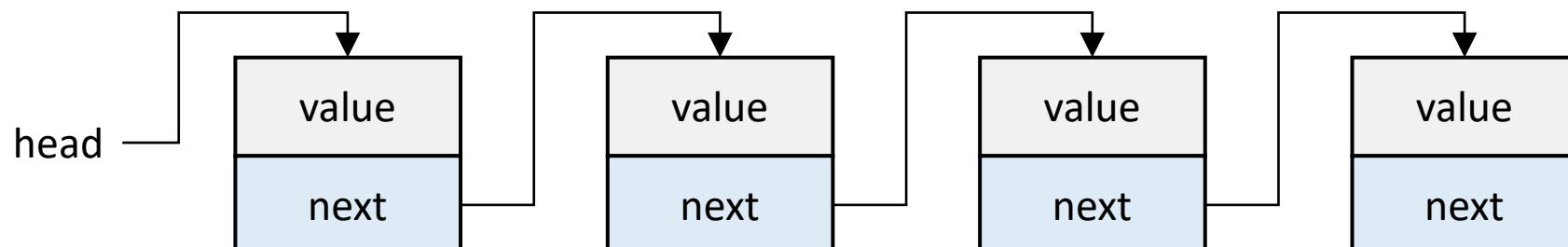
Cache Coherence Solution

- **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.

Synchronization

- When accessing shared data across CPUs, **mutual exclusion primitives** (e.g., locks) 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 }
```



Synchronization with Locks

```
1 pthread_mutex_t m;
2
3 typedef struct __Node_t {
4     int value;
5     struct __Node_t *next;
6 } Node_t;
7
8 int List_Pop() {
9     lock(&m);
10    Node_t *tmp = head;           // remember old head ...
11    int value = head->value;    // ... and its value
12    head = head->next;          // advance head to next pointer
13    free(tmp);                 // free old head
14    unlock(&m);
15    return value;               // return value at head
16 }
```

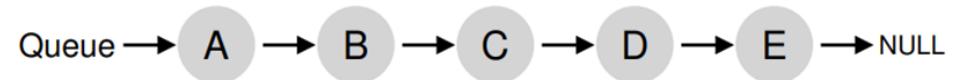
Cache Affinity

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

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

Single-Queue Multiprocessor Scheduling (SQMS)

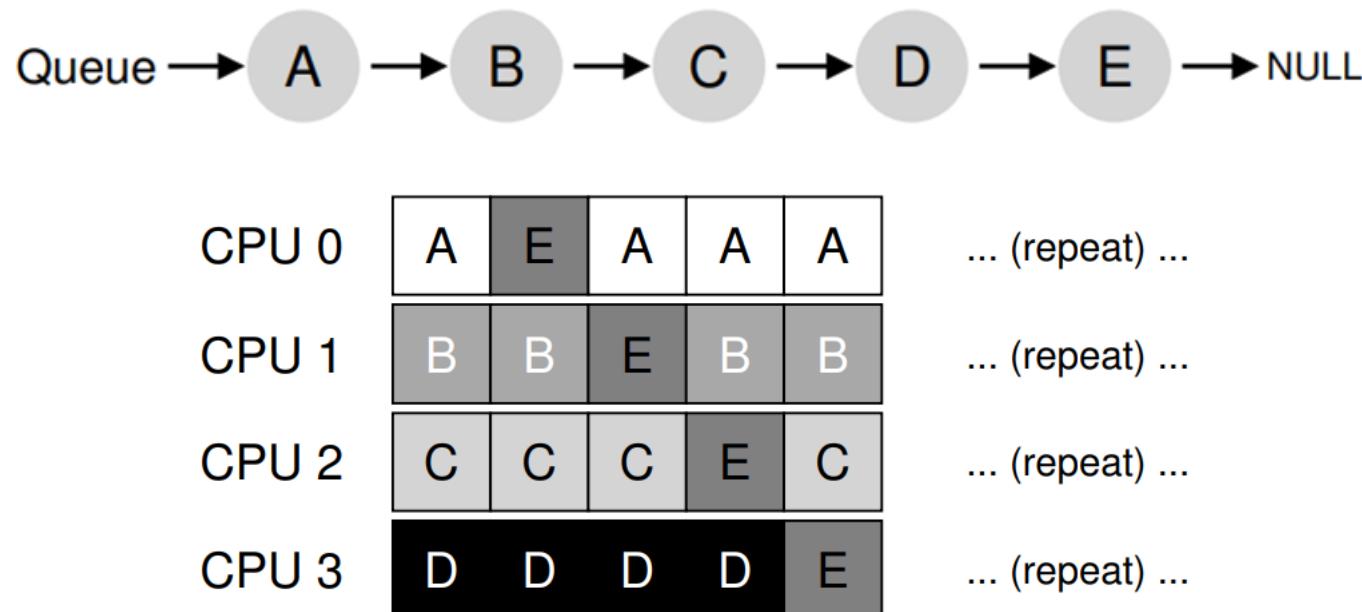
- Put all jobs that need to be scheduled into a **single queue**.
- Each CPU simply picks the next job from the globally shared queue.
- Cons:
 - Some form of **locking** have to be inserted → Lack of scalability
 - Cache affinity
- Example:
 - Possible job scheduler across CPUs:



CPU 0	A	E	D	C	B	... (repeat) ...
CPU 1	B	A	E	D	C	... (repeat) ...
CPU 2	C	B	A	E	D	... (repeat) ...
CPU 3	D	C	B	A	E	... (repeat) ...

SQMS Example With Affinity Mechanism

- **Preserving affinity** for most jobs.
 - Jobs A through D are not moved across CPUs.
 - Only job E is **migrating** from CPU to CPU.
- Implementing such a scheme, however, can be complex.

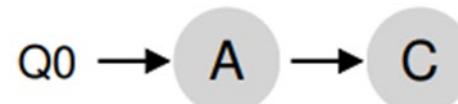


Multi-Queue Multiprocessor Scheduling (MQMS)

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

MQMS Example

- Two CPUs. Each CPU has a scheduling queue.
- Four jobs (**A**, **B**, **C** and **D**) enter the system.



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

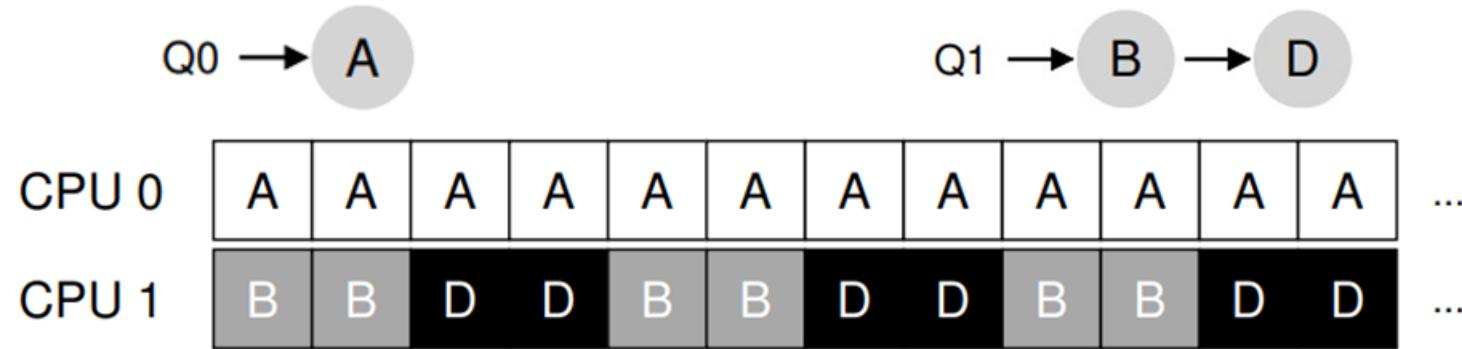
CPU 0	A	A	C	C	A	A	C	C	A	A	C	C	...
CPU 1	B	B	D	D	B	B	D	D	B	B	D	D	...

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

Load Imbalance in MQMS

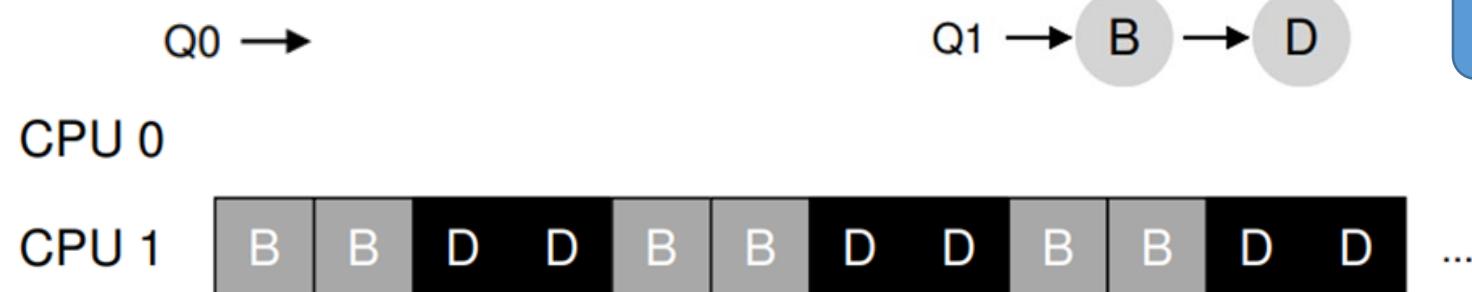
- After job **C** finishes

A gets twice as much CPU as B and D.



- After job **A** finishes

CPU 0 is idle!



How To Deal With Load Imbalance

- Move jobs around, a technique which is referred to as **migration**.
- Example 1:



- Move **B** or **D** to CPU 0

- Example 2:



- **Continuous migration** of one or more jobs.



Work Stealing

- A basic approach on deciding which jobs to migrate.
- 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:
 - Suffer from **high overhead** and have trouble **scaling**.

Linux Multiprocessor Schedulers

- **O(1)**
 - A priority-based scheduler (similar to MLFQ)
 - Uses multiple queues.
 - Changes a process's priority over time.
 - Schedules those with highest priority.
 - Interactivity is a particular focus.
- **Completely Fair Scheduler (CFS)**
 - Deterministic proportional-share approach (more like Stride scheduling)
 - Uses multiple queues.

Linux Multiprocessor Schedulers

- **BF Scheduler (BFS)**
 - Uses single-queue approach.
 - Proportional-share
 - Based on Earliest Eligible Virtual Deadline First (EEVDF)

SQMS vs MQMS Summary

- **SQMS**
 - Rather **straightforward** to build and **balances load** well.
 - But inherently has difficulty with **scaling** to many processors and **cache affinity**.
- **MQMS**
 - **Scales** better and handles **cache affinity** well.
 - But has trouble with **load imbalance** and is **more complicated**.

COMP 4736

Introduction to Operating Systems

05. Memory Management I
(OSTEP Ch. 13 & 14)

Ch. 13. Address Spaces

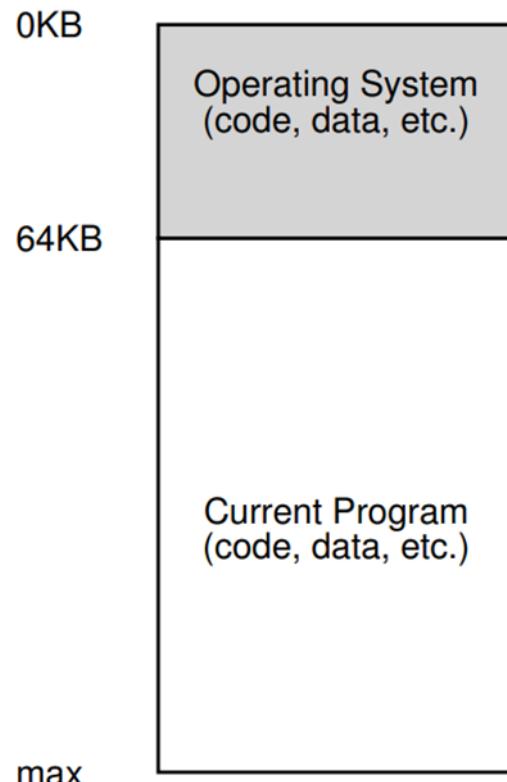
(OSTEP Ch. 13)

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 .

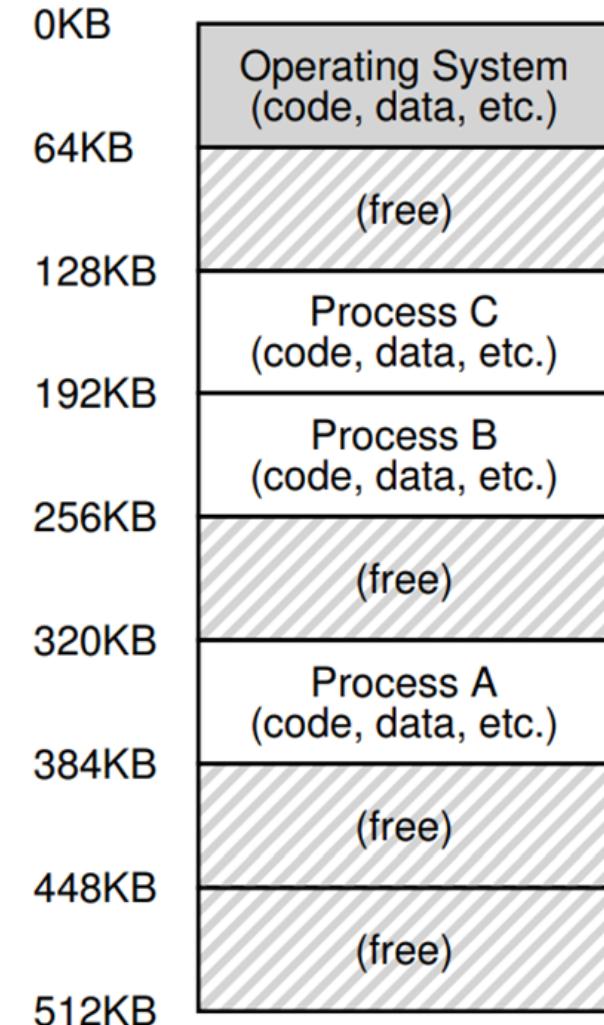
OS in the Early Days

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



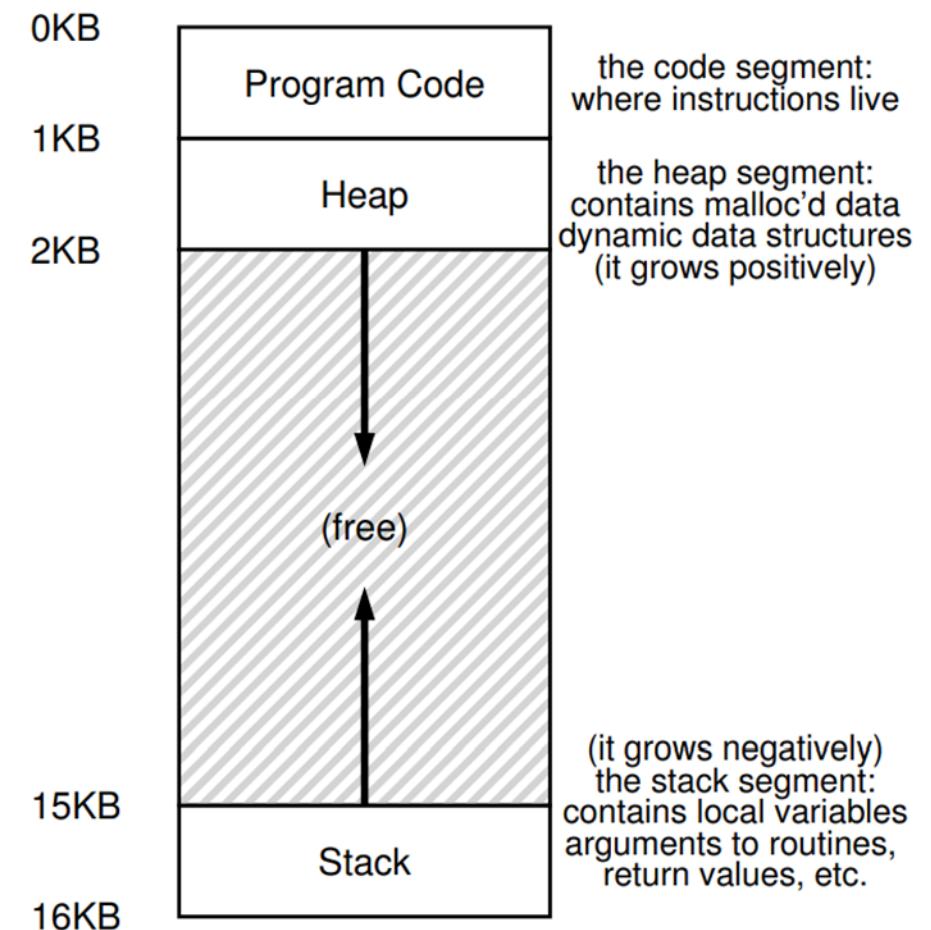
Multiprogramming and Time Sharing

- Load **multiple processes** in memory.
 - Execute one for a short while.
 - Switch processes between them in memory.
 - Increase **utilization** and **efficiency**.
- **Protection** becomes an important issue.
 - Need to prevent memory accesses from other processes.



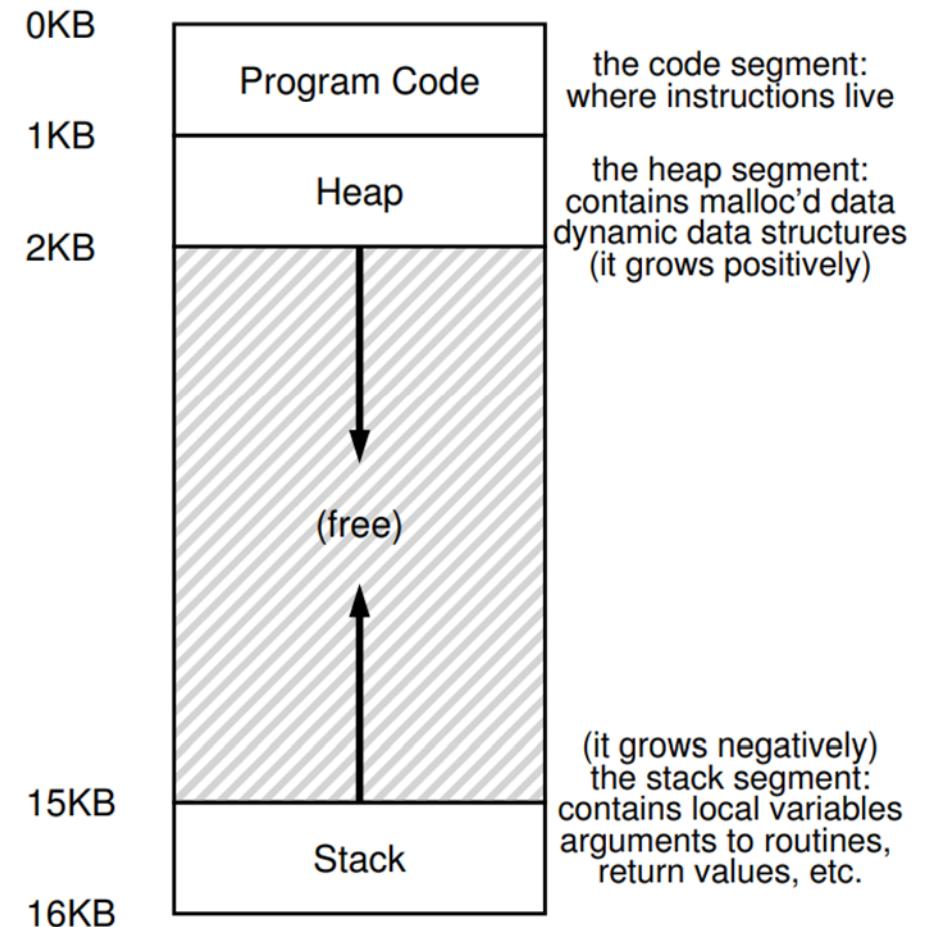
Address Space

- 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

- **Code**
 - Where instructions live
- **Heap**
 - Used for dynamically allocated memory.
 - `malloc()` in C language
 - `new` in object-oriented language
- **Stack**
 - Stores return addresses or values.
 - Contains local variables and arguments to routines.



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", main);
    printf("location of heap : %p\n", malloc(100e6));
    int x = 3;
    printf("location of stack: %p\n", &x);
    return x;
}
```

- Output

```
location of code : 0x7f9e6ce1f189
location of heap : 0x7f9e66c30010
location of stack: 0x7ffff9b3d6f4
```

Goals of Memory Virtualization

- **Transparency**
 - The implementation should be invisible to running programs.
 - User programs should not be aware of the messy details.
- **Efficiency**
 - Minimize overhead and wastage in terms of memory space and access time.
- **Isolation and Protection**
 - A user process should not be able to access anything outside its address space.

Summary

- **Virtual memory**, a major OS subsystem, is responsible for providing the illusion of a large, sparse, private address space to **each** running program.
- Each **virtual address space** contains all of a program's instructions and data.
- The OS converts the **virtual memory references** into **physical address** with hardware help.
- The OS will provide the translation for many processes at once, making sure to **protect** programs from one another as well as to protect the OS.

Ch. 14. Memory API

(OSTEP Ch. 14)

Types of Memory

- **Stack** Memory
 - Also called **automatic** memory
 - They are managed **implicitly** by the compiler.

```
void func() {  
    int x; // declares an integer on the stack  
    ...  
}
```

- Compiler makes space on the stack when `func()` is called.
- It deallocates the memory when the function is returned.
- That is why local variables have limited scopes.

Types of Memory

- **Heap** Memory
 - Allocations and deallocations are **explicitly** handled by programs.

```
void func() {  
    int *x = (int *) malloc(sizeof(int));  
    ...  
}
```

- An integer is malloc'd on the **heap**.
- Its address is returned and stored on the **stack**.

The malloc() Call

```
#include <stdlib.h>
...
void *malloc(size_t size);
```

- Allocates 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
 - On **success**: a void type pointer to the memory block allocated by **malloc**
 - On **failure**: 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
 - When the actual size of 'x' is known at **compile-time**.

```
int x[10];
printf("%d\n", sizeof(x));           // 64-bit system
40                                     // returning size of array (sizeof(int) = 4)
```

- When the actual size of 'x' is known at **run-time**.

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));           // 64-bit system
8                                     // returning size of the pointer
```

The free() Call

```
#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

Forgetting to Allocate Memory

- Erroneous code:

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

- Proper code:

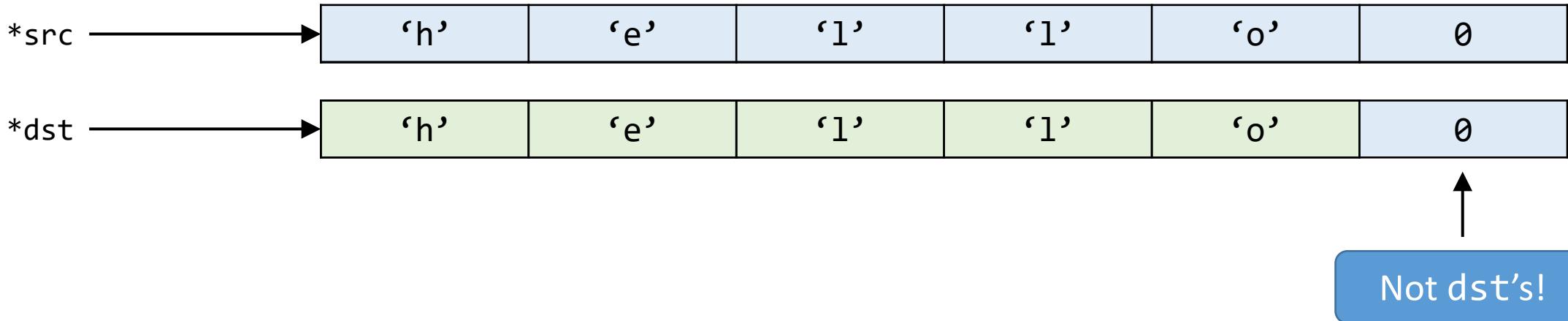
```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

Not Allocating Enough Memory

- Code in question:

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // seems to work
```

- Problem:



Forgetting to Initialize Allocated Memory

- Uninitialized read

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

- What is the data pointed by x?
 - At best, it might be **meaningless** (e.g., zero).
 - Or, it can be something **random** and harmful.
 - Or, the program **intentionally** tries to read the memory content that is written by another process (a typical malicious attack).

Forgetting to Free Memory

- **Memory leak**

```
while (1)  
    malloc(4);
```

- A program keeps allocating memory without freeing it.
- A program runs out of memory and eventually is killed by OS.
- Note:
 - OS **reclaims** all the memory of the process when it **exits**. So, no memory is really “**lost**”.
 - But in long-running programs (e.g., web server or DBMS), leaked memory is a much bigger issue, and will lead to a crash.

Other Errors in Freeing Memory

- **Freeing memory before you are done with it**
 - The freed memory is no longer valid (i.e., a **dangling pointer**).
- **Freeing memory repeatedly**
 - Since the freed memory is no longer valid, **double freeing** it would cause undefined results.
- **Calling free() incorrectly**
 - The `free()` call can only accept a pointer returned by `malloc()` earlier.
 - Freeing other values will generate unexpected results.

Other Memory APIs: `calloc()` and `realloc()`

```
#include <stdlib.h>
void *calloc(size_t num, size_t size);
```

- Allocate memory and **zeroes** it before returning.
 - `size_t num`: number of objects to allocate
 - `size_t size`: size of each object (in bytes)

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

- **Change** the size of memory block.
 - `void *ptr`: Pointer of memory block allocated with `malloc`, `calloc` or `realloc`
 - `size_t size`: new size of the memory block (in bytes)

Underlying OS Support

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

- Not enough heap space → Ask OS to expand heap.
- **System calls:** brk and sbrk
- brk is called to change the location of the program's **break**.
 - **break:** The location of the end of the heap.
- sbrk is similar to brk but takes an argument of memory increment.
- Library calls (e.g., malloc and free) use these system calls.
- Programmers should **never** directly call either brk or sbrk.

COMP 4736

Introduction to Operating Systems

06. Memory Management II

(OSTEP Ch. 15 & 16)

Ch. 15. Address Translation

(OSTEP Ch. 15)

Virtualizing Memory

- Recall in **CPU virtualization**, we use **limited direct execution (LDE)**
 - Let the program run directly on the hardware. (**Efficient**)
 - Interpose at certain critical points in time such as at system call or timer interrupt. (**Control**)
- Again, we try to attain both efficiency and control in **memory virtualization**
 - **Efficiency**: Use hardware support (e.g., registers, TLBs, page tables).
 - **Control**: No application (including OS) is allowed to access any memory but its own.
 - **Flexibility**: Programs should be able to use their address spaces in whatever way they would like

Address Translation

- Hardware transforms a **virtual** address to a **physical** address.
 - The desired information is actually located at a physical address.
- The OS must get involved at key points to set up the hardware. It must **manage memory** by
 - keeping track of which locations are free and in use
 - judiciously intervening to maintain control over how memory is used

Assumptions

- The user's address space must be placed **contiguously** in physical memory.
- The size of the address space is **less than** the size of physical memory.
- Each address space is exactly the **same size**.

Address Translation Example

- C-language code

```
void func() {  
    int x = 3000;  
    ...  
    x = x + 3;    // line of code we are interested in
```

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

Address Translation Example

- 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
```

- Assume the address of x is stored in register ebx.
- **Load** the value at that address into register eax.
- **Add** three to eax.
- **Store** the value in eax back into memory.

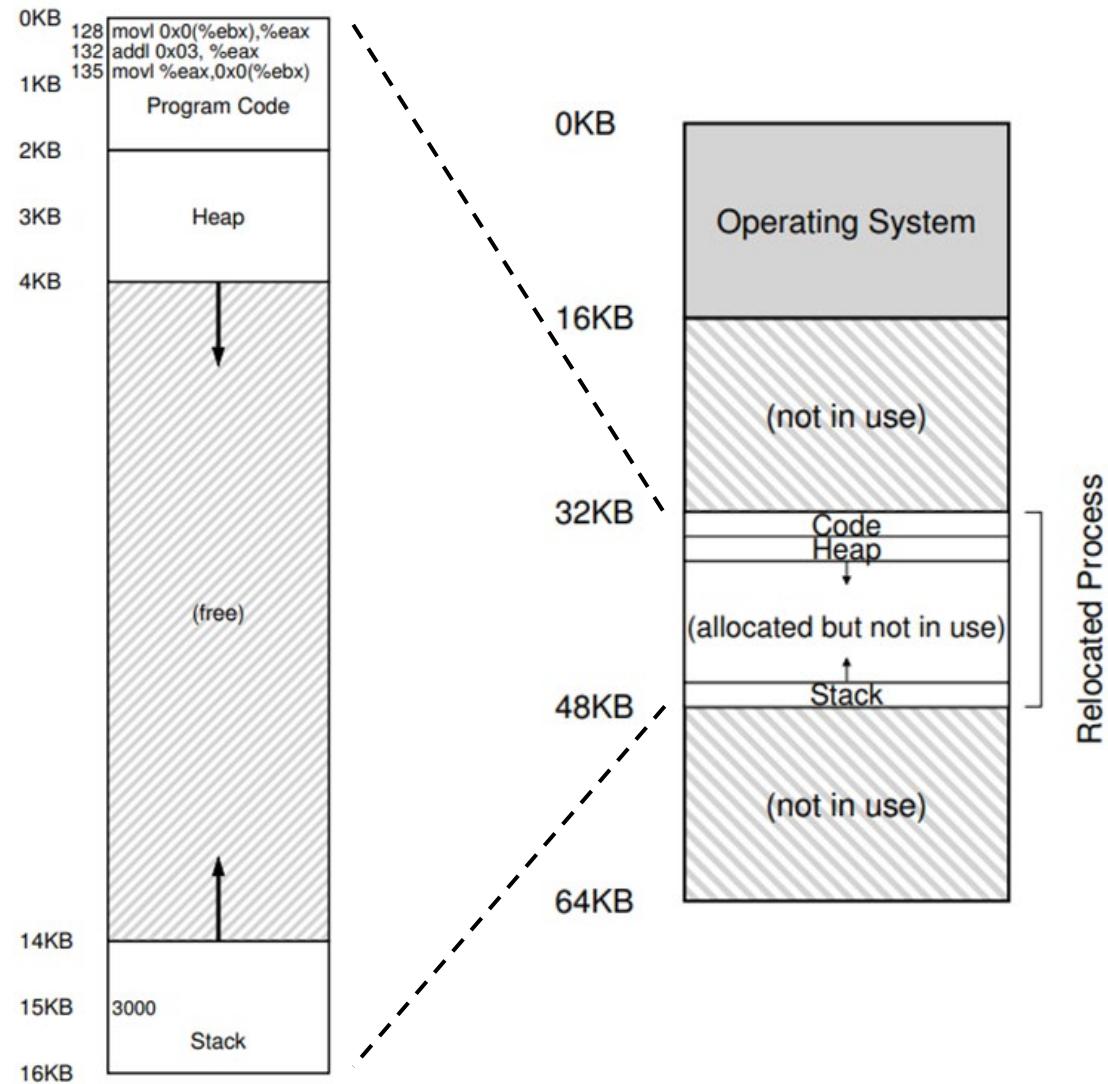
Address Translation Example

- Fetch instruction at address 128
- Execute this instruction (load from address 15 KB)
- 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)



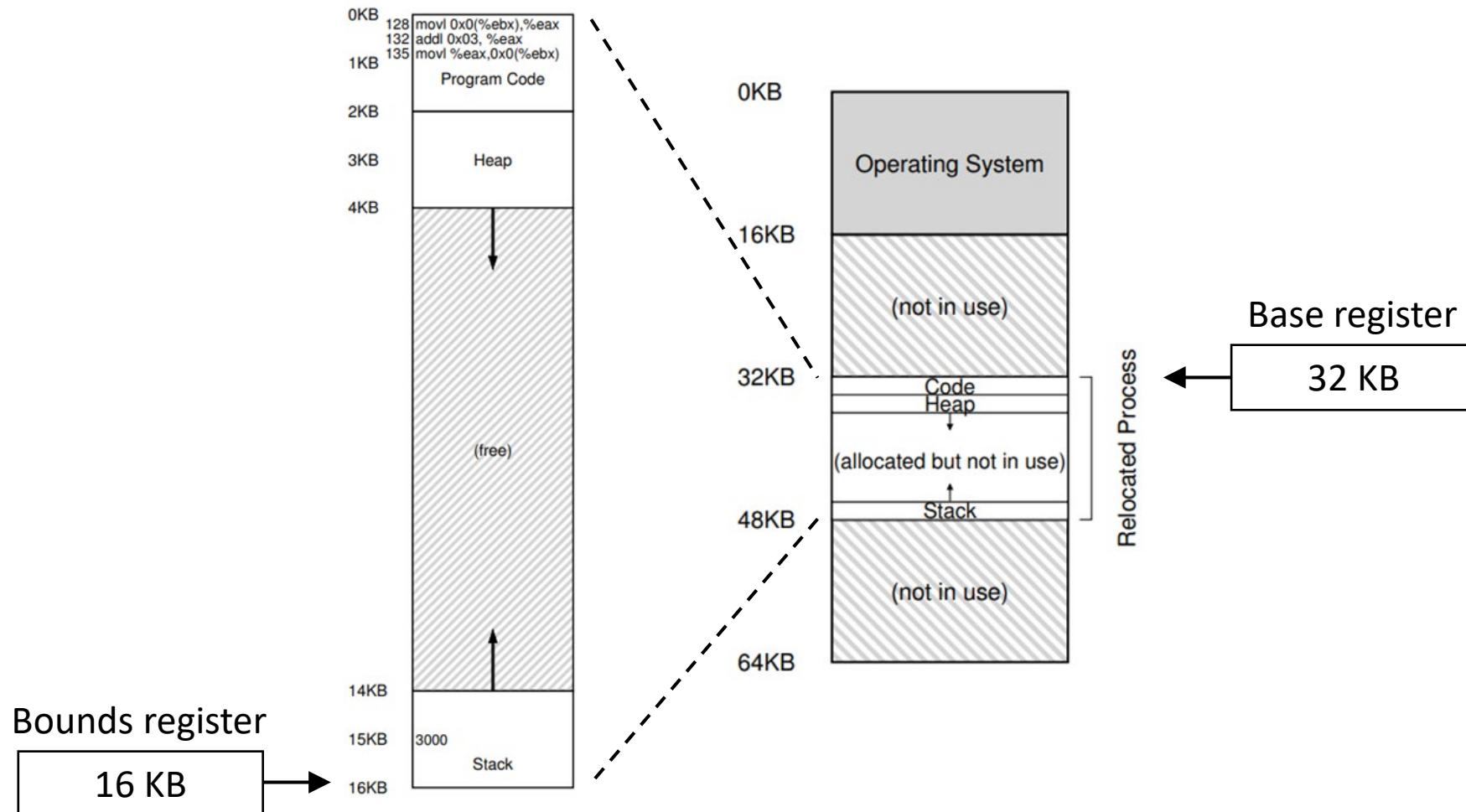
Process Relocation

- From the program's perspective, its **address space** starts at address 0 and grows to a maximum of 16 KB.
- OS may want to place the process somewhere else in physical memory, not necessarily at address 0.
- How can we **relocate** this process in memory in a way that is **transparent** to the process?



Dynamic (Hardware-based) Relocation

- Base and bounds (or dynamic relocation)



Dynamic (Hardware-based) Relocation

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

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

- Each virtual address must be within the bounds and non-negative.

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

Relocation and Address Translation

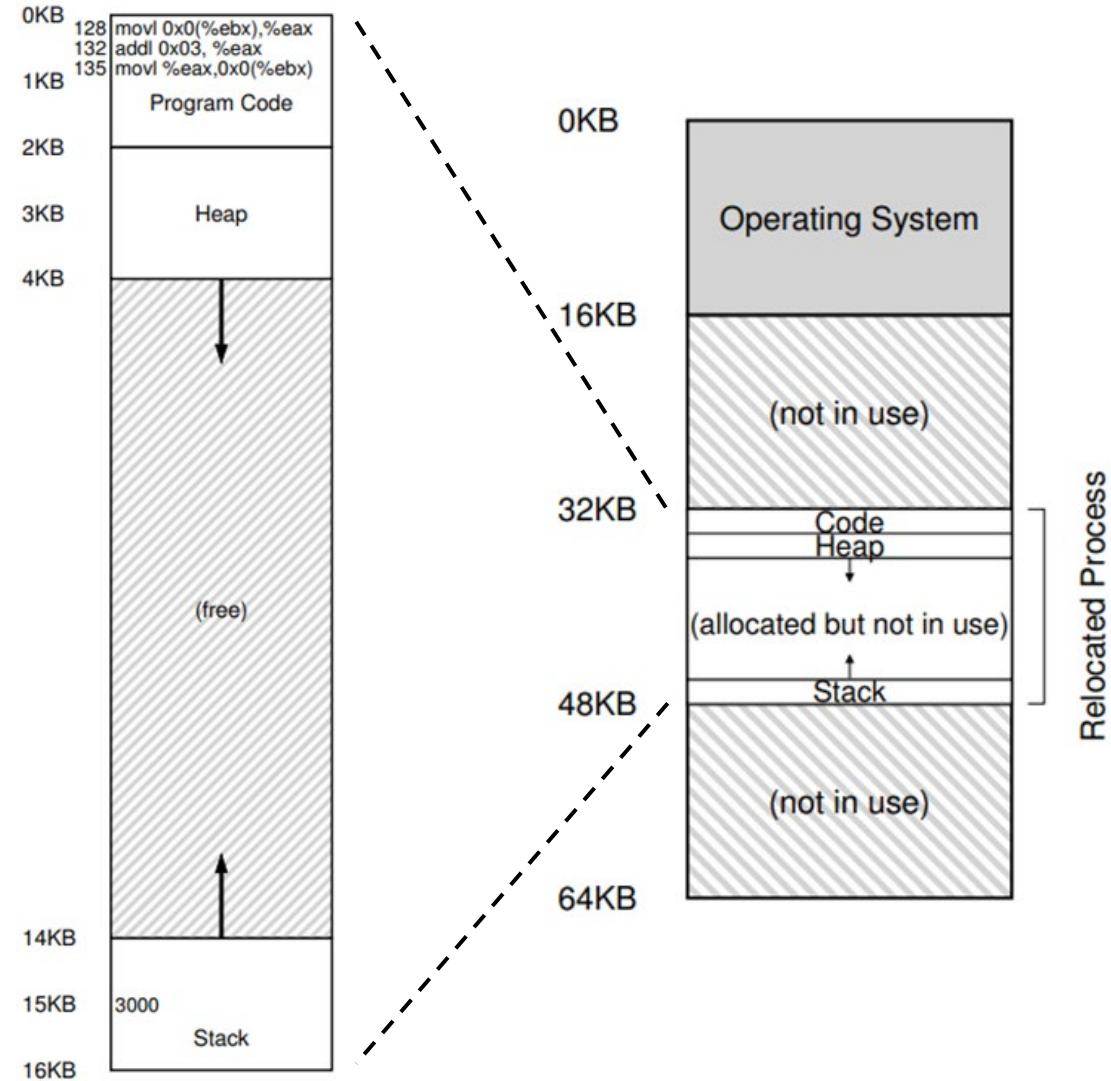
```
128: movl 0x0(%ebx), %eax
```

- Fetch instruction at address **128**

Virtual address	Base	Physical address
128	+ 32 KB (32768)	= 32896

- Execute this instruction (load from address **15 KB**)

Virtual address	Base	Physical address
15 KB	+ 32 KB	= 47 KB



Hardware Support

- **Base** and **bounds** registers are hardware structures on CPU.
- Sometimes the part of the processor that helps with address translation is called the **memory management unit (MMU)**.
- Two ways to define bound registers

- Bounds = **size**



- Bounds = **end** of physical address



Dynamic Relocation: Hardware Requirements

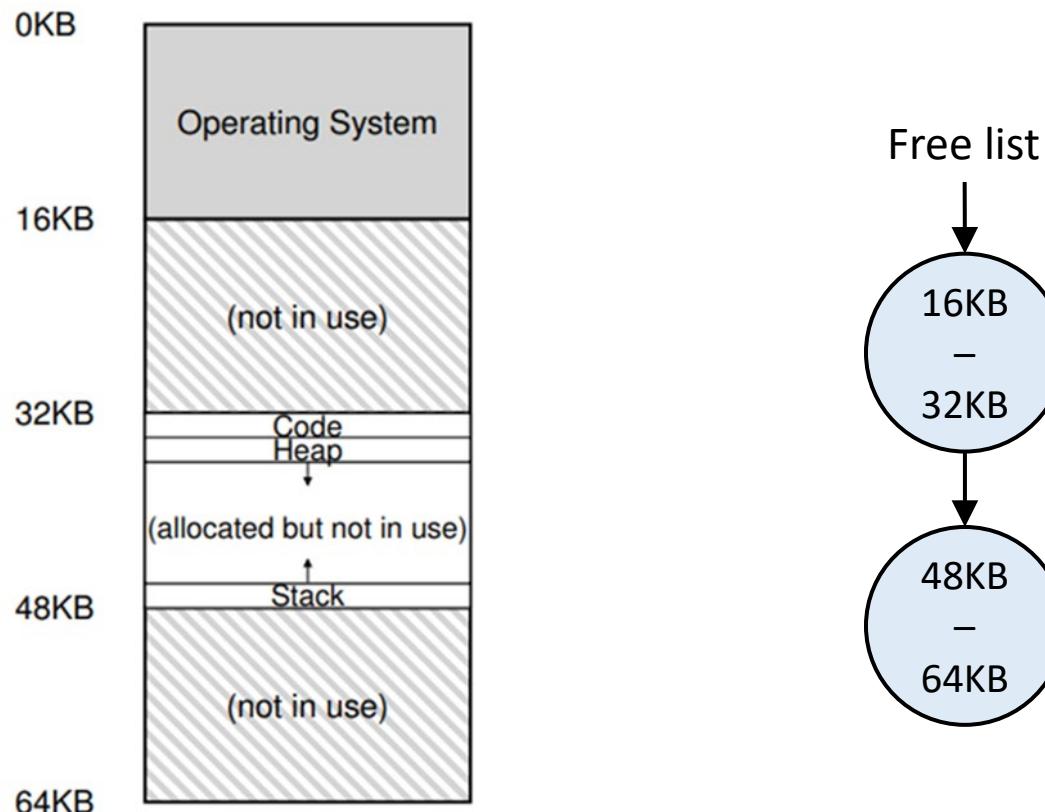
Hardware Requirements	Notes
Privileged mode	Needed to prevent user-mode processes from executing privileged operations
Base/bounds registers	Need pair of registers per CPU to support address translation and bounds checks
Ability to translate virtual addresses and check if within bounds	Circuitry to do translations and check limits; in this case, quite simple
Privileged instruction(s) to update base/bounds	OS must be able to set these values before letting a user program run
Privileged instruction(s) to register exception handlers	OS must be able to tell hardware what code to run if exception occurs
Ability to raise exceptions	When processes try to access privileged instructions or out-of-bounds memory

Operating System Issues

- The OS must take action to implement base-and-bounds approach.
- When a process **starts running**:
 - Find space for address space in physical memory.
- When a process is **terminated**:
 - Reclaim the memory for use.
- When **context switch** occurs:
 - Save and store the base-and-bounds pair.
- When **exception** is raised by hardware (e.g., out-of-bounds access):
 - Have exception handlers in place.

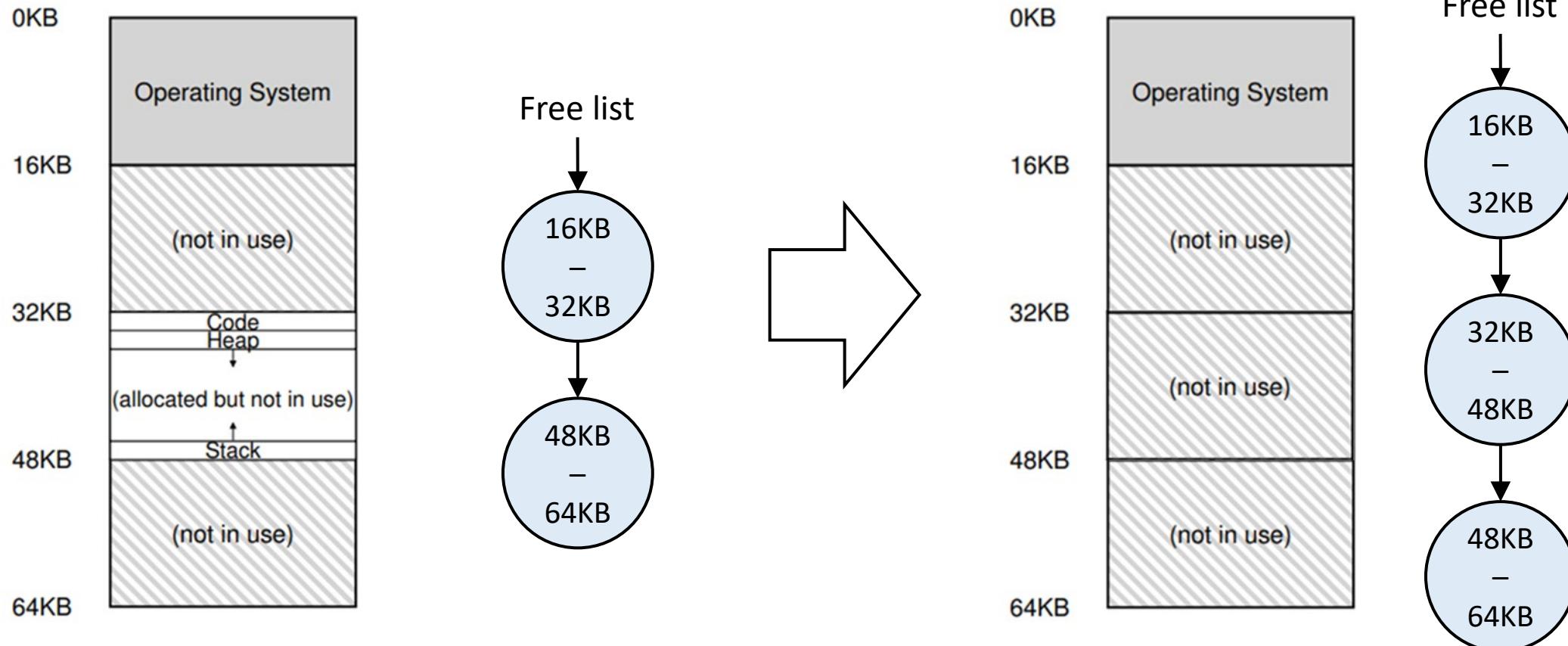
OS Issues: When a Process Starts Running

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



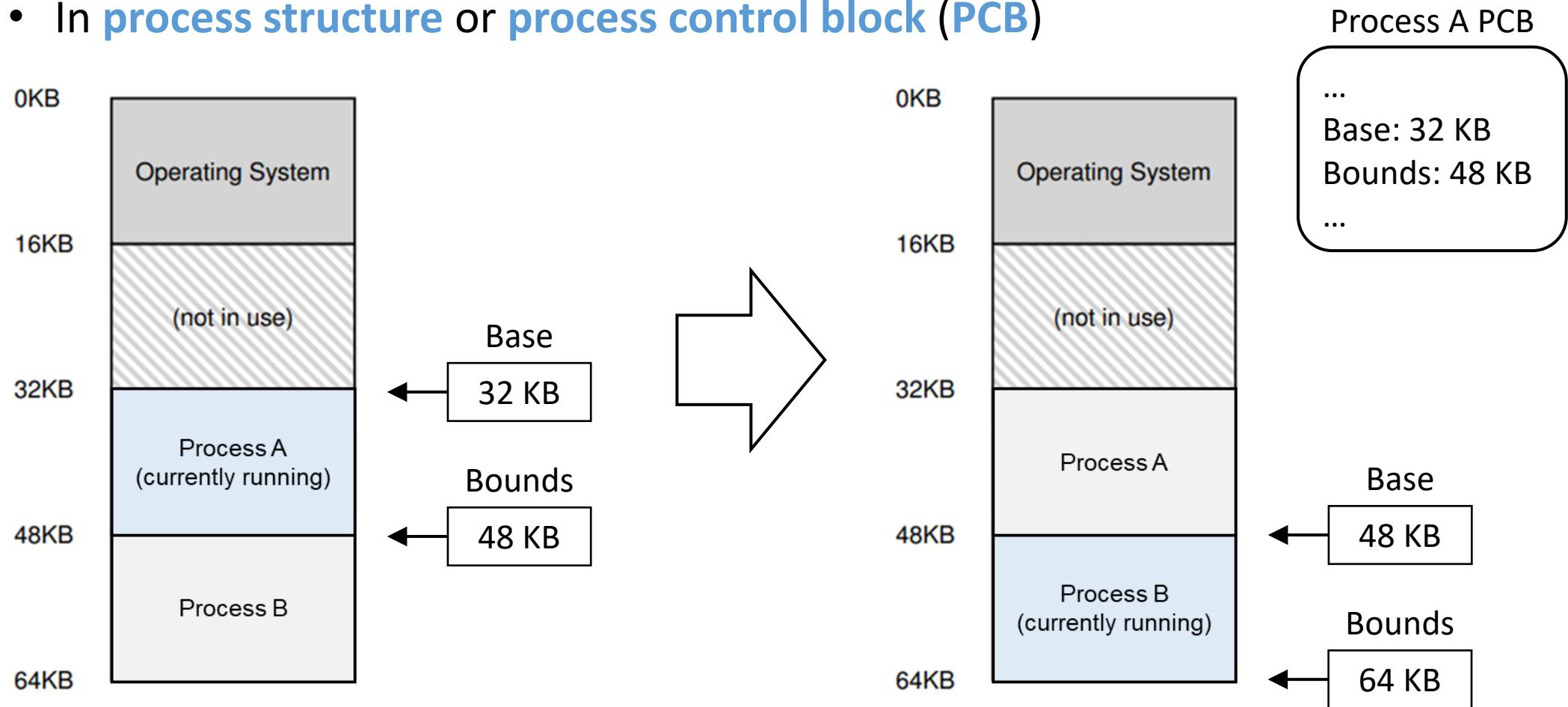
OS Issues: When a Process Is Terminated

- 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)**



OS Issues: Provide Exception Handlers

- The OS must provide **exception handlers**, and it installs these handlers at **boot time** via **privileged instructions**.
 - Exception handler for **segmentation fault**.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of ... syscall handler timer handler illegal mem-access handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table initialize free list		

Summary

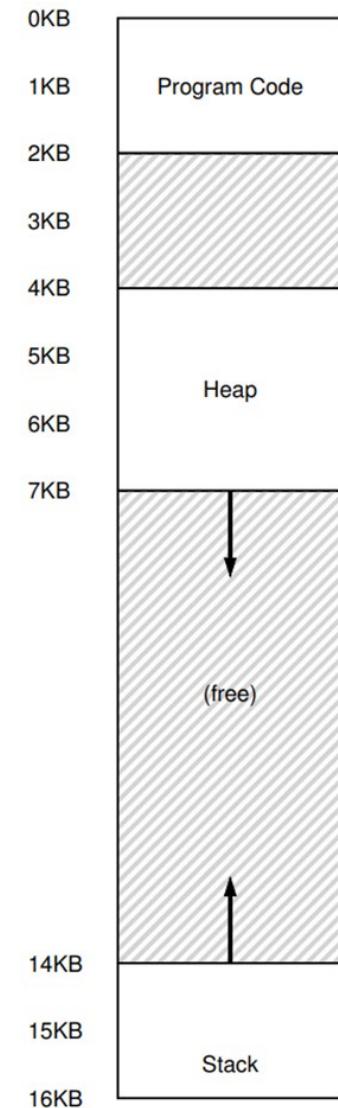
- We have introduced a mechanism used in virtual memory known as **address translation**.
- With address translation, the OS can **control** each and every memory access from a process, **ensuring** the accesses stay within the bounds of the address space.
- We have also seen one particular form of virtualization, known as **base and bounds** or **dynamic relocation**. It is **efficient** and offers **protection**.
- But it has its inefficiencies, such as internal fragmentation.

Ch. 16. Segmentation

(OSTEP Ch. 16)

Inefficiency of Base-and-Bounds Approach

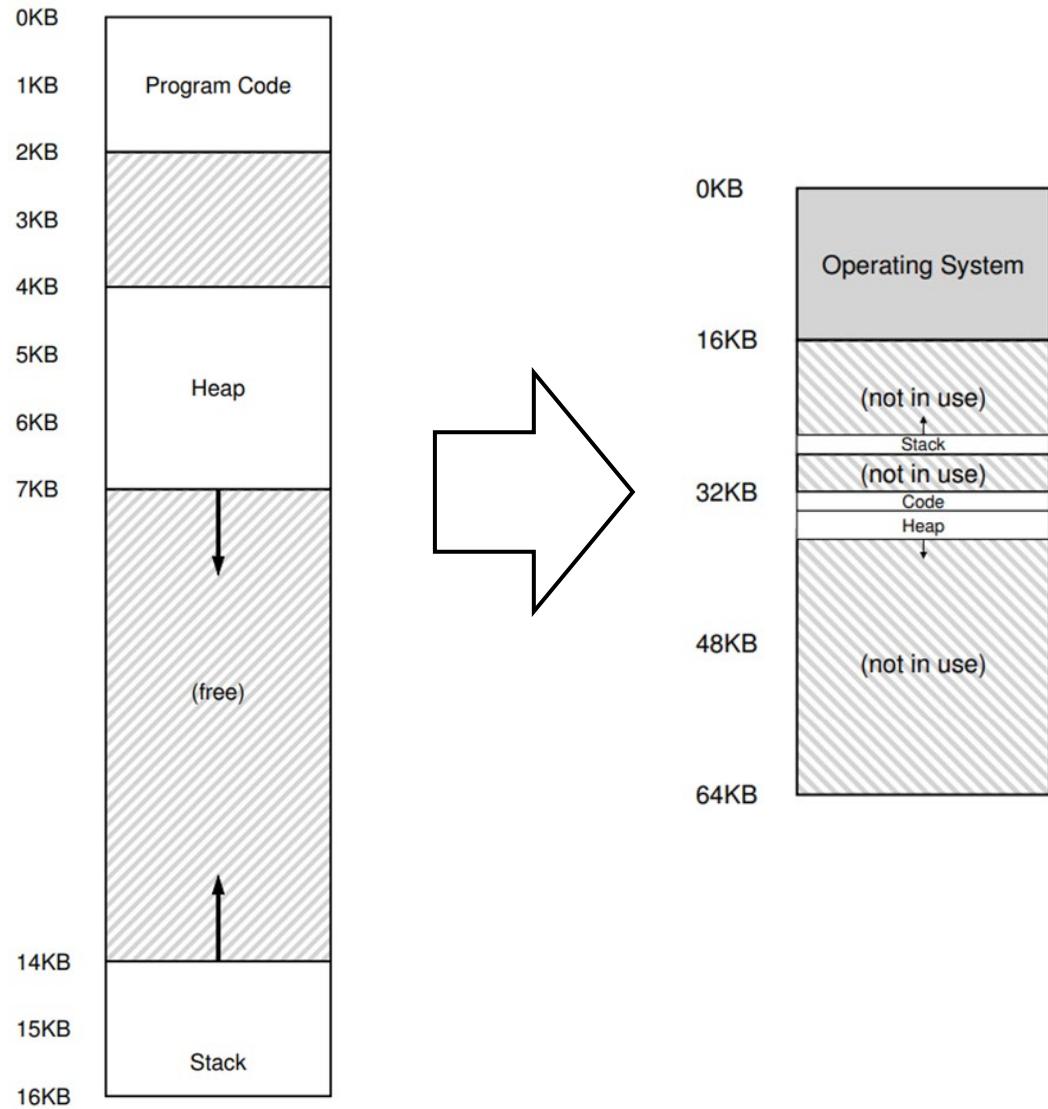
- A **big chunk** of “free” space between stack and heap.
- Although it is not being used, it still **takes up** physical memory.
- It is also hard to run a program when the entire address space **does not fit** into physical memory.



Segmentation

- A **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**.
 - Each **segment** has its **base and bounds**.

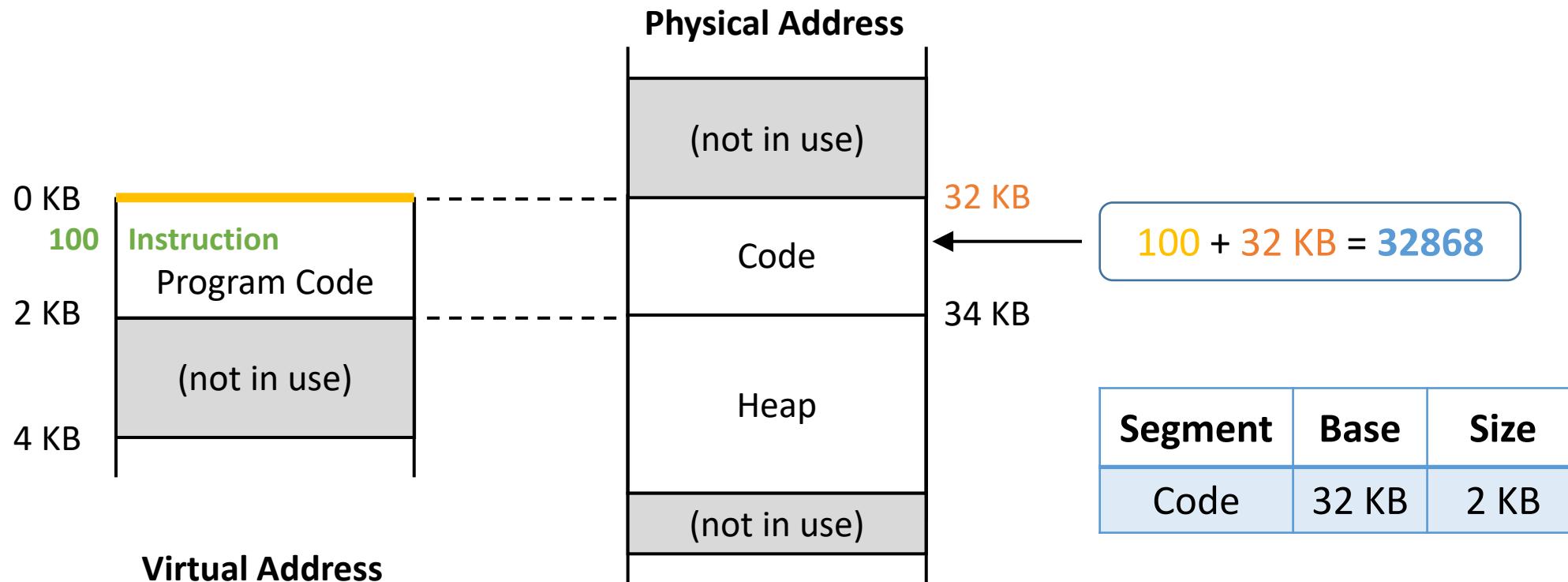
Placing Segments in Physical Memory



Segment	Base	Size
Code	32 KB	2 KB
Heap	34 KB	3 KB
Stack	28 KB	2 KB

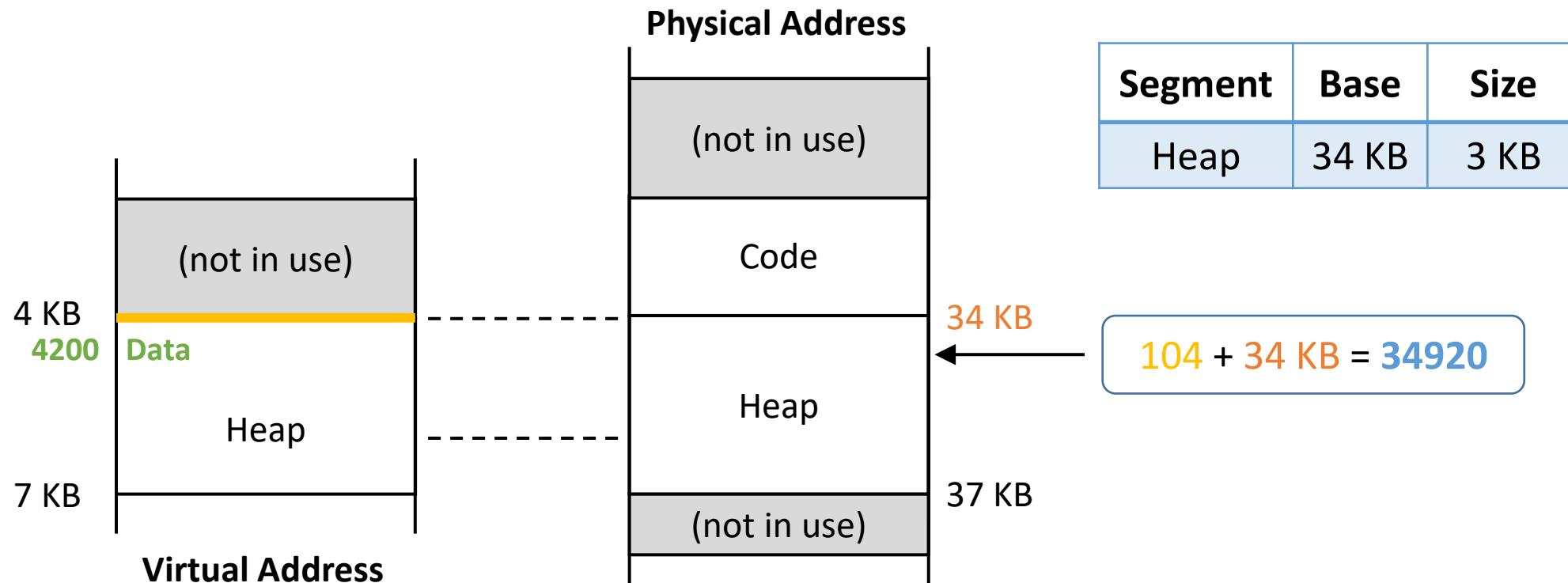
Address Translation on Segmentation: Code

- Translate **virtual address 100** to **physical address**.
 - It is in the **code segment**, which starts at virtual address **0**.
 - Offset into the segment = **100**
 - Physical address = **Offset + Base** = **100 + 32 KB = 32868**



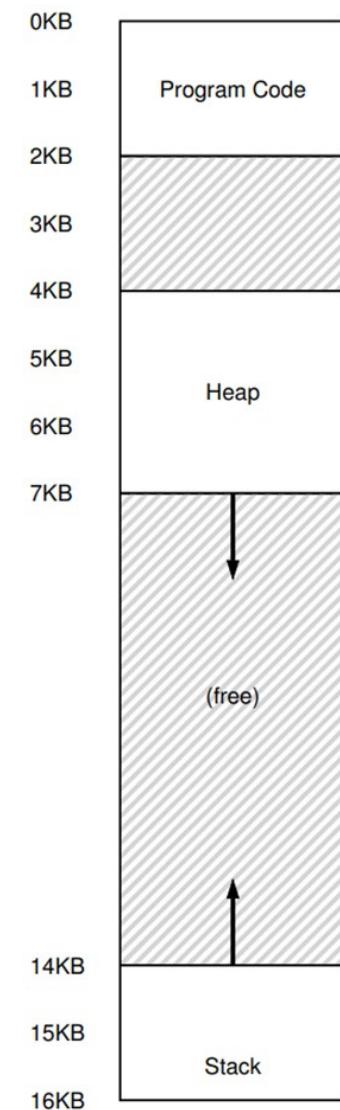
Address Translation on Segmentation: Heap

- Translate **virtual address 4200** to **physical address**.
 - It is in the **heap segment**, which starts at virtual address **4 KB**.
 - Offset into the segment = $4200 - 4 \text{ KB} = 104$
 - Physical address = **Offset + Base** = $104 + 34 \text{ KB} = 34920$



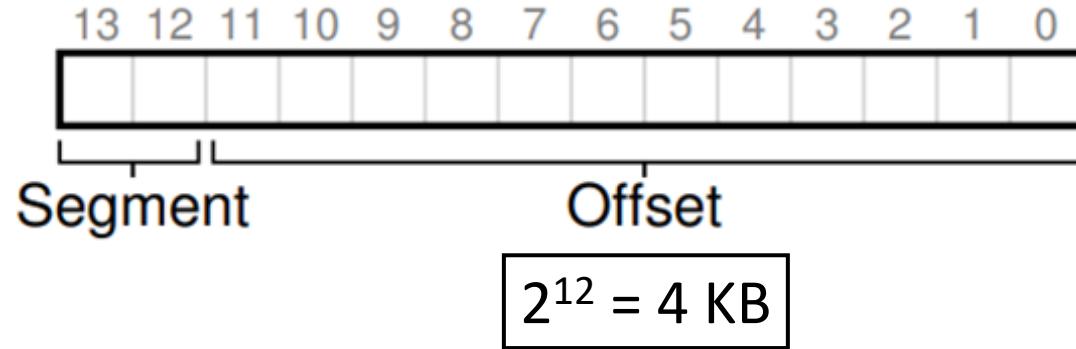
Segmentation Violation or Fault

- If an **illegal** virtual address of 7 KB or greater is referenced...
 - The hardware detects that the address is **out of bounds**.
 - Results in **segmentation violation** or **segmentation fault**.

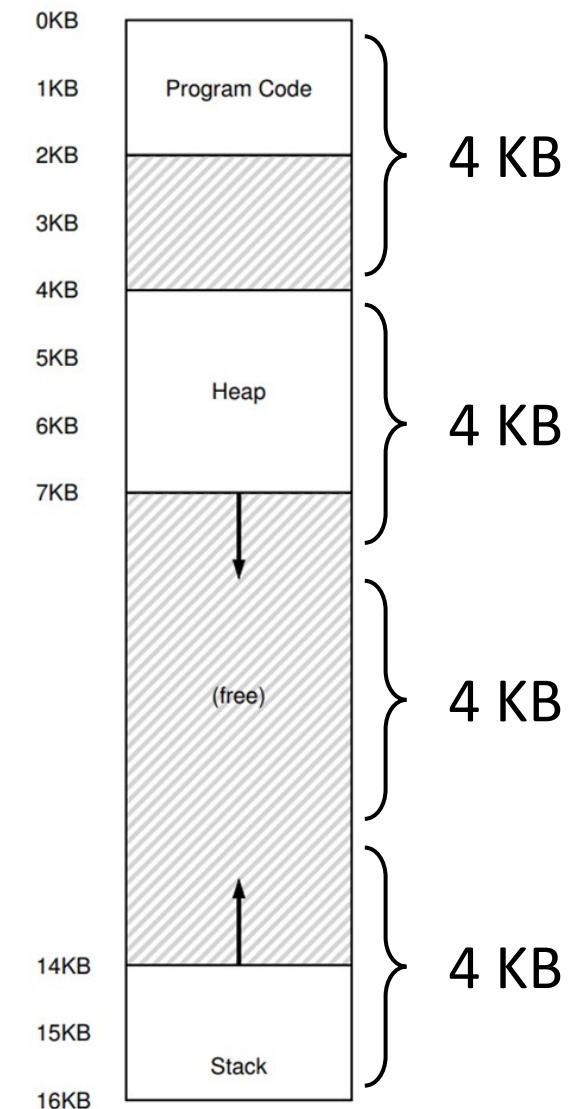


Determining Segment and Offset

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

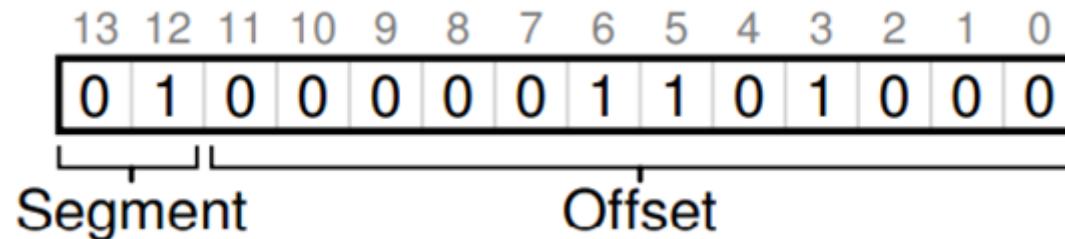


Segment	Top two bits
Code	00
Heap	01
Stack	11



Determining Segment and Offset

- Example
 - VA **4200** = 0b01 0000 0110 1000



Segment	Top two bits
Code	00
Heap	01
Stack	11

- Segment = 01 = **Heap**
- Offset = 0x068 = **104**

Physical Address Translation

```
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 (0b11 0000 0000 0000)
- **SEG_SHIFT** = 12
- **OFFSET_MASK** = 0xFFFF (0b00 1111 1111 1111)

Stack Segment

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

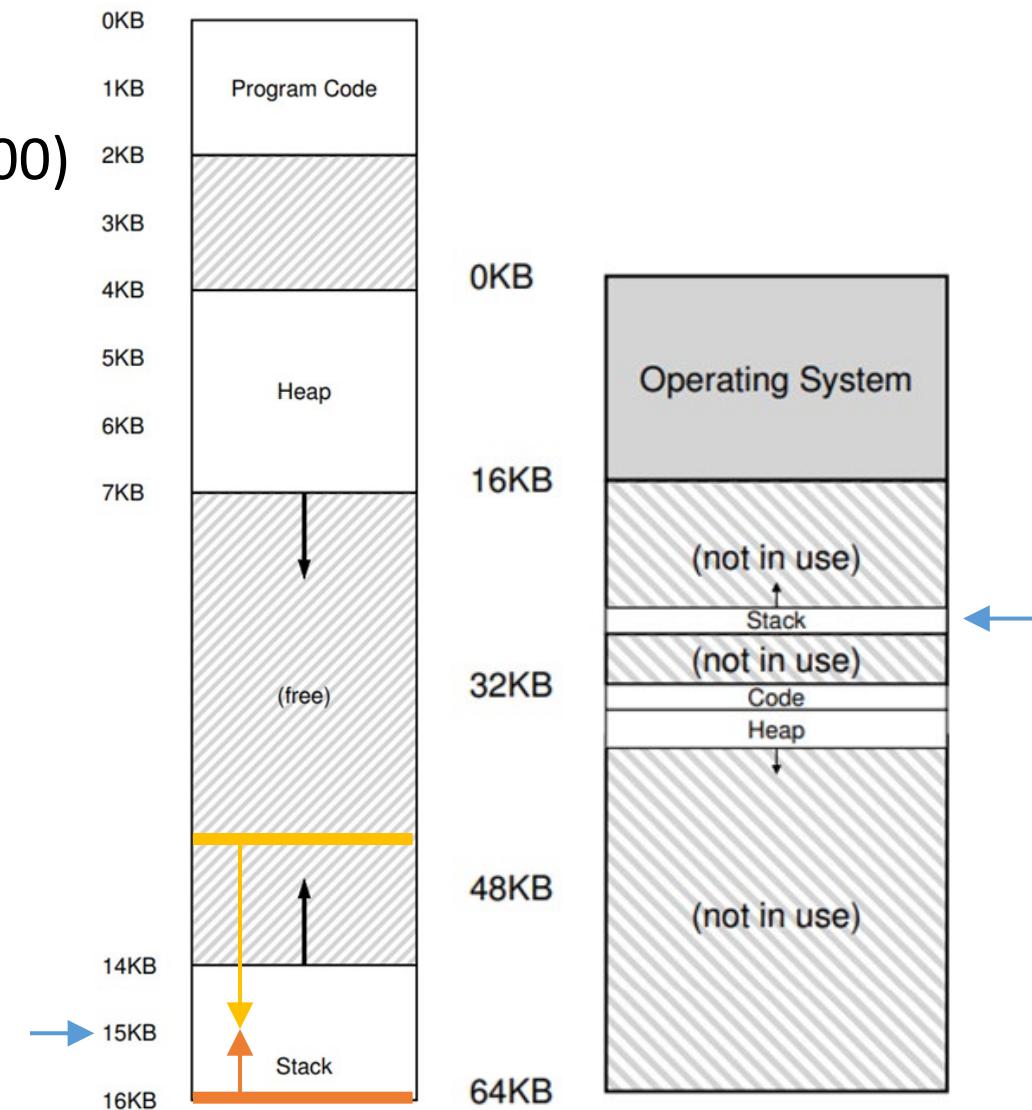
Segment	Base	Size (max 4 KB)	Grows Positive?
Code ₀₀	32 KB	2 KB	1
Heap ₀₁	34 KB	3 KB	1
Stack ₁₁	28 KB	2 KB	0

Stack Segment

- Example

- VA **15360** = 0b11 1100 0000 0000 (0x3C00)
- Segment = 11 = **Stack**
- **Offset** = 0xC00 = 3072
- **Negative offset** = $3072 - 4 \text{ KB} = -1 \text{ KB}$
- PA = Base[Stack] + Negative offset
 $= 28 \text{ KB} - 1 \text{ KB} = \text{27 KB}$
- Check: $|\text{Negative offset}| < \text{Size}[Stack]$

Segment	Base	Size (max 4 KB)	Grows Positive?
Code ₀₀	32 KB	2 KB	1
Heap ₀₁	34 KB	3 KB	1
Stack ₁₁	28 KB	2 KB	0



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	Base	Size (max 4 KB)	Grows Positive?	Protection
Code ₀₀	32 KB	2 KB	1	Read-Execute
Heap ₀₁	34 KB	3 KB	1	Read-Write
Stack ₁₁	28 KB	2 KB	0	Read-Write

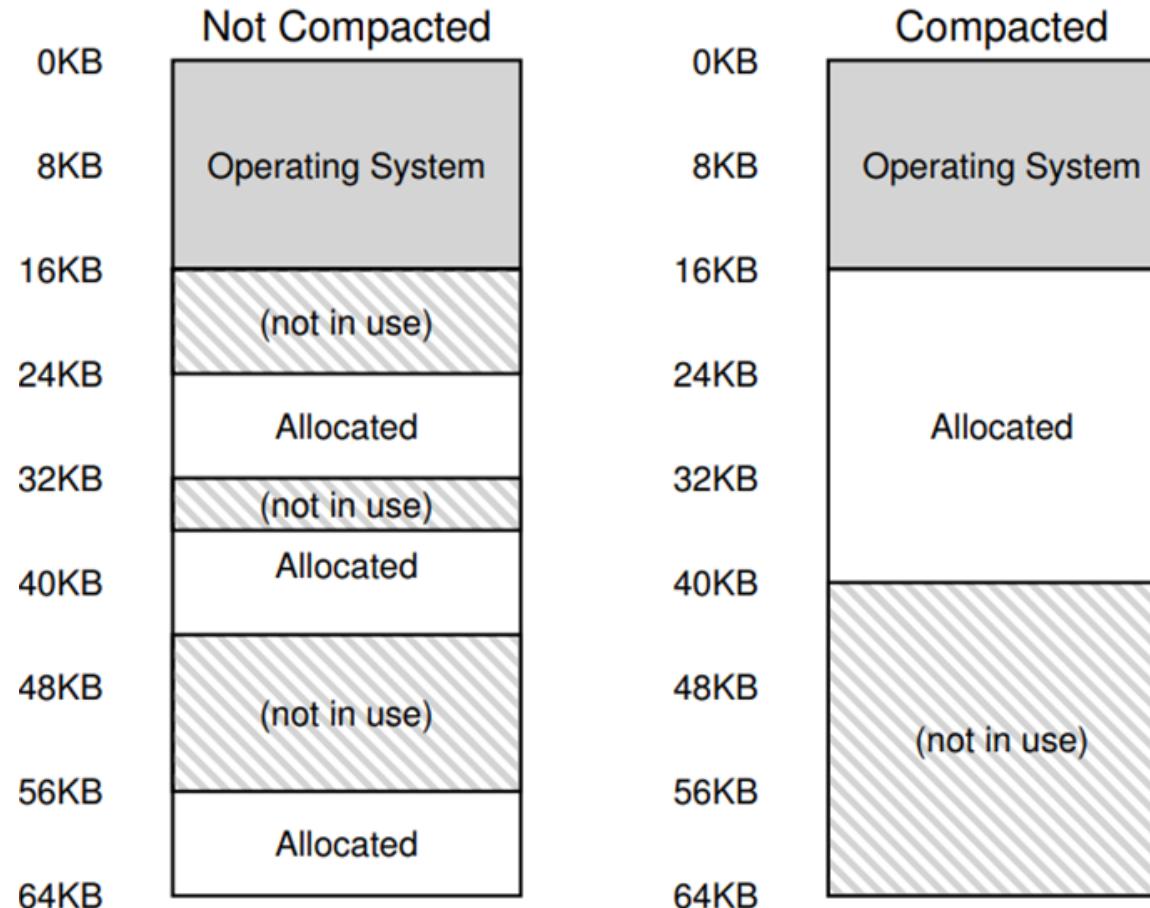
Fine-grained and Coarse-grained Segmentation

- **Coarse-grained** segmentation chops up the address space into relatively large, coarse chunks. Smaller number of segments.
 - E.g., code, heap, stack, etc.
- **Fine-grained** segmentation allows for address spaces consist of a large number of smaller segments.
 - To support many segments, hardware support with a **segment table** is required.

OS Support

- When **context switch** takes place
 - OS must **save** and **restore** the **segment registers** correctly between context switches.
- When segment **grows** (or shrinks)
 - When user process `malloc()`, and there are not enough space in heap, `sbrk()` system call will be invoked. OS will (usually) provide **more space** and update **segment size register**. OS could also reject the request.
- When **new address space** is created
 - OS has to find space in physical memory for its segments. Physical memory quickly becomes full of little **holes** of free space, making it difficult to allocate new segments (called **external fragmentation**).

External Fragmentation



Summary

- Segmentation can better support sparse address spaces.
- The overheads of translation are minimal. Extracting segment and offsets are easy and well-suited to hardware.
- If code is placed within a separate segment, code sharing can be done across multiple running programs.
- Issues
 - External fragmentation
 - Large sparse segments

COMP 4736

Introduction to Operating Systems

07. Memory Management III

(OSTEP Ch. 17 & 18)

Ch. 17. Free-Space Management

(OSTEP Ch. 17)

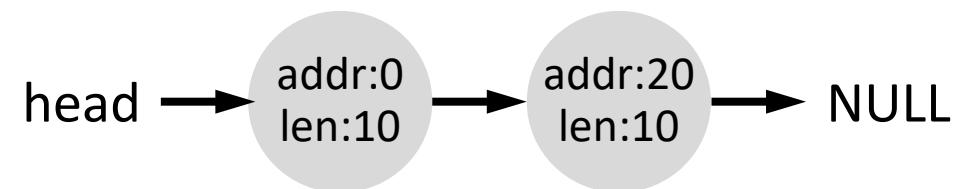
Free-Space Management

- Managing free space can be easy.
 - E.g., when space is divided into **fixed-sized** units.
 - Just keep a list and return the first entry when requested.
- It becomes more difficult when the free space consists of **variable-sized** units.
 - **External fragmentation**
 - 20 bytes of free space, but a request of 15 bytes will fail.

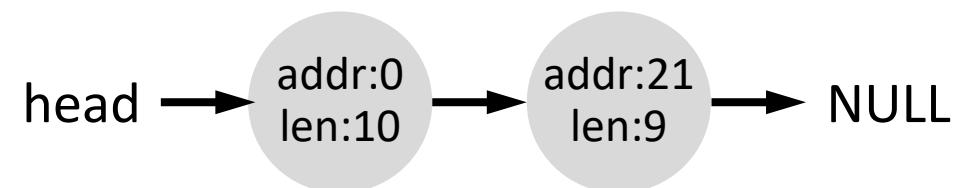


Splitting

- When the memory allocation request is smaller than the size of free chunks,
 - Find a **free** chunk of memory that can satisfy the request and splitting it into two: **allocated** and **free**

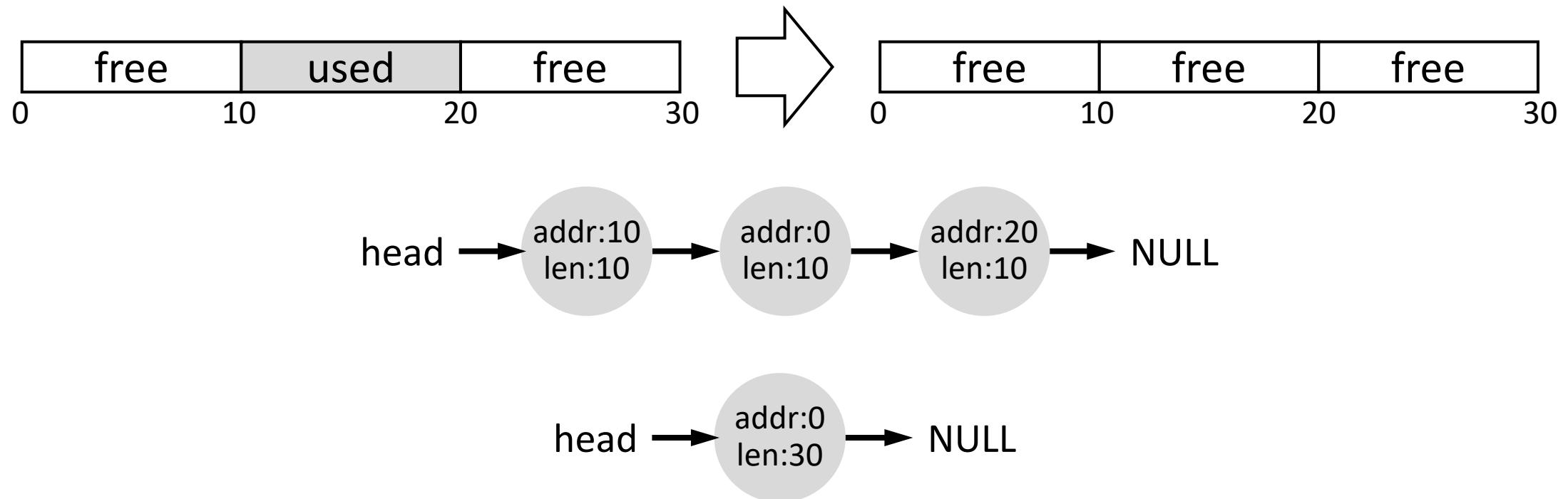


- 1 byte is requested



Coalescing

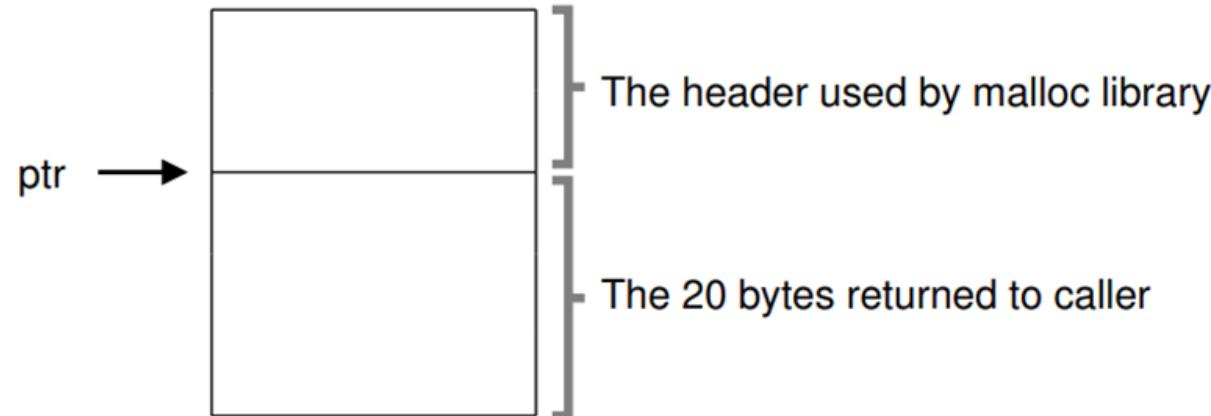
- **Coalescing**: Merge returning free chunk with existing chunks into a single large free chunk if they are right next to another.



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**?

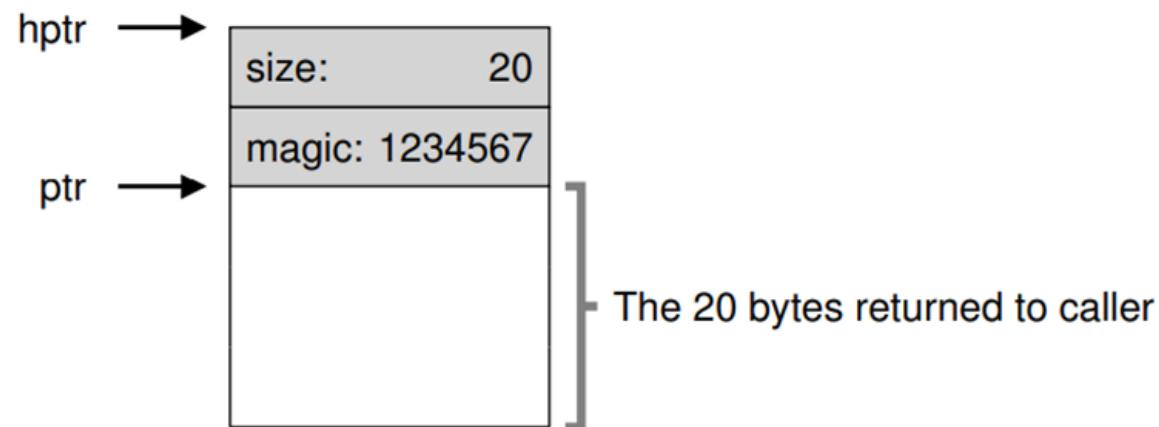
```
ptr = malloc(20);
```



Tracking the Size of Allocated Regions

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

- Actual **chunk size** of malloc(N) is N plus the size of header.



Tracking the Size of Allocated Regions

```
void free(void *ptr) {
    header_t *hptr = (void *)ptr - sizeof(header_t);
    ...
    assert(hptr->magic == 1234567);
    ...
}
```

- When N bytes of memory is requested, the library does not search for a free chunk of size N . It searches for a free chunk of size **N plus the size of the header**.

Embedding a Free List

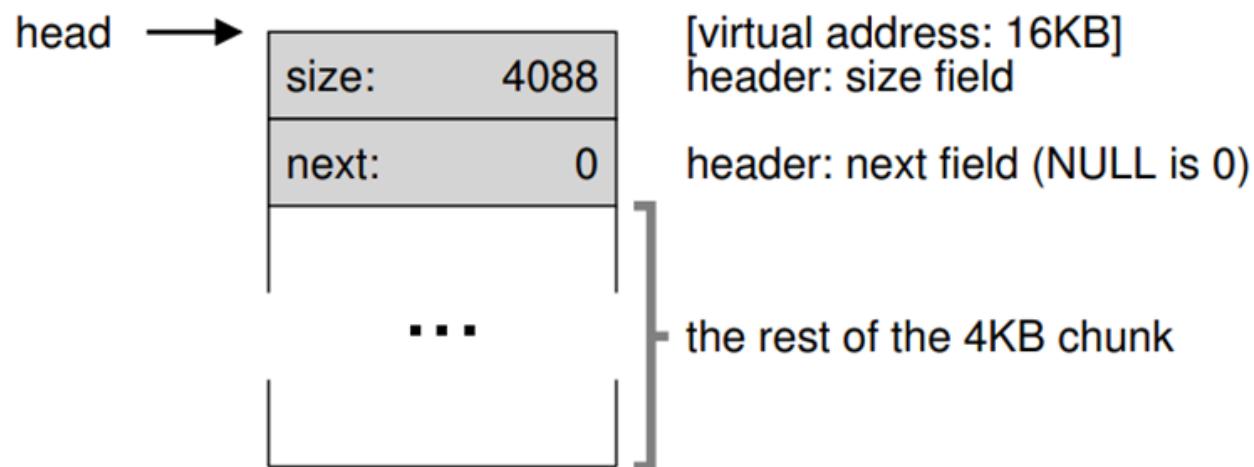
- In a typical list, when allocating a new node, `malloc()` is called.
- In memory-allocation library, the node is built **inside** the free space itself.

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

Embedding a Free List

```
// 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;
```

- Heap with one free chunk

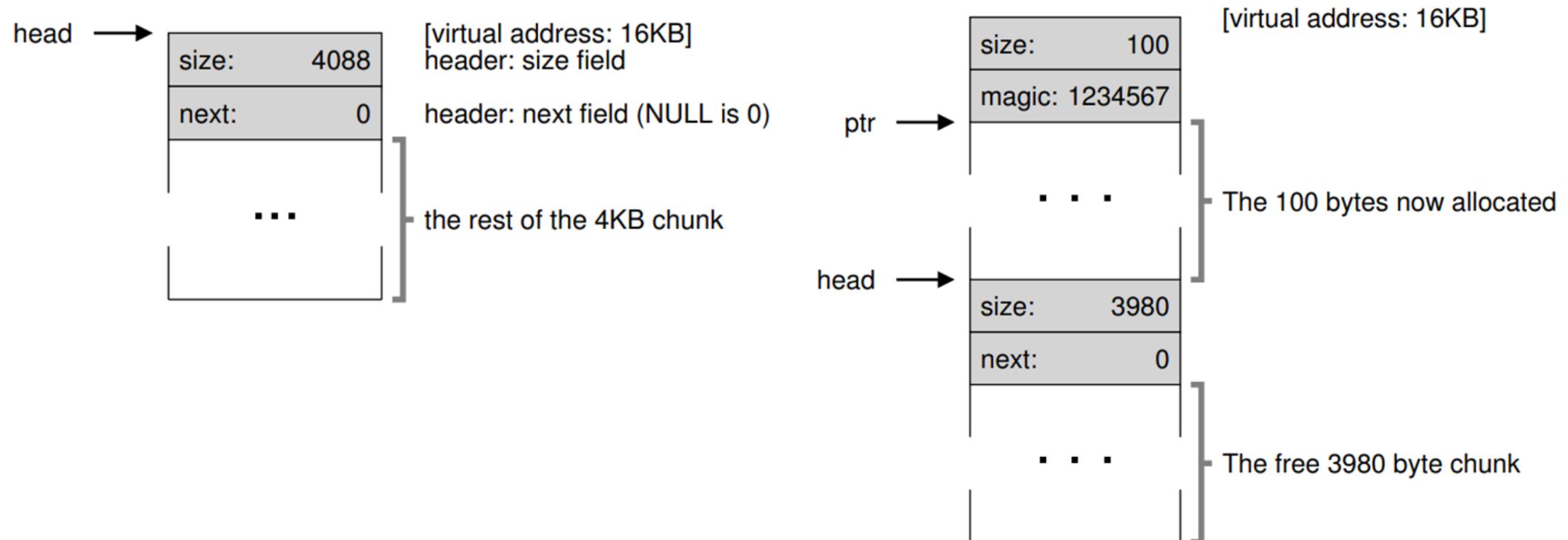


Embedding a Free List: Allocation

- 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.

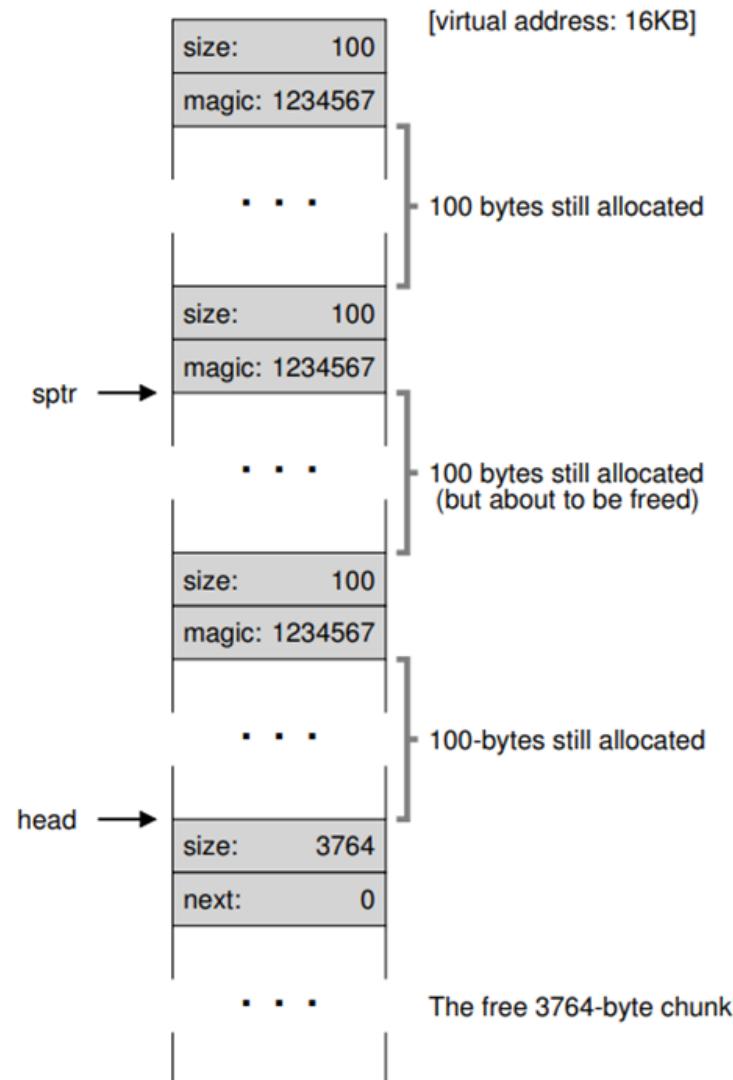
Embedding a Free List: Allocation

- Example: a request for 100 bytes by `ptr = malloc(100)`
→ 108 byte is allocated



Free Space With Three Chunks Allocated

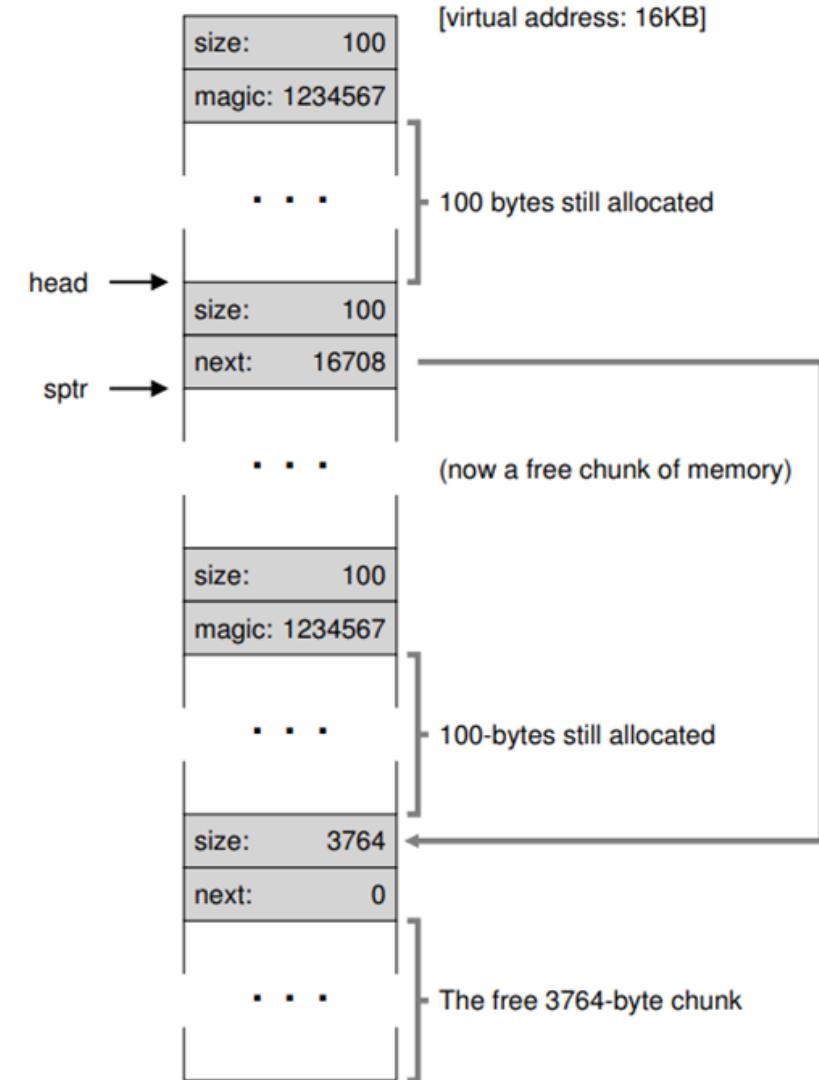
- After three 100-byte requests (or 108 including the header).



Free Space With free()

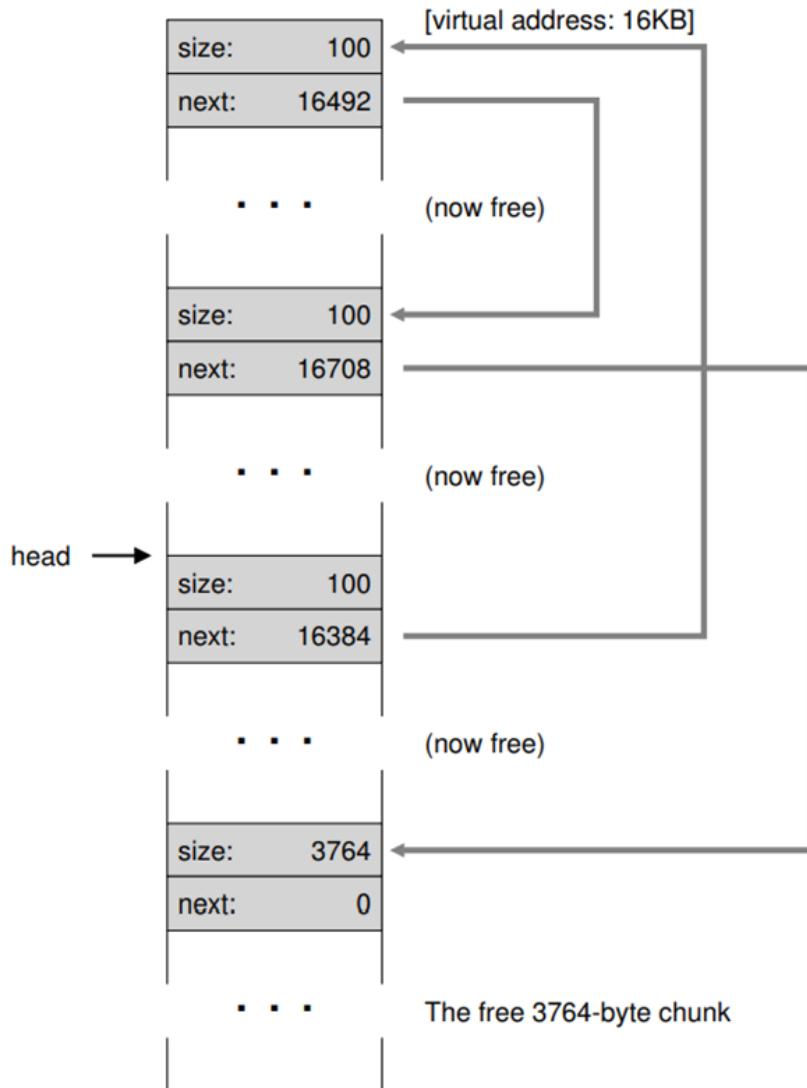
- The middle chunk is freed.
- Assume the free chunk is inserted back at the **head** of the **free list**.

```
free(sptr);
node_t *prev_head = head;
head = sptr;
head->size = 100;
head->next = prev_head;
```



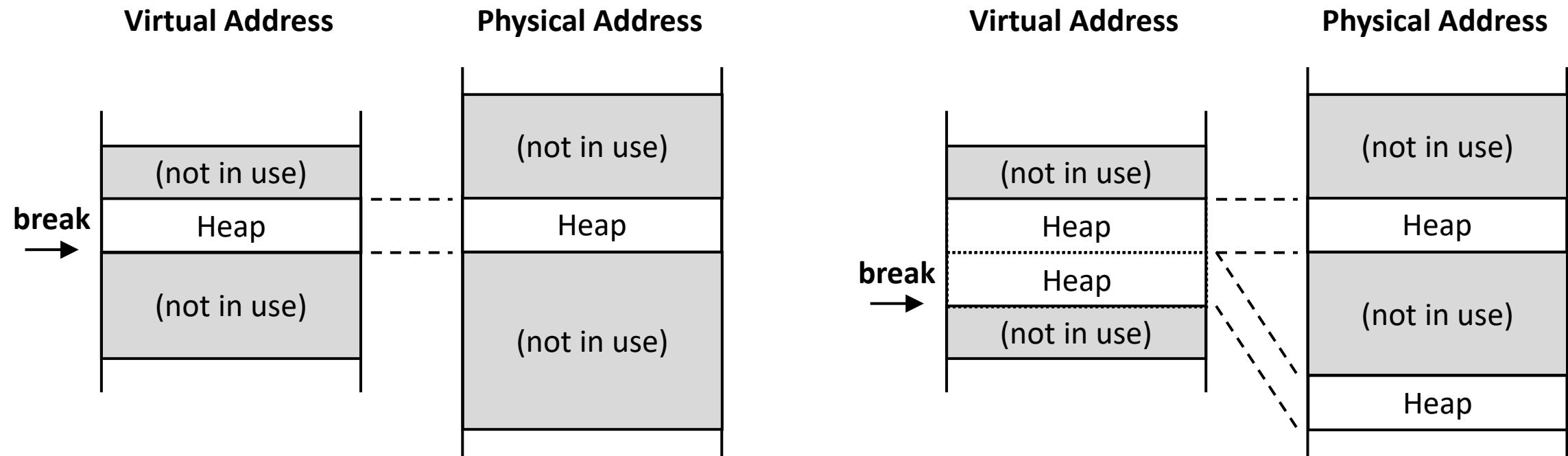
A Non-Coalesced Free List

- After the last two chunks are freed, without **coalescing**.



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 **as big** or **bigger** than the **requested** size.
- Returning the one that is the smallest in that group of candidates.
- Need to **search exhaustively** for the correct free block.

- **Worst Fit**

- Finding the **largest** chunk and returning the requested amount.
- Keeping the remaining chunk on the free list.
- Again, a **full search** is needed.
- Worse, it leads to **excess fragmentation**.

Managing Free Space: Basic Strategies

- **First Fit**
 - Finding the **first** chunk that is **big enough** for the request.
 - Returning the requested amount and keeping the remaining free space on the free list.
 - **No exhaustive search** is needed.
 - May **pollute** the **beginning** of the free list with small objects.
- **Next Fit**
 - Finding the first chunk that is big enough for the request, from the location within the list where **one was looking last**.
 - Similar performance to first fit.
 - Tries to spread the searches throughout the list **more uniformly**.

Examples of Basic Strategies

- The starting free list. An allocation of size 15 is requested.



- Best Fit**

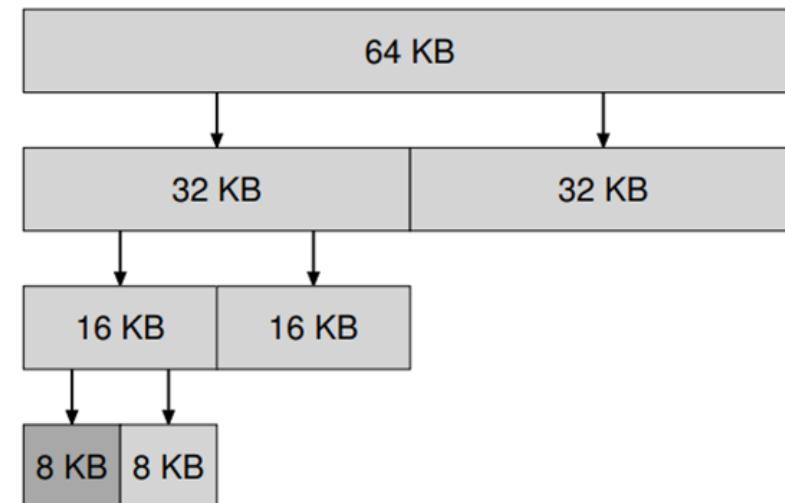


- Worst Fit**



Other Approaches

- **Segregated Lists**
 - Keep a specialized list to manage objects of popular size (e.g., locks, file-system inodes, etc.).
 - All other requests are forwarded to a more general list.
- **Buddy Allocation**
 - Designed to make coalescing simple.



Summary

- The most rudimentary forms of **memory allocators** have been discussed.
- Such allocators exist everywhere, linked into **every** C program you write, as well as in the **underlying OS** which is managing memory for its own data structures.
- As with many systems, there are many trade-offs.
- Making a **fast, space-efficient, scalable** allocator remains an on-going challenge.

Ch. 18. Paging: Introduction

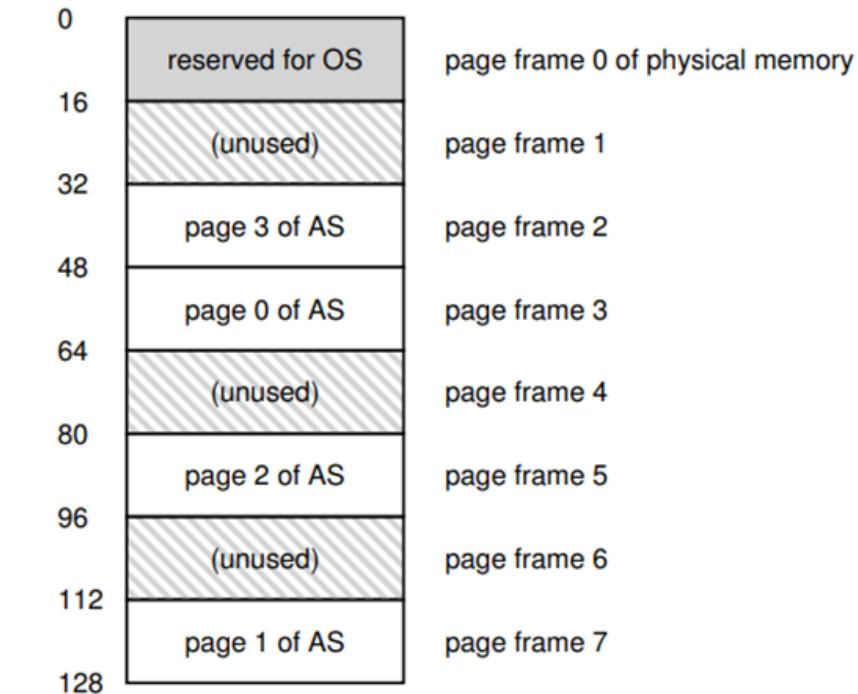
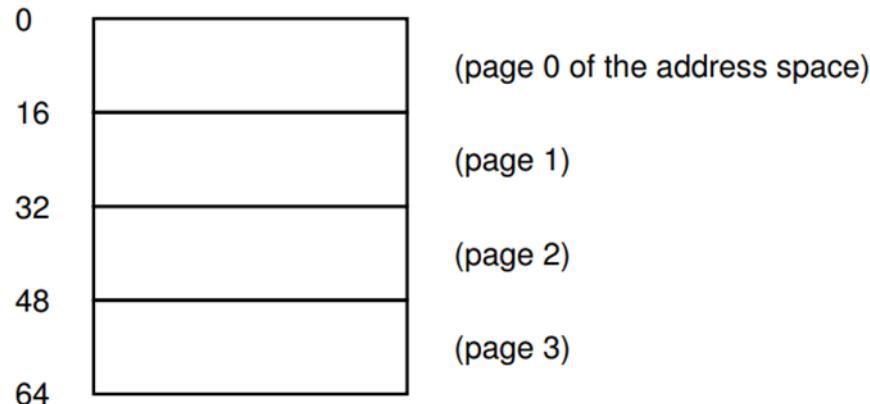
(OSTEP Ch. 18)

Concept of Paging

- **Segmentation** splits address space into **variable-sized** logical segments (e.g., code, stack, heap, etc.).
- **Paging** divides address space into **fixed-sized** unit called a **page**.
- With paging, **physical memory** can be viewed as array of fixed-sized slots called **page frames**.
- **Page table** per process is needed to translate the virtual address to physical address.

Simple Examples

- 64 bytes in **address space**, with four 16-byte **pages**.
- 128 bytes in **physical memory**, with eight 16-byte **page frames**.

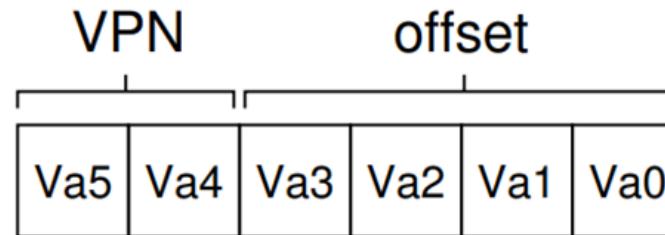


Advantages of Paging

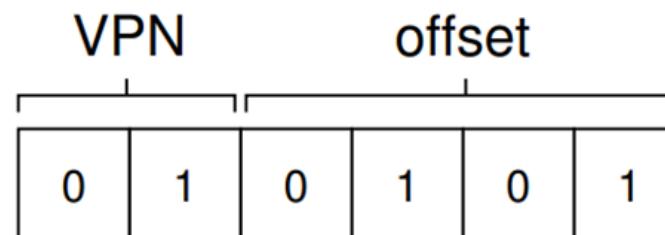
- **Flexibility**
 - Supports the abstraction of address space effectively.
 - E.g., no need to make assumption about the direction the heap and stack grow and how they are used.
- **Simplicity** of free-space management
 - The pages in address space and the page frames are of the same size.
 - Easy to allocate and keep a free list.

Address Translation

- Two components in the virtual address
 - **Virtual page number (VPN)**
 - **Offset** within the page

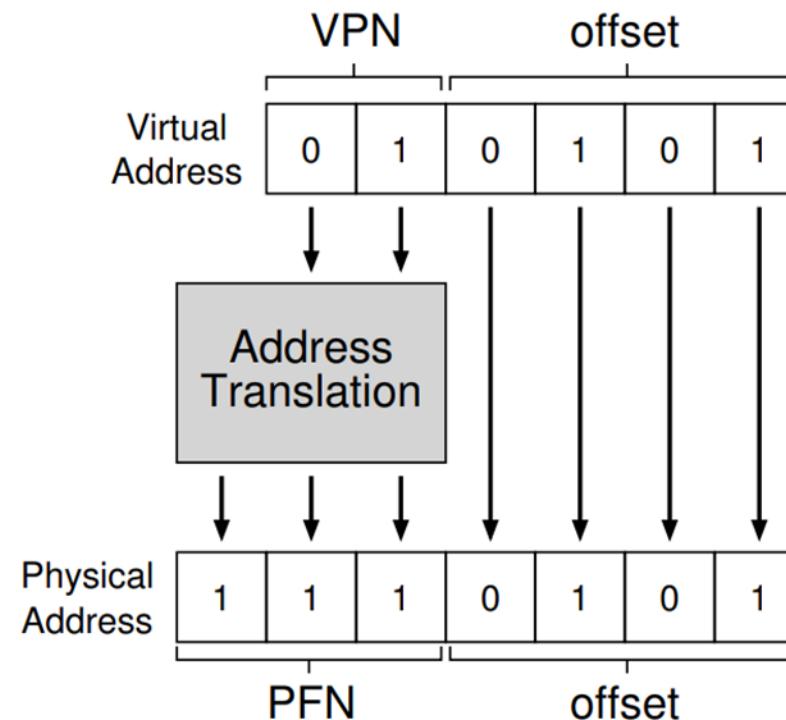


- Example: virtual address 21 in 64-byte address space



Address Translation

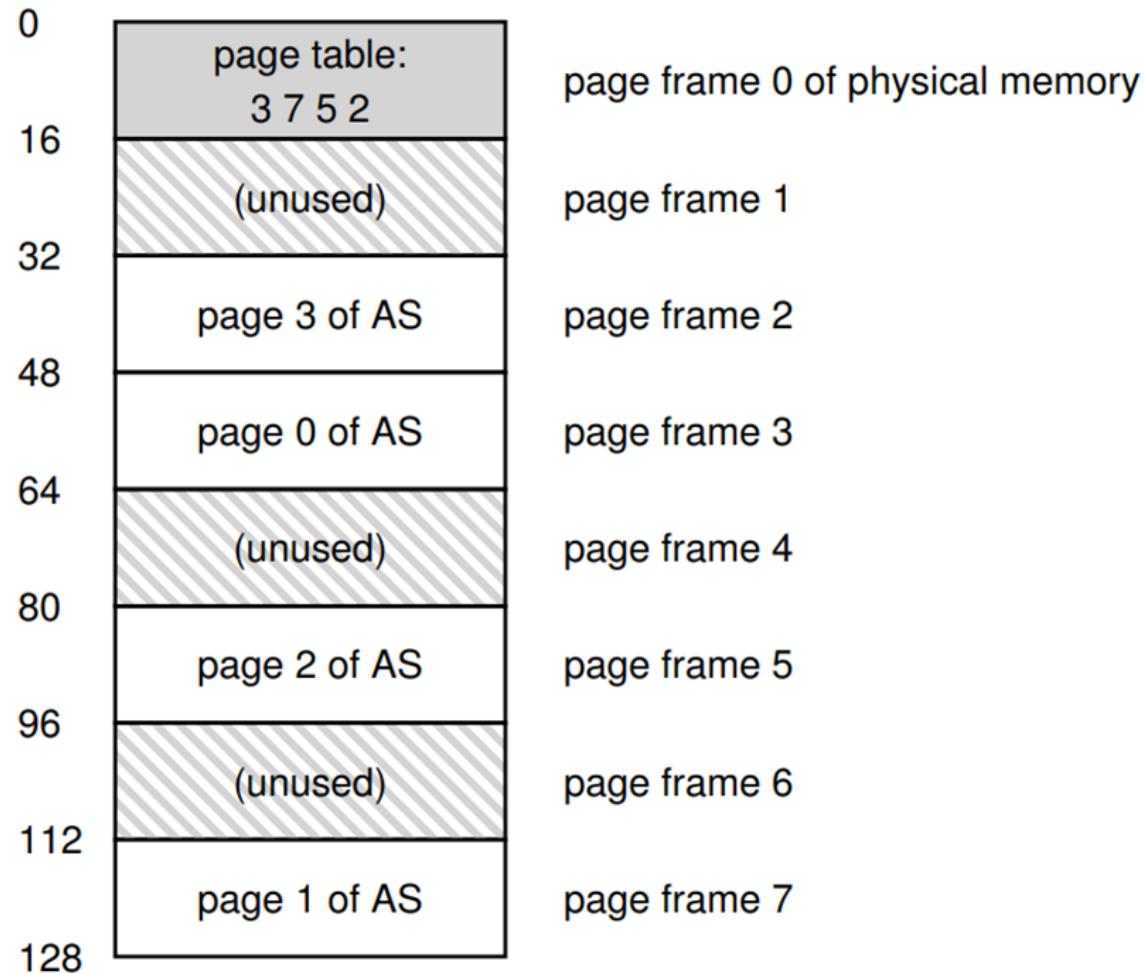
- In the **page table**, find which page frame VPN 1 is placed in.
- That is called **physical frame number (PFN)**



Where are Page Tables Stored?

- **Page tables** can get terribly large.
 - 32-bit address space with 4-KB pages
 - 20 bits for VPN, 12 bits for offset
 - $(2^{20} \text{ entries}) \times (4 \text{ B per page table entry (PTE)}) = 4 \text{ MB}$
 - That is 4 MB per process. 400 MB for 100 processes.
- Since page tables are so large, we cannot keep them in special on-chip hardware in the MMU.
- The page table for each process is kept in **memory**.

Page Table in Kernel Physical Memory



What is Actually in the Page Table?

- The page table is a data structure that is used to map the **virtual address** (VPN) to **physical address** (PFN).
- The simplest form is called a **linear page table**, which is just an array.
- The OS **indexes** the array by VPN, and looks up the PTE at that index, which will return the PFN.

Common Bits of Page Table Entry

- **Valid Bit**: Indicating whether the particular translation is valid.
- **Protection Bits**: 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.

An x86 Page Table Entry (PTE)



- **P**: Present bit
- **R/W**: Read/Write bit
- **U/S**: User/Supervisor bit
- **A**: Accessed bit
- **D**: Dirty bit
- **PFN**: 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

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

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

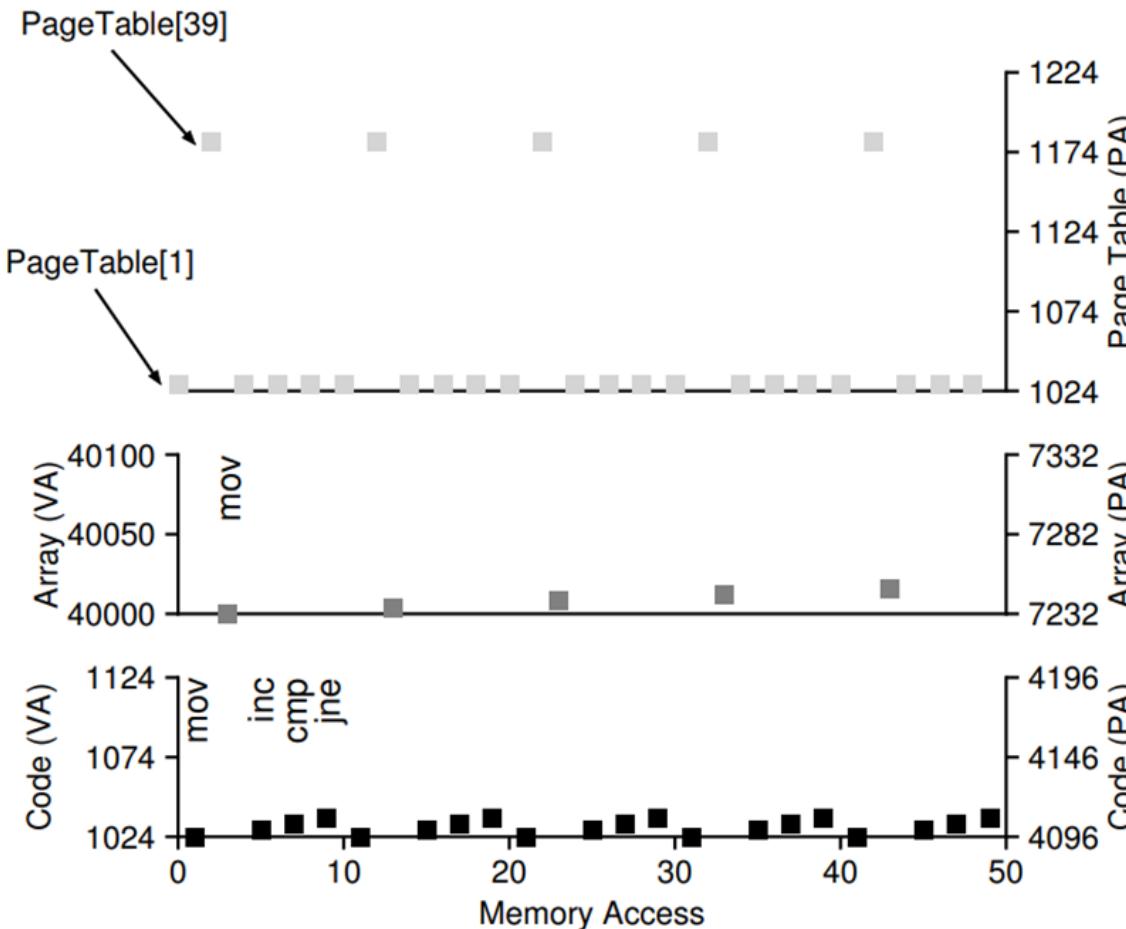
```
1024 movl $0x0,(%edi,%eax,4)          // [%edi + %eax * 4] = 0
1028 incl %eax
1032 cmpl $0x03e8,%eax                // 0x03e8 equals 1000
1036 jne  1024
```

A Virtual (and Physical) Memory Trace

- Assume
 - 64-KB virtual address space
 - 1-KB page
 - VPN 1 → PFN 4
 - VPN 39, 40, 41, 42 → PFN 7, 8, 9, 10
- Code VA 1024 resides on the second page (i.e., VPN 1).
- Array has VA 40000–44000 (i.e., VPN 39–42).

A Virtual (and Physical) Memory Trace

- 10 memory accesses per loop iteration.



Summary

- The concept of **paging** is introduced as a solution to our challenge of virtualizing memory.
- It has many advantages, such as **no external fragmentation** (paging divides memory into fixed-sized units by design) and **flexibility** (enabling the sparse use of virtual address spaces).
- However, implementing paging can lead to **slower machine** (with many extra memory accesses to access the page table) and **memory waste** (with memory filled with page tables instead of useful application data).

COMP 4736

Introduction to Operating Systems

08. Memory Management IV

(OSTEP Ch. 19 & 20)

Ch. 19. Paging: Faster Translations (TLBs)

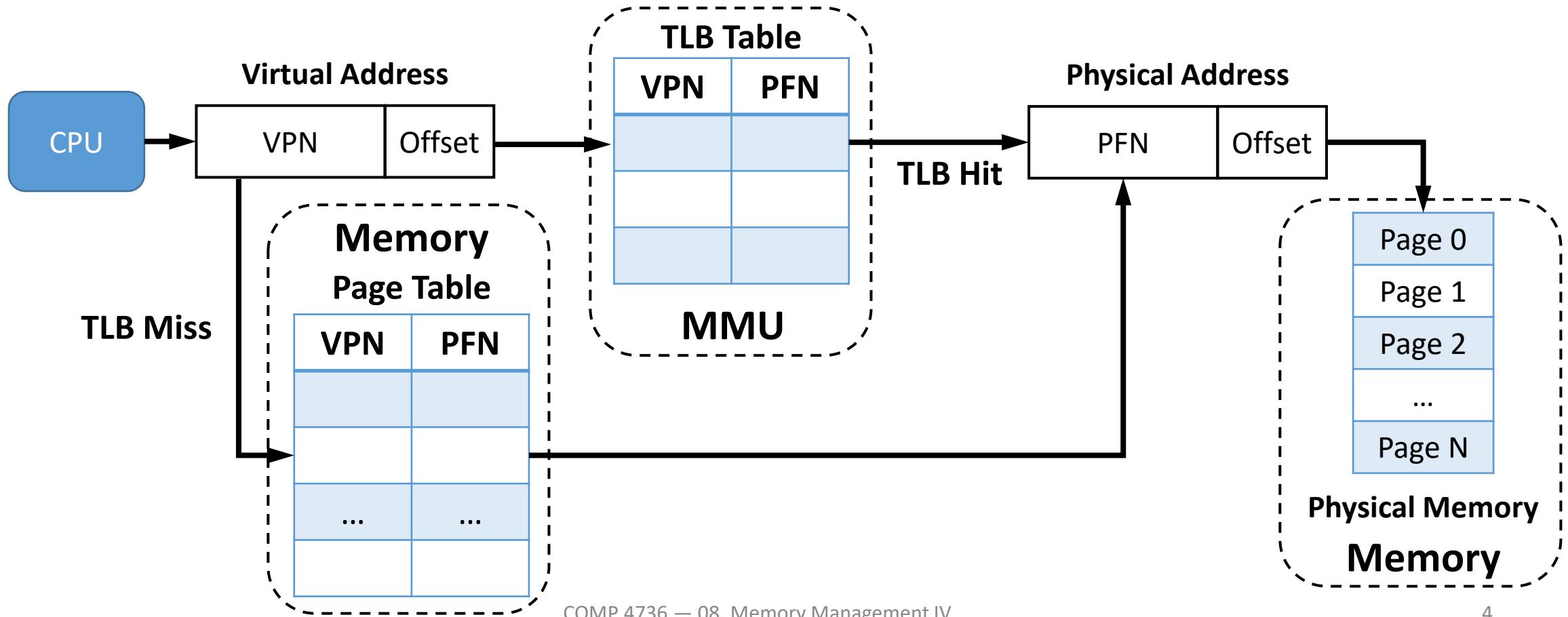
(OSTEP Ch. 19)

Issues

- **Paging** maps **virtual** page number (VPN) to **physical** frame number (PFN).
- It requires a large amount of mapping information, and is generally stored in physical memory.
- That means, before **every** instruction fetch or explicit load or store, we need to go to the **memory first** to translate the address.
- How can we speed up address translation?

Translation-Lookaside Buffer (TLB)

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



TLB Control Flow Algorithm

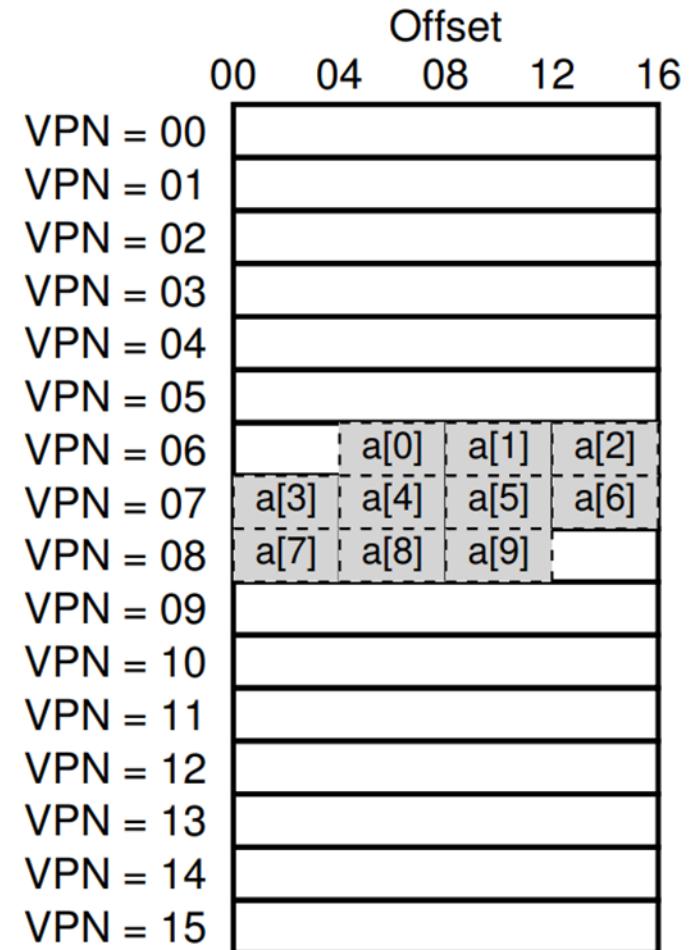
```
1      VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2      (Success, TlbEntry) = TLB_Lookup(VPN)
3      if (Success == True)    // TLB Hit
4          if (CanAccess(TlbEntry.ProtectBits) == True)
5              Offset = VirtualAddress & OFFSET_MASK
6              PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7              Register = AccessMemory(PhysAddr)
8          else
9              RaiseException(PROTECTION_FAULT)
10     else                      // TLB Miss
11         PTEAddr = PTBR + (VPN * sizeof(PTE))
12         PTE = AccessMemory(PTEAddr)
13         if (PTE.Valid == False)
14             RaiseException(SEGMENTATION_FAULT)
15         else if (CanAccess(PTE.ProtectBits) == False)
16             RaiseException(PROTECTION_FAULT)
17         else
18             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19             RetryInstruction()
```

Example: Accessing an Array

```
int i, sum = 0;  
for (i = 0; i < 10; i++) {  
    sum += a[i];  
}
```

- 3 TLB miss, 7 TLB hits.
- TLB **hit rate** = 70%

TLB improves performance due to **spatial locality**.



Locality

- **Temporal Locality**
 - An instruction or data item that has been **recently accessed** will likely be **re-accessed** soon in the future.
 - Think of loop variables or instructions in a loop.
- **Spatial Locality**
 - If a program accesses memory at **address x**, it will likely soon access memory **near x**.
 - Imagine streaming through an array of some kind.

Who Handles the TLB Miss?

- Hardware handles the TLB miss entirely on **CISC**.
 - The hardware has to know exactly **where** the page tables are located in memory, as well as their **exact format**.
 - 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** in Intel x86
- **RISC** has 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)
```

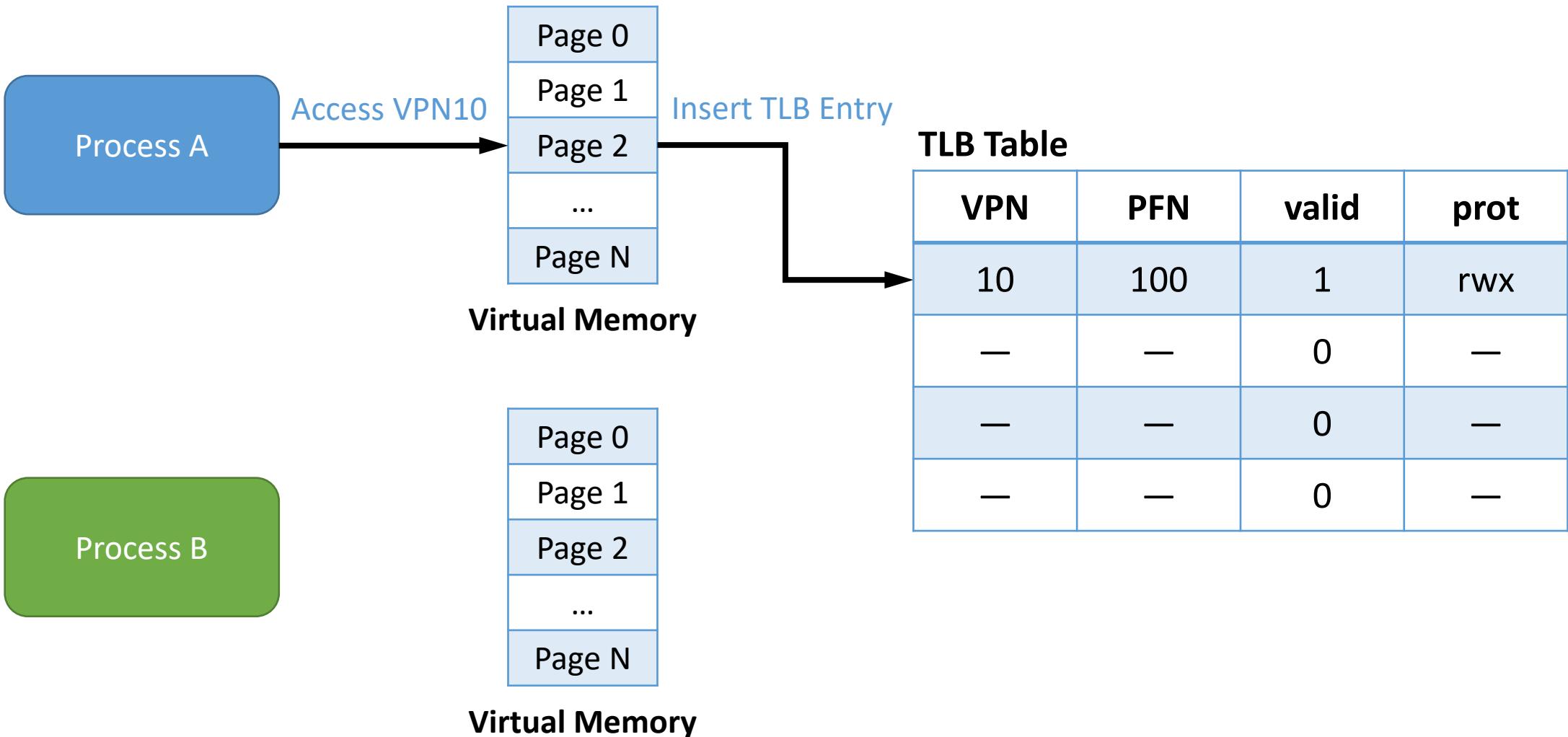
TLB Entry

- A typical TLB has 32, 64, or 128 entries, and is **fully associative**.
- Hardware searches the entire TLB in parallel to find the desired translation.

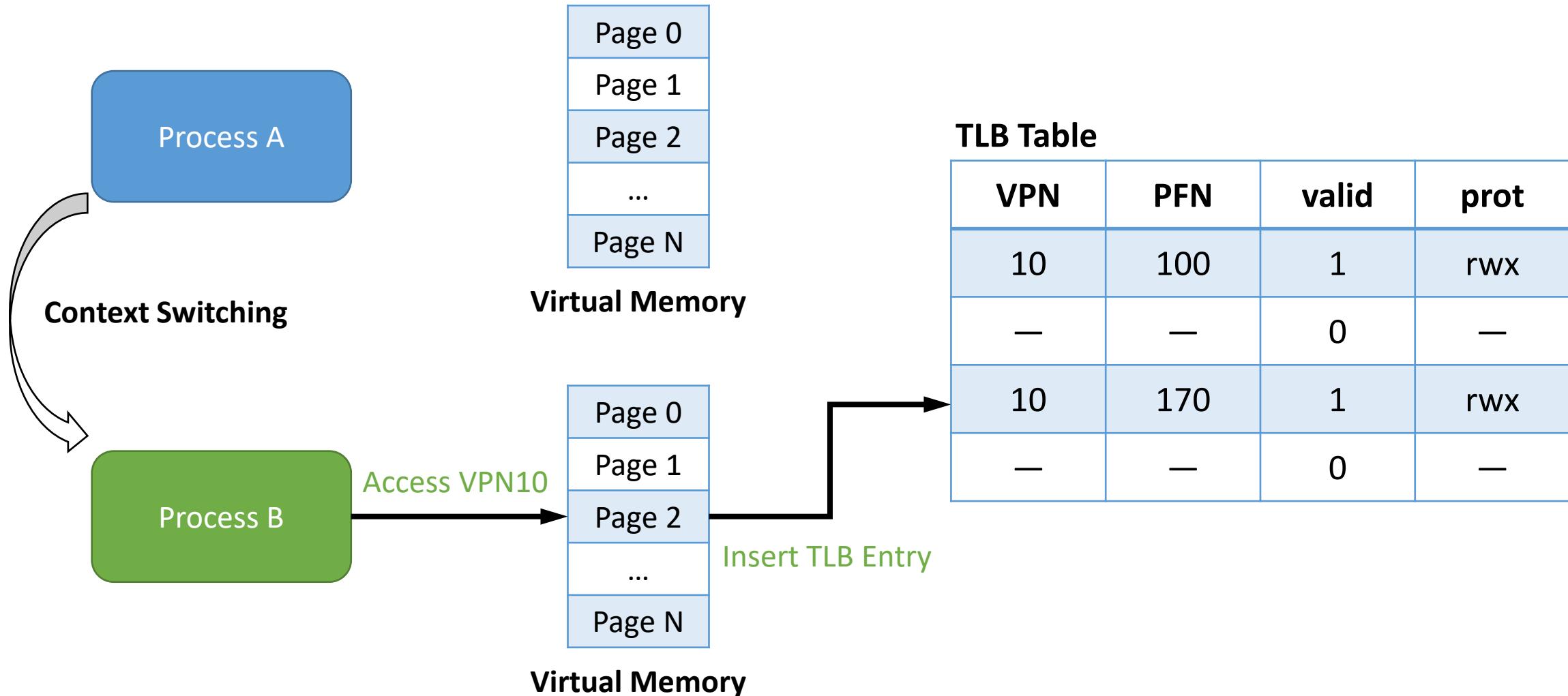


- Other bits include: **valid** bit, **protection** bits, **address-space identifier**, **dirty bit**, etc.

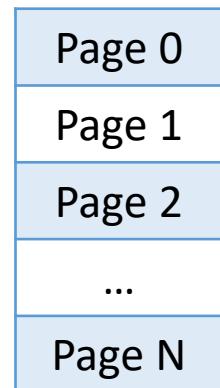
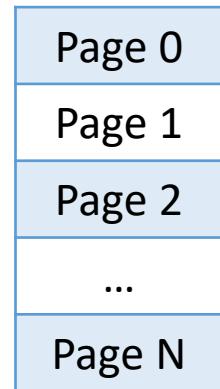
TLB Issue: Context Switches



TLB Issue: Context Switches



TLB Issue: Context Switches



TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

Can't distinguish which entry is meant for which process.

Address Space Identifier (ASID)

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

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

Another Case

- Two **different processes** with two **different VPNs** point to the **same physical page**.

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

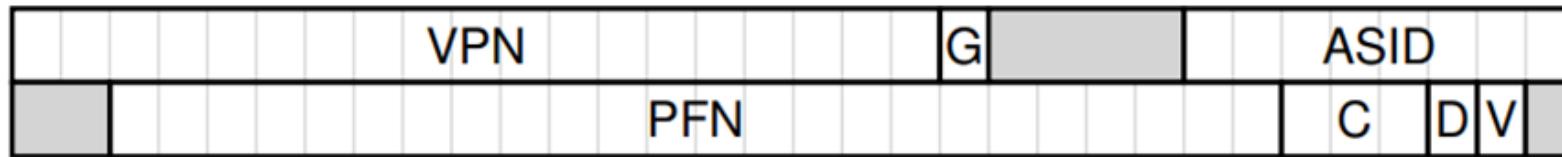
- It might arise when two processes **share** a page.
 - Process 1 is sharing physical page 101 with Process 2.
 - P1 maps this page into the 10th page of its address space.
 - P2 maps it to the 50th page of its address space.

TLB Replacement Policy

- One common approach is to evict the **least-recently-used (LRU)** entry.
 - Take advantage of **locality** in the memory-reference stream.
- Another typical approach is to use a **random** policy.
 - Useful due to its **simplicity** and ability to avoid corner-case behaviors.

A Real TLB Entry

- MIPS R4000 supports 32-bit address space with 4-KB pages.



Bits	Content
19-bit VPN	User addresses will only come from half the 20-bit address space.
24-bit PFN	System can support 64 GB of physical main memory (2^{24} 4-KB pages).
Global bit (G)	Used for pages that are globally-shared among processes.
8-bit ASID	Used by OS to distinguish between address spaces.
Coherence bits (C)	Determines how a page is cached by the hardware.
Dirty bit (D)	Is marked when the page has been written to.
Valid bit (V)	Tells the hardware if there is a valid translation present in the entry.

Summary

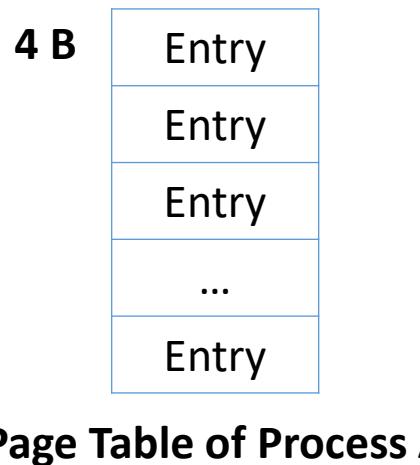
- Translation-Lookaside Buffer (TLB), can make **address translation faster** by working as a hardware **cache**.
- For **TLB hits**, memory reference can be handled **without** having to access the **page table** in the **main memory**.
- Basic organization and common issues of TLB are discussed.

Ch. 20. Paging: Smaller Tables

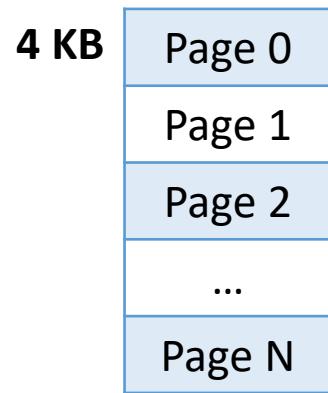
(OSTEP Ch. 20)

Paging: Linear Tables

- We usually have one page table for **every process** in the system.
- Assume a 32-bit address space (2^{32} bytes) with 4-KB (2^{12} bytes) pages and a 4-byte page-table entry.



Page Table of Process A



Physical Memory

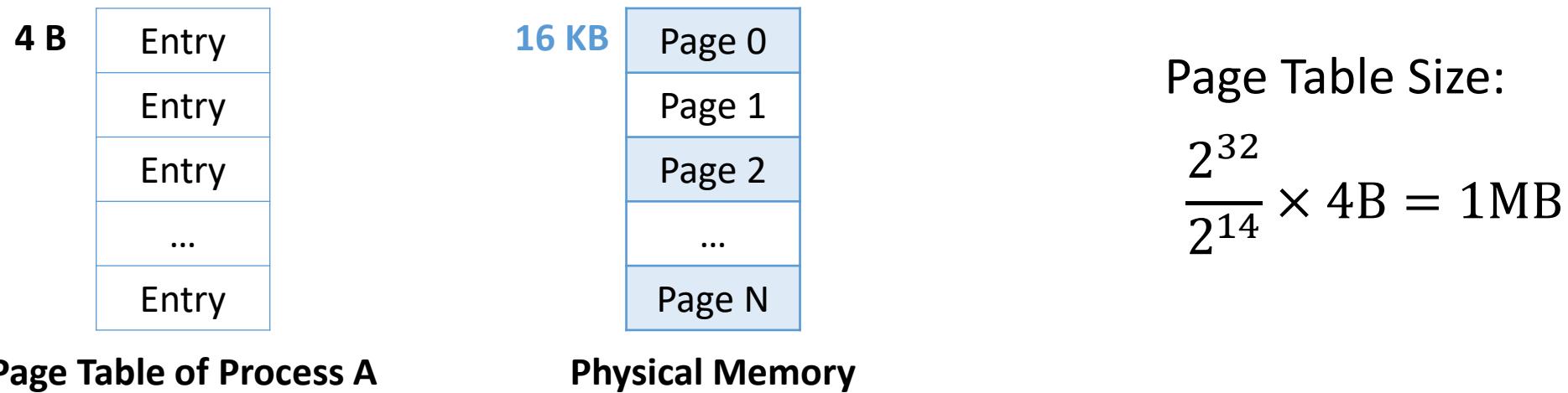
Page Table Size:

$$\frac{2^{32}}{2^{12}} \times 4\text{B} = 4\text{MB}$$

Simple array-based page tables are too big,
taking up far too much memory.

Paging: Bigger Pages

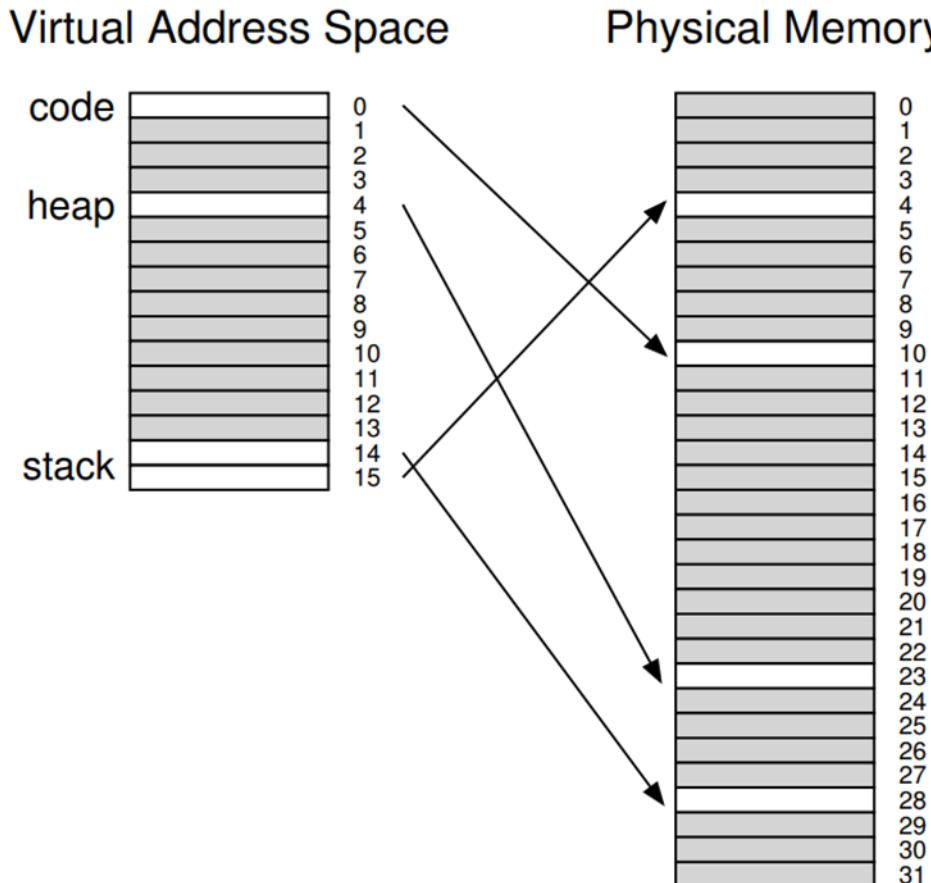
- We usually have one page table for **every process** in the system.
- Assume a 32-bit address space (2^{32} bytes) with **16-KB** (2^{14} bytes) pages and a 4-byte page-table entry.



However, big pages lead to waste within each page, also known as **internal fragmentation**.

Unused Page Table

- A 16-KB address space with 1-KB pages.

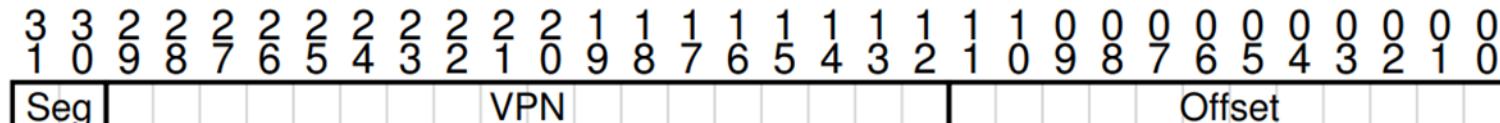


PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Most of the page table is unused!

Hybrid Approach: Paging and Segments

- To have one page table for **each segment**.
 - **Base register** contains the **physical address** of a **linear page table** for that segment.
 - **Bound register** is used to indicate the end of the page table.
- Example: Each process has **three** page tables associated with it.



Seg	Segment
00	Unused
01	Code
10	Heap
11	Stack

TLB Miss on Hybrid Approach

- Assuming a hardware-managed TLB.
- 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) .

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

Hybrid Approach

- The critical difference in the hybrid scheme is the presence of a **bounds register** per **segment**.
 - Example: if code segment is using three pages, its bounds register is set to 3.
 - Significant memory is saved compared to linear page table; **unallocated pages** between stack and heap **no longer** takes up space in the **page table**.
- 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.
 - It can also cause **external fragmentation** to arise again.

Multi-Level Page Tables

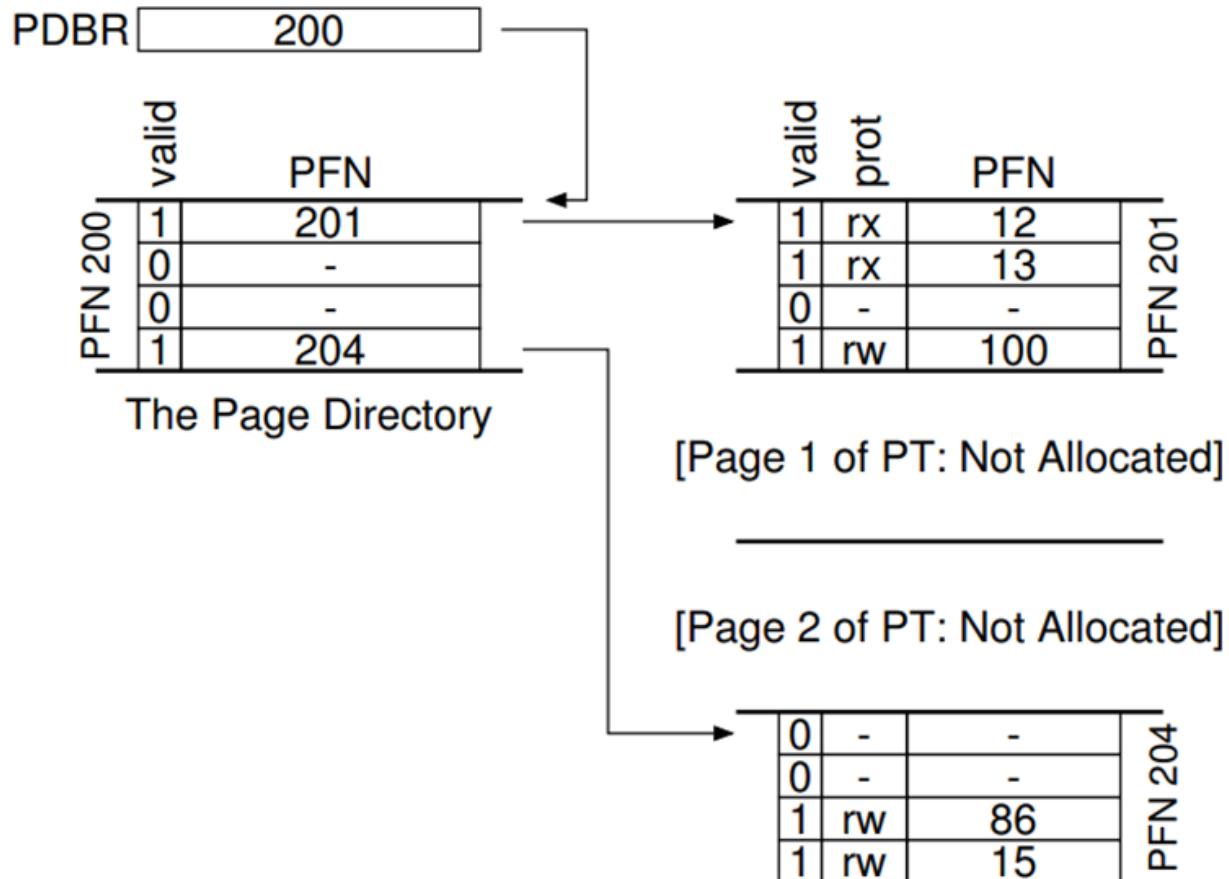
- Turn the linear page table into something like a tree.
 - Chop up the page table into page-sized units.
 - If an **entire page** of page-table entries is invalid, **don't allocate** that page of the page table at all.
 - To track whether a page of the page table is valid, use a new structure, called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

Multi-Level Page Tables: Page directory

Linear Page Table

PTBR			201
valid	prot	PFN	
1	rx	12	PFN 201
1	rx	13	
0	-	-	
1	rw	100	PFN 202
0	-	-	
0	-	-	
0	-	-	
0	-	-	
0	-	-	
0	-	-	
0	-	-	
0	-	-	
1	rw	86	PFN 203
1	rw	15	PFN 204

Multi-level Page Table

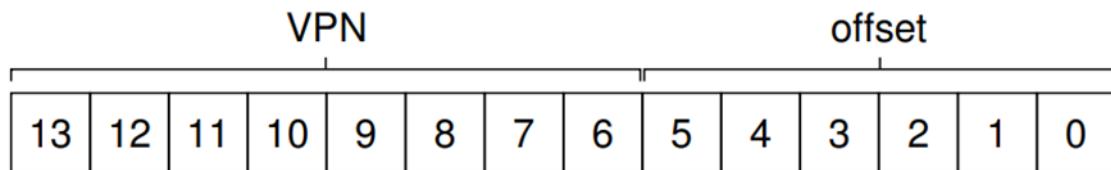


Multi-Level Page Tables

- **Page Directory**
 - 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)**.
- **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**.

Example: Linear Page Table

Specifications	Details
Address Space	$16\text{ KB} = 2^{14}\text{ bytes}$
Page Size	$64\text{ bytes} = 2^6\text{ bytes}$
Virtual Address	14-bit
VPN	8-bit
Offset	6-bit
Page Table Entry (PTE) Size	4 bytes



0000 0000	code
0000 0001	code
0000 0010	(free)
0000 0011	(free)
0000 0100	heap
0000 0101	heap
0000 0110	(free)
0000 0111	(free)
.....	... all free ...
1111 1100	(free)
1111 1101	(free)
1111 1110	stack
1111 1111	stack

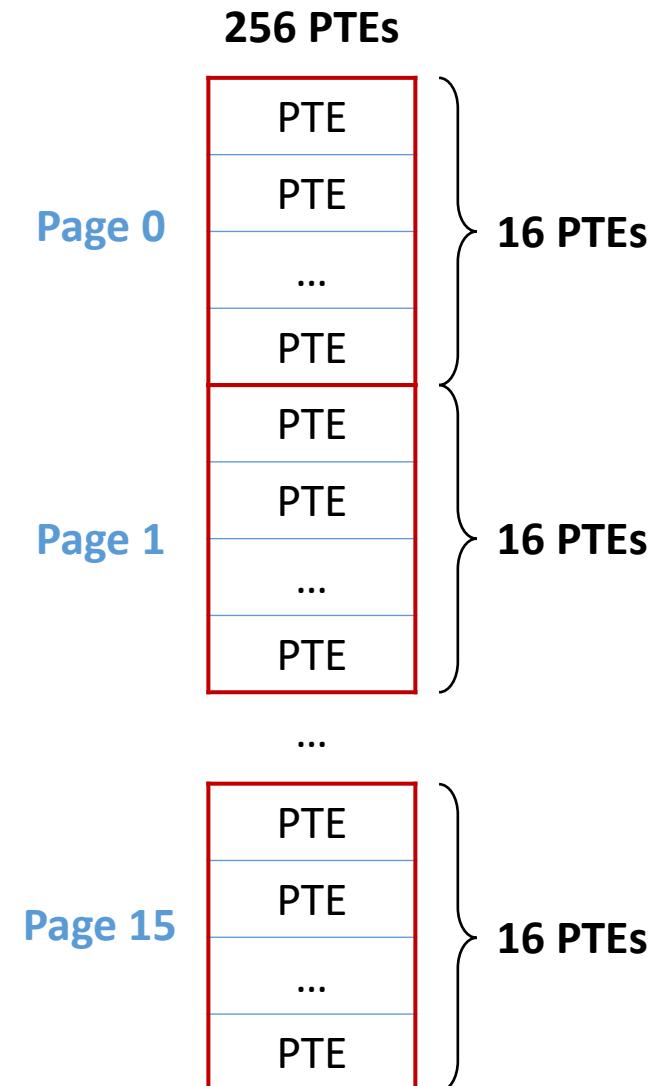
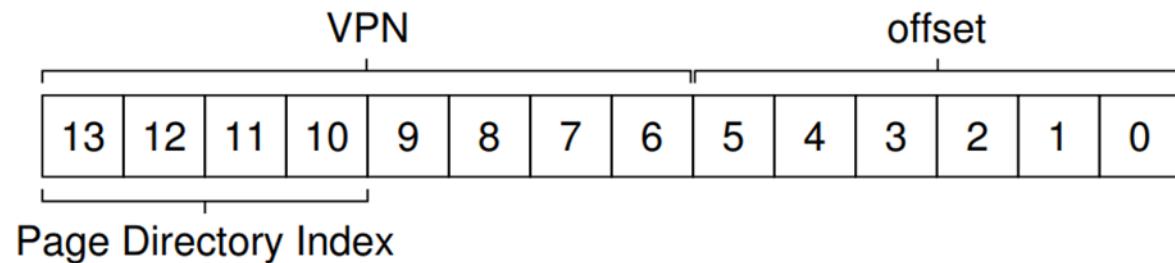
- **Linear Page Table:**

- $2^8 = 256$ PTEs
- Page table size = $256 \times 4\text{B} = 1\text{ KB}$

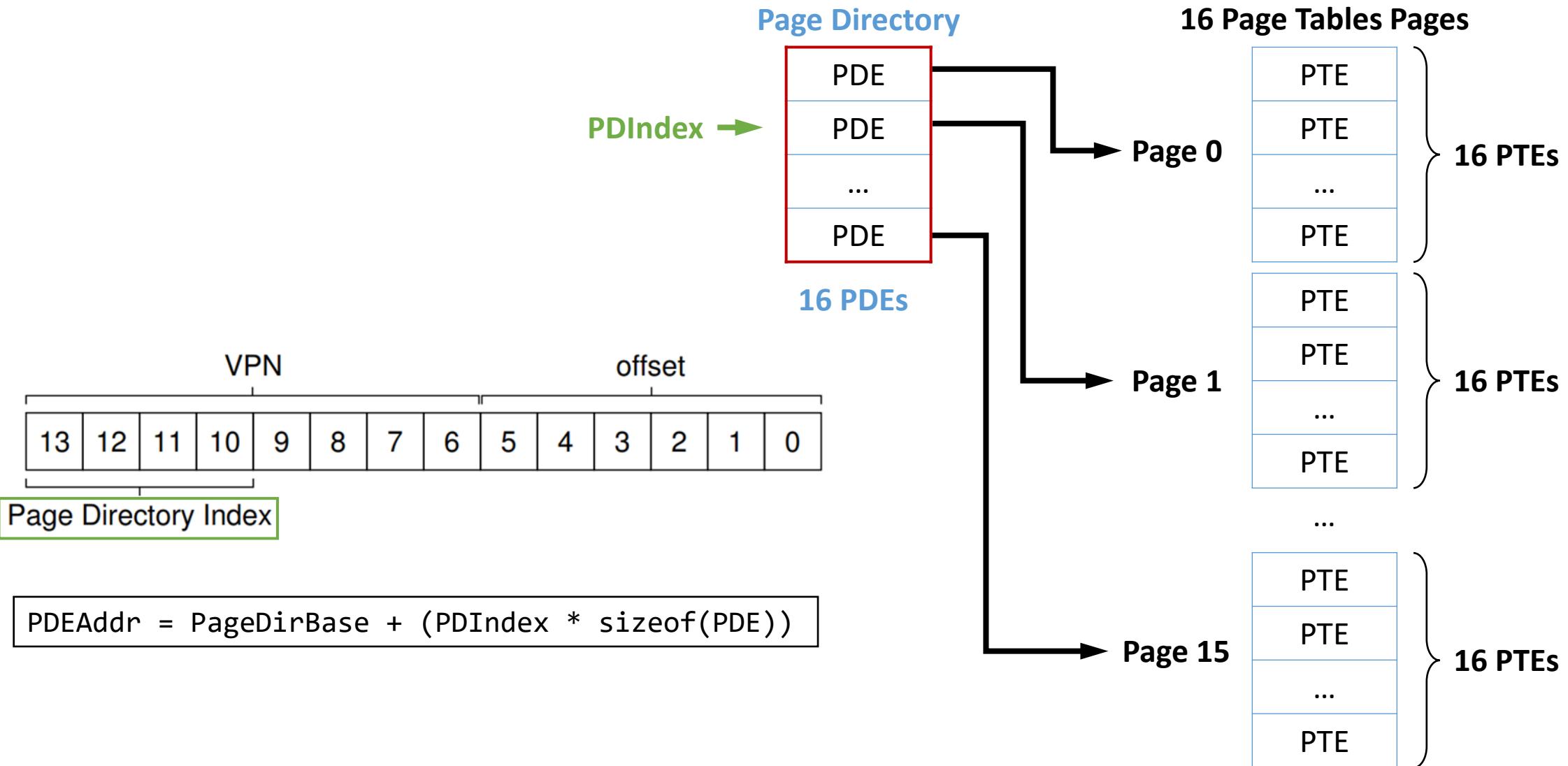
Example: Two-Level Page Table

- **Two-Level Page Table:**

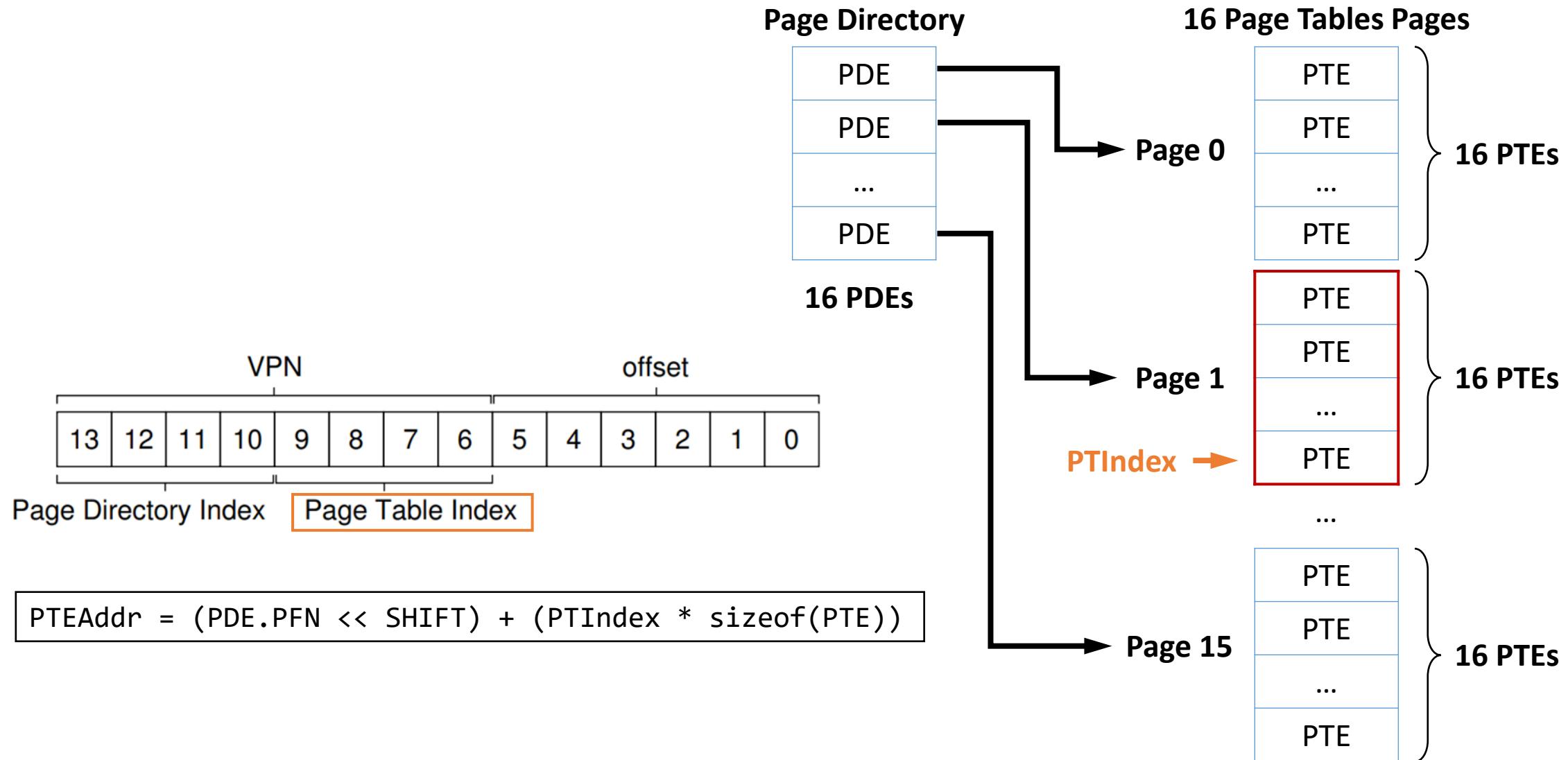
- $2^8 = 256$ PTEs
- Page table size = $256 \times 4B = 1$ KB
- Given that we have 64-byte pages, the page table can fit in $1\text{ KB}/64 = 16$ pages.
- Each page can hold $64/4 = 16$ PTEs.



Example: Two-Level Page Table



Example: Two-Level Page Table



Example: Two-Level Page Table

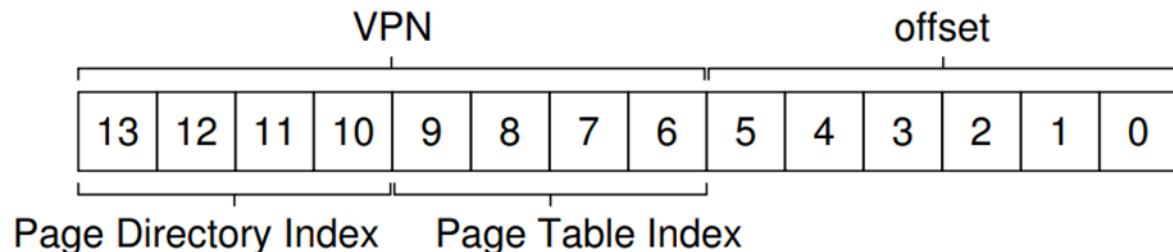
Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)			
PFN	valid?	PFN	valid	prot	PFN	valid	prot	
100	1	10	1	r-x	—	0	—	0000 0000 code PFN: 10
—	0	23	1	r-x	—	0	—	0000 0001 code PFN: 23
—	0	—	0	—	—	0	—	0000 0010 (free)
—	0	—	0	—	—	0	—	0000 0011 (free)
—	0	80	1	rw-	—	0	—	0000 0100 heap PFN: 80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

- Linear Page Table: 16 pages
- Two-Level Page Table: 3 pages

Example: Two-Level Page Table

- Translate VA **0x3F80** (0b`11 1111 1000 0000`) to PA.
- VPN = 0b11111110 = 254
- PDIndex (top 4 bits) = 0b1111 (Page 15 of the Page Table)
→ PFN 101
- PTIndex (next 4 bits) = 0b1110 (PTE 14)
→ PFN 55 (0x37 or 0b`0011 0111`)
- Offset = 000000
- PA = 0b`00 1101 1100 0000` = **0xDC0**

```
PhysAddr = (PTE.PFN << SHIFT) + offset
```



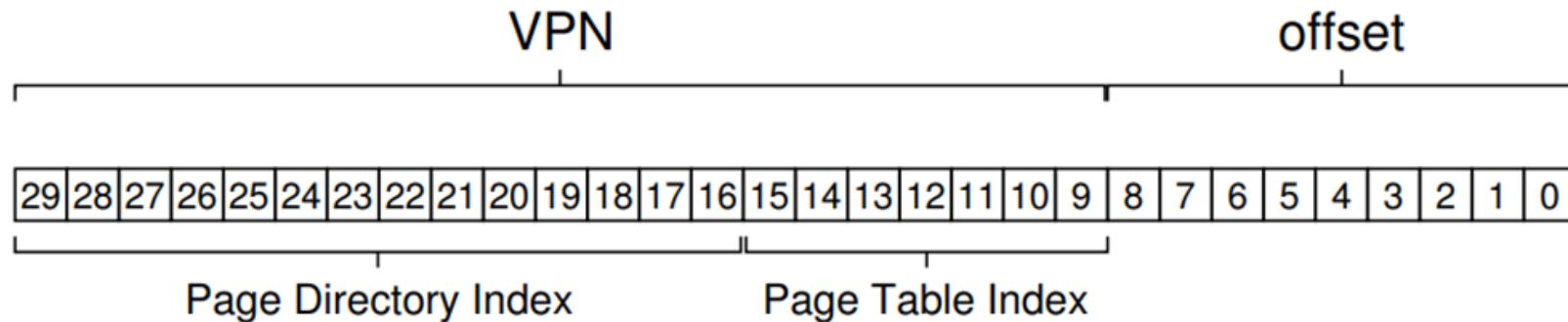
More Than Two Levels

- In some cases, a deeper tree is required. Consider the following address space.

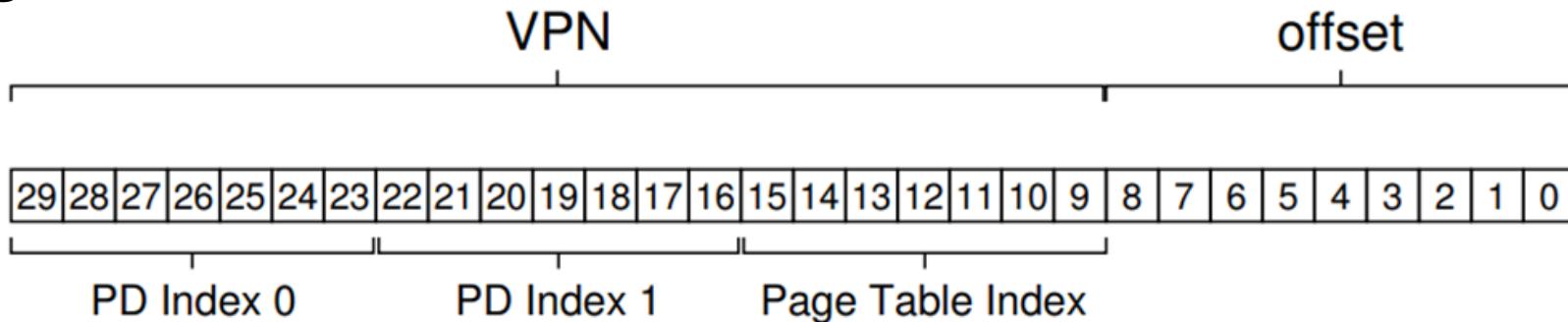
Specifications	Details
Virtual Address	30-bit
Page Size	512 bytes
VPN	21-bit
Offset	9-bit
PTE Size	4 bytes

- One page (512 bytes) can fit $512/4 = 128$ PTEs
 - 7 bits are needed for PTIndex
 - 14 bits are needed for PDIndex

More Than Two Levels

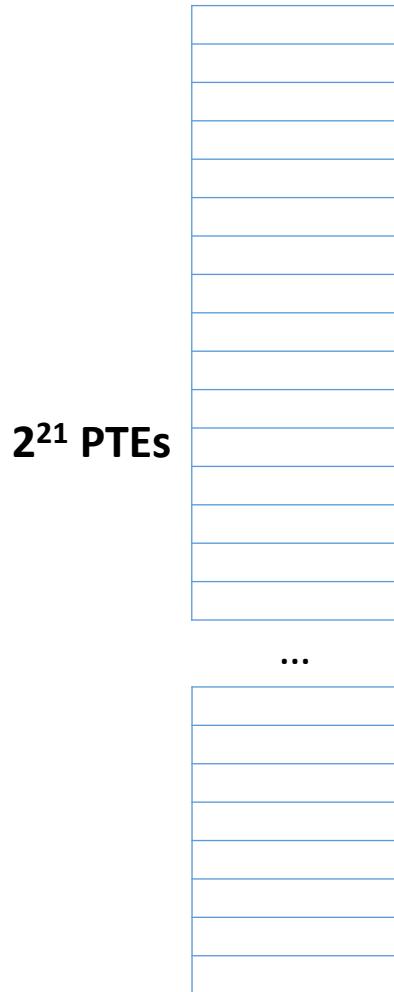


- Page Directory has 2^{14} PDEs, which occupies $2^{14} \times 4\text{B} = 2^{16}\text{ B}$
→ needs $2^{16}/512 = 2^7 = 128$ pages
- To remedy this, a further level is built by splitting the page directory into pages.

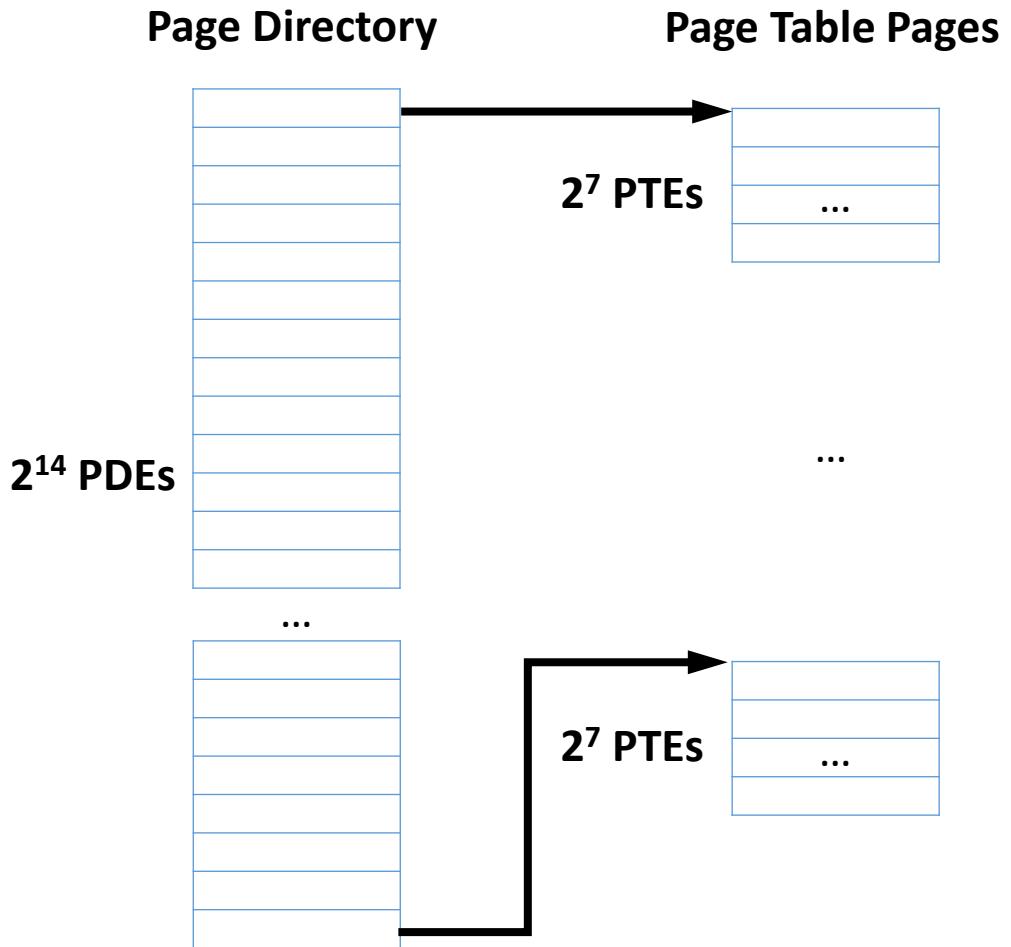


Multi-Level Page Table

Linear Page Table

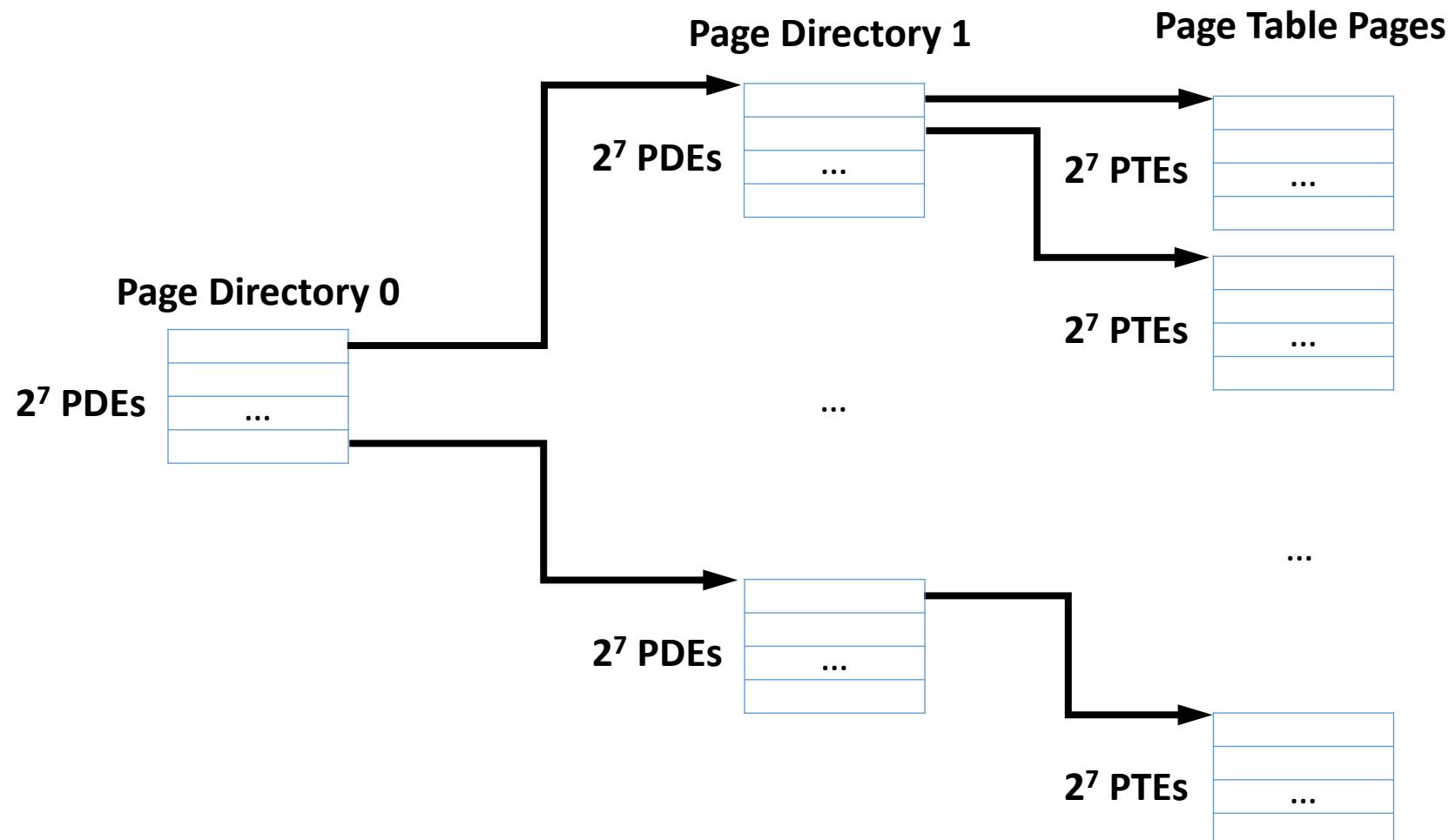


Two-Level Page Table



Multi-Level Page Table

Three-Level Page Table



Two-Level Page Table Control Flow

```
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         // first, get page directory entry
12         PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13         PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14         PDE = AccessMemory(PDEAddr)
```

Two-Level Page Table Control Flow

```
15         if (PDE.Valid == False)
16             RaiseException(SEGMENTATION_FAULT)
17         else
18             // PDE is valid: now fetch PTE from page table
19             PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20             PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21             PTE = AccessMemory(PTEAddr)
22             if (PTE.Valid == False)
23                 RaiseException(SEGMENTATION_FAULT)
24             else if (CanAccess(PTE.ProtectBits) == False)
25                 RaiseException(PROTECTION_FAULT)
26             else
27                 TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28                 RetryInstruction()
```

Inverted Page Tables

- Instead of many page tables (one per process), we keep 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.
- A linear scan would be expensive, and thus a hash table is often built to speed up lookups.

Summary

- Page tables can be built with **linear arrays**, or **complex data structures**.
- The trade-offs such tables present are in time and space. The bigger the table, the faster a TLB miss can be serviced.
- The right choice of structure depends strongly on the constraints of the given environment.

COMP 4736

Introduction to Operating Systems

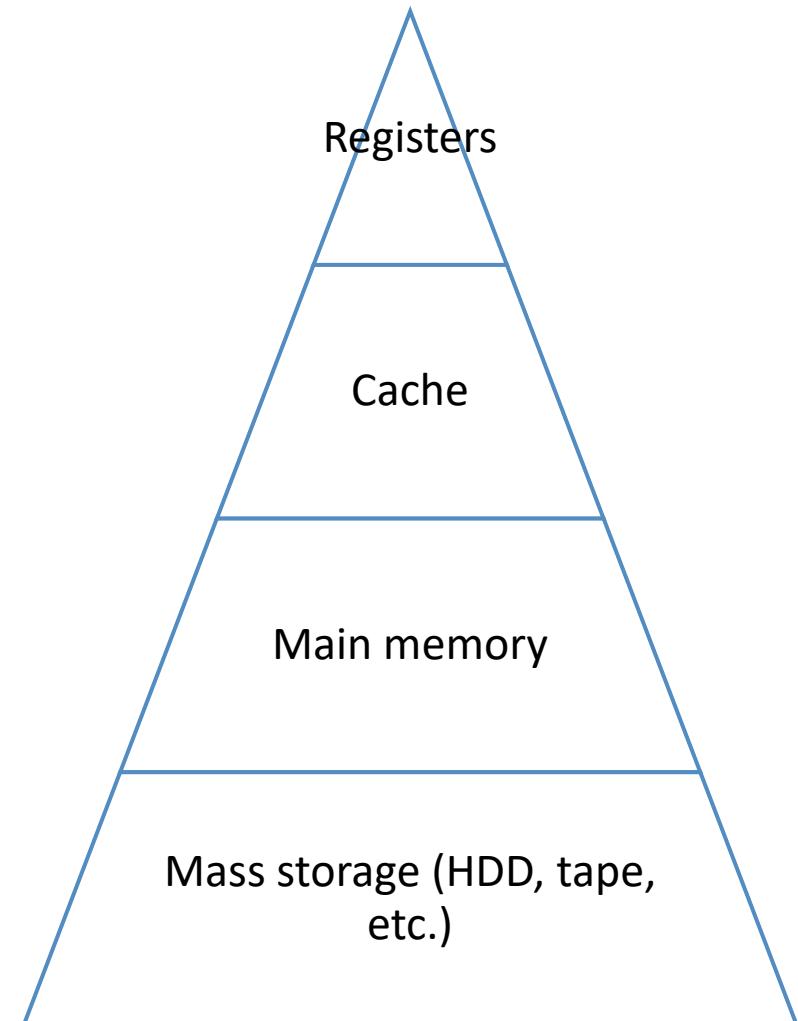
09. Swapping
(OSTEP Ch. 21 & 22)

Ch. 21. Beyond Physical Memory: Mechanisms

(OSTEP Ch. 21)

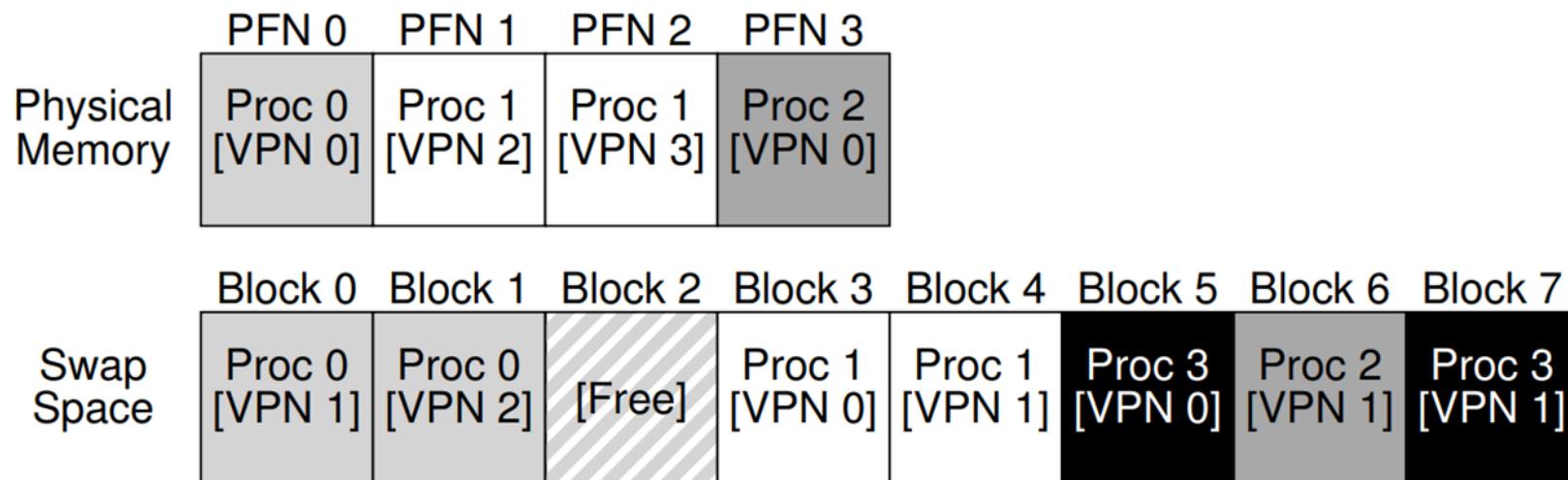
Beyond Physical Memory: Mechanisms

- We have been assuming that every address space of every running process fits into memory. Let's relax this assumption.
- To support many concurrently-running large address spaces, we 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**.



Swap Space

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



Present Bit

- Add some machinery higher up in the system in order to support swapping the 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.



- **P:** Present bit
 - 1: the page is present in physical memory
 - 0: the page is not in memory but rather on disk somewhere

Page Fault

- **Page fault**
 - Occurs when accessing page that is not in physical memory.
 - If a page is not present and has been swapped to disk, the OS needs to swap the page back into memory in order to service the page fault.
- The OS can use the bits in the PTE normally used for data such as **PFN** for a **disk address**.
- When the disk I/O completes, OS will need to update the page table.
 - Mark the page as **present**.
 - Update the **PFN** field of PTE of the newly-fetched page.
- TLB miss may be generated and TLB may need to be updated.
- When the I/O is in flight, the process will be **blocked**.

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
9              RaiseException(PROTECTION_FAULT)
10     else                      // TLB Miss
11         PTEAddr = PTBR + (VPN * sizeof(PTE))
12         PTE = AccessMemory(PTEAddr)
13         if (PTE.Valid == False)
14             RaiseException(SEGMENTATION_FAULT)
```

Page Fault Control Flow (Hardware)

```
15      else
16          if (CanAccess(PTE.ProtectBits) == False)
17              RaiseException(PROTECTION_FAULT)
18          else if (PTE.Present == True)
19              // assuming hardware-managed TLB
20              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21              RetryInstruction()
22          else if (PTE.Present == False)
23              RaiseException(PAGEFAULT)
```

Page Fault Control Flow (Software)

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

- First, the OS must find a physical frame for the **soon-to-be-faulted-in page** to reside within.
- If there is no such page, we'll have to wait for the **replacement algorithm** to run and kick some pages out of memory.

When Replacements Really Occur

- We have assumed that the OS waits until memory is **entirely full**, and only then **replaces** (evicts) a page to make room for some other page.
- To keep a small amount of memory free, most operating systems have some kind of **high watermark** (HW) and **low watermark** (LW) to help decide when to start evicting pages from memory.
- **Swap daemon/page daemon**
 - When OS notices that there are fewer than LW pages available, this daemon runs to free memory.
 - The daemon evicts pages until there are HW pages available, then it goes to sleep.

Summary

- To access more memory than it is physically present, we may **swap** memory pages between **memory** and **hard disk drive**.
- **Present** bit and **page fault handler** are needed. **Page replacement** may also be required.
- All these actions take place **transparently** to the process.

Ch. 22. Beyond Physical Memory: Policies

(OSTEP Ch. 22)

How to Decide Which Page to Evict?

- Memory can be viewed as a **cache** for virtual memory pages.
- Thus, our goal picking a replacement policy for this cache is to minimize the number of **cache misses**.
- **Average memory access time (AMAT)**

$$AMAT = T_M + (P_{\text{miss}} \cdot T_D)$$

- T_M = Cost of accessing memory
- T_D = Cost of accessing disk
- P_{miss} = Probability of a cache miss (from 0.0 to 1.0)

The Optimal Replacement Policy

- An **optimal** policy was developed many years ago.
 - Leads to the fewest number of misses overall.
 - Replaces the page that will be accessed **furthest in the future**.
 - Results in the fewest-possible cache misses.
- The approach is simple but difficult to implement.
- It is not very practical, but is incredibly useful as a comparison point.

Tracing the Optimal Policy

- Assume a program accesses the following stream of virtual pages:
0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1
- The cache can only fit **three** pages.

Hit rate

Access	Hit/Miss?	Evict	Resulting Cache State	Hit rate
0	Miss		0	$\frac{Hits}{Hits + Misses}$
1	Miss		0, 1	$= \frac{6}{6 + 5}$
2	Miss		0, 1, 2	$= 54.5\%$
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	

Unfortunately, the future is not generally 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

- Reminder:
 - Reference stream: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1
 - Max pages in cache: 3

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in→ 0
1	Miss		First-in→ 0, 1
2	Miss		First-in→ 0, 1, 2
0	Hit		First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2
3	Miss	0	First-in→ 1, 2, 3
0	Miss	1	First-in→ 2, 3, 0
3	Hit		First-in→ 2, 3, 0
1	Miss	2	First-in→ 3, 0, 1
2	Miss	3	First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2

Hit rate

$$\begin{aligned} \text{Hit rate} &= \frac{\text{Hits}}{\text{Hits} + \text{Misses}} \\ &= \frac{4}{4 + 7} \\ &= 36.4\% \end{aligned}$$

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

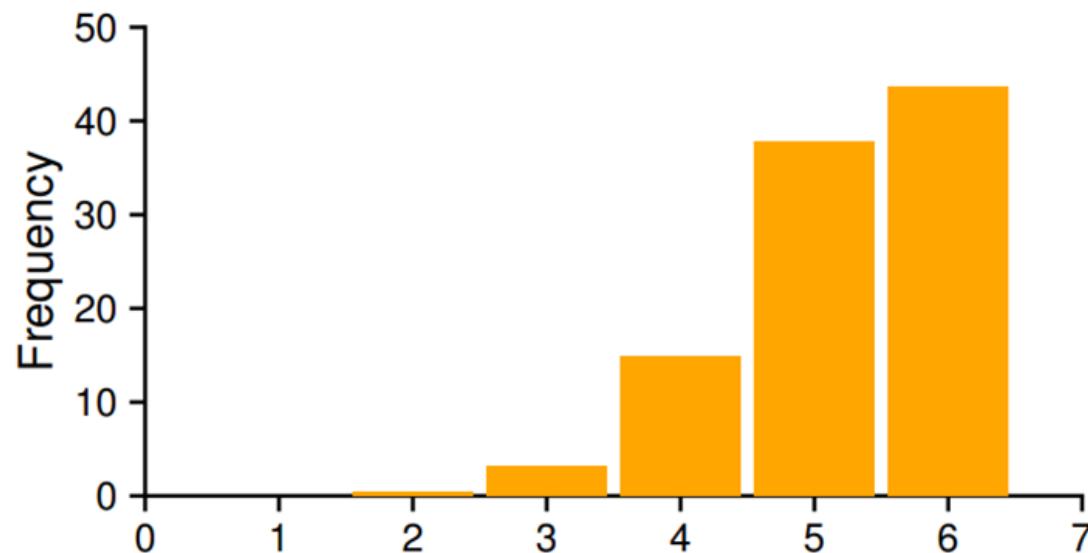
Another Simple Policy: Random

- 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

- Below shows how many hits Random achieves over 10,000 trials.
- As you can see, sometimes (just over 40% of the time), Random is as good as optimal, achieving 6 hits on the example trace.
- Sometimes it does much worse, achieving 2 hits or fewer.



Using History

- We can use **history** to improve our guess in the future, using the **principle of locality**.
- **Frequency**
 - If a page has been accessed many times, perhaps it should not be replaced as it clearly has some value.
 - **Least-Frequently-Used (LFU)** policy
- **Recency**
 - The more recently a page has been accessed, perhaps the more likely it will be accessed again.
 - **Least-Recently-Used (LRU)** policy

Using History: LRU

- Replaces the least-recently used page.

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

Hit rate

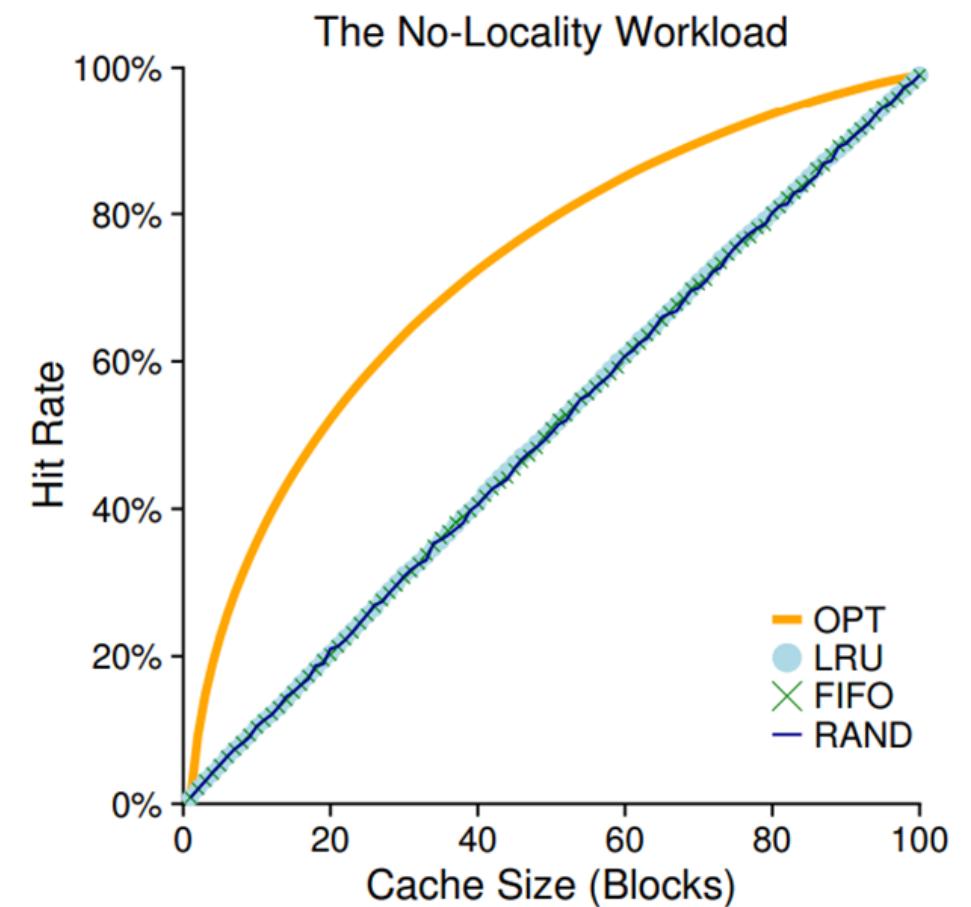
$$= \frac{\text{Hits}}{\text{Hits} + \text{Misses}}$$

$$= \frac{6}{6 + 5}$$
$$= 54.5\%$$

Workload Examples: 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.

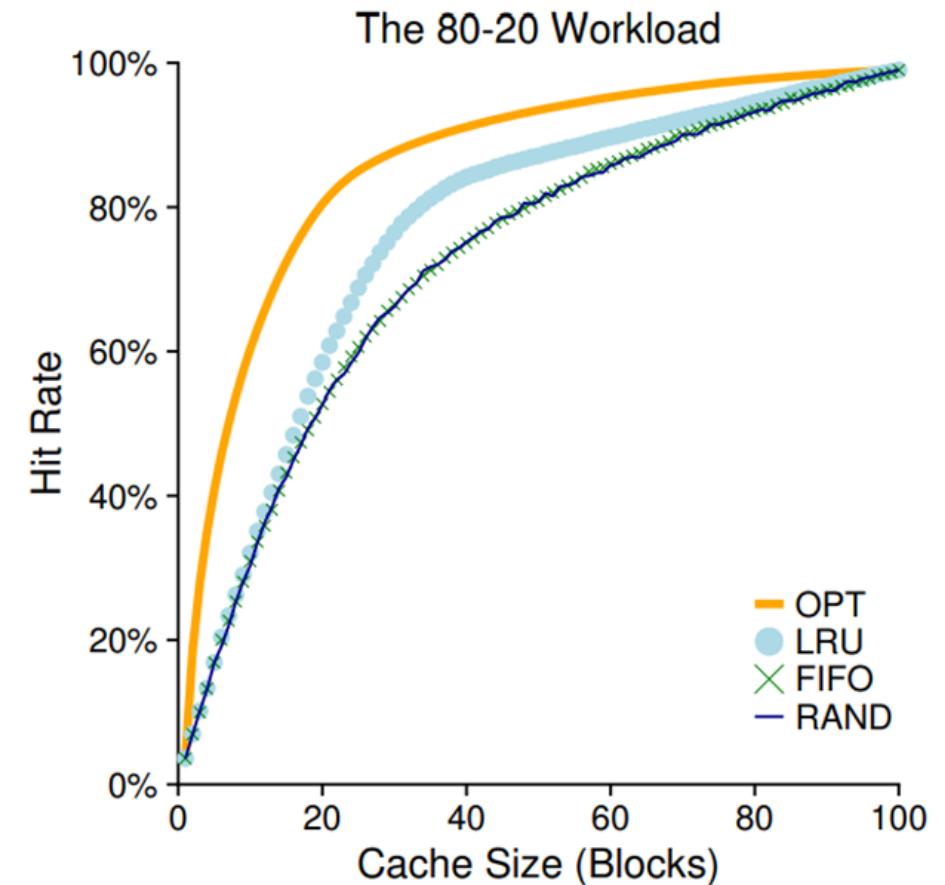
When the cache is large enough to fit the entire workload, it also **doesn't matter** which policy you use.



Workload Examples: The 80-20 Workload

- This workload exhibits locality: 80% of the reference are made to 20% of the page (“**hot**” pages).
- The remaining 20% of the reference are made to the remaining 80% of the pages (“**cold**” pages).

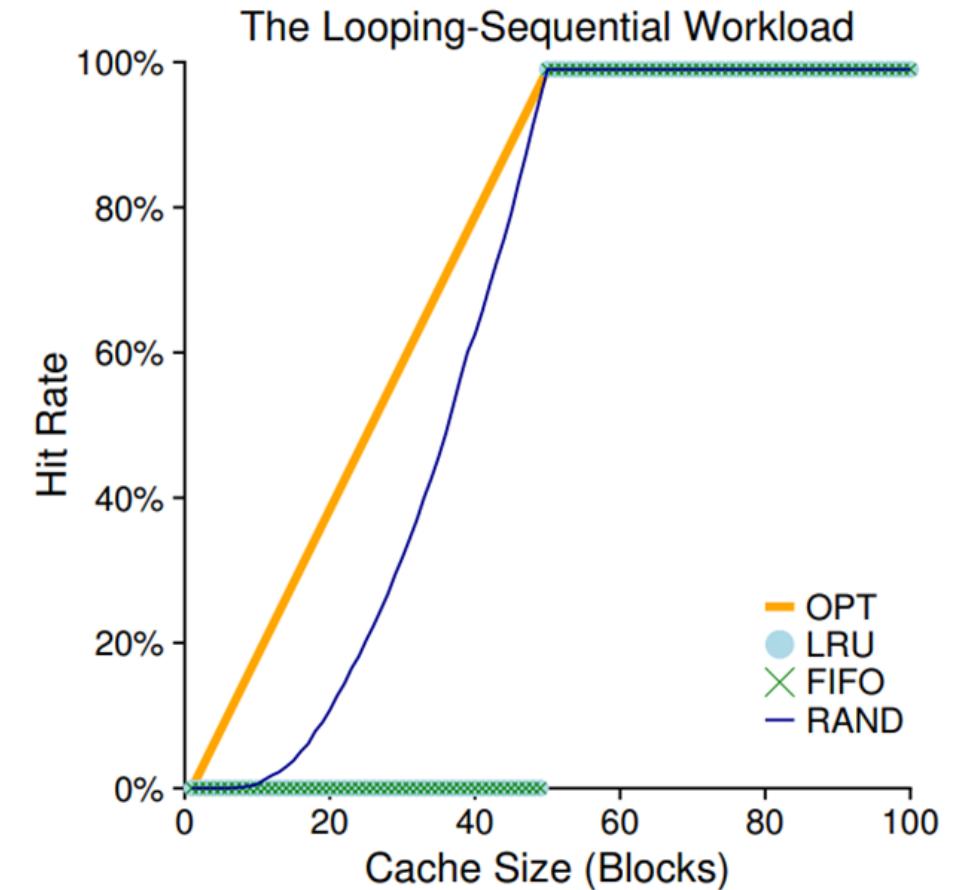
LRU is more likely to hold onto the **hot** pages.



Workload Examples: 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.

Worst-case for both **LRU** and **FIFO**.
Random fares notably better, not having weird corner-case behaviours.

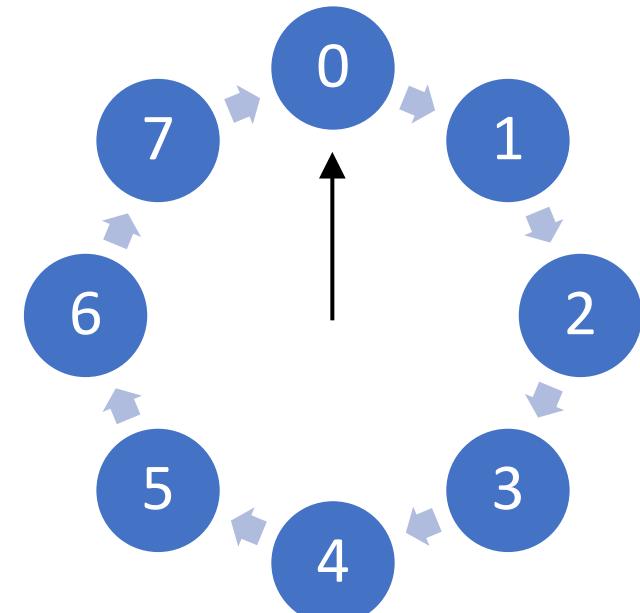


Implementing Historical Algorithms

- To implement **LRU** perfectly, a lot of work is needed.
- Upon **each page access**, we must update some data structure to move this page to the front of the list.
- Contrast to FIFO, where the is only accessed when a page is **evicted** (by removing the first-in page) or when a new page is **added** (to the last-in side).
- To keep track of which pages have been least- and most-recently used, the system has to do some accounting work on **every memory reference**.
- Such accounting could greatly reduce performance.

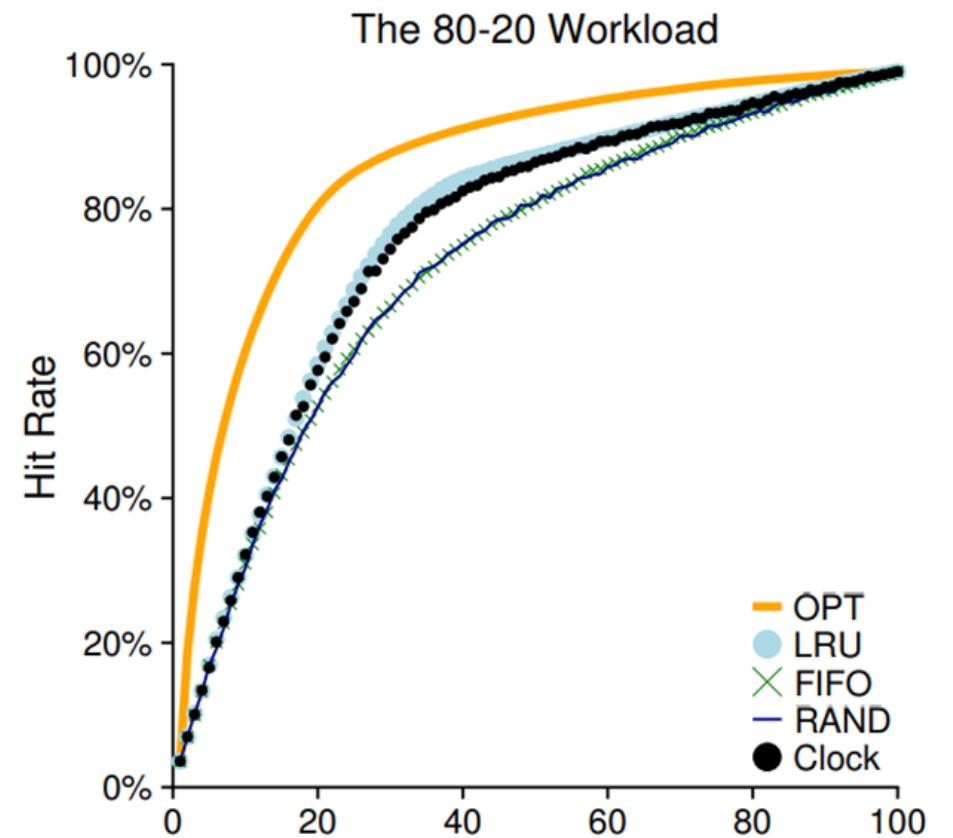
Approximating LRU: Clock Algorithm

- Require hardware support: 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.
 - When a replacement is needed, check currently pointed page.
 - If use bit = 1, clear it and clock hand is incremented to next page. If use bit = 0, evict this page.



The 80-20 Workload With Clock

- Although it doesn't do quite as well as **perfect LRU**, **clock algorithm** does better than approaches that don't consider history at all.



Considering Dirty Pages

- The hardware includes 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 (which is expensive).
- Page has not been modified (and is thus **clean**), the eviction is **free**.
- For example, the **clock algorithm** could be changed to...
 - Scan for pages that are both **unused** and **clean** to evict first.
 - Failing to find those, then for **unused** pages that are **dirty**, and so forth.

Other VM Policies

- For most pages, the OS simply uses **demand paging**: brings the page into memory when it is accessed.
- OS could also perform **prefetching**: guess that a page is about to be used, and thus bring it in ahead of time.
- For example, if a code page P is brought into memory, that code page $P+1$ will likely soon be accessed and thus should be brought into memory too.
- Instead of writing pages out to disk one at a time, OS can collect a number of pending writes together in memory and write them to disk in one (more efficient) write, called **clustering** or simply **grouping** of writes
- It is effective because of the nature of disk drives, which perform a single large write more efficiently than many small ones.

Thrashing

- When memory is simply **oversubscribed**, and the memory demands of the set of running processes simply **exceeds** the available physical memory, the system will constantly be paging.
- This is sometimes referred to as **thrashing**.
- OS could decide not to run a subset of processes (**admission control**), or run an **out-of-memory killer** when memory is oversubscribed.

Summary

- Different **page-replacement policies** are introduced (e.g., FIFO, random, LRU, etc.).
- Modern systems add some tweaks to straightforward LRU approximations like clock, to try to avoid the worst-case behaviour of LRU.
- Recent innovations in much faster storage devices leads to the renaissance in page replacement algorithms.

COMP 4736

Introduction to Operating Systems

10. Concurrency I
(OSTEP Ch. 26 & 27)

Ch. 26. Concurrency: An Introduction

(OSTEP Ch. 26)

Thread

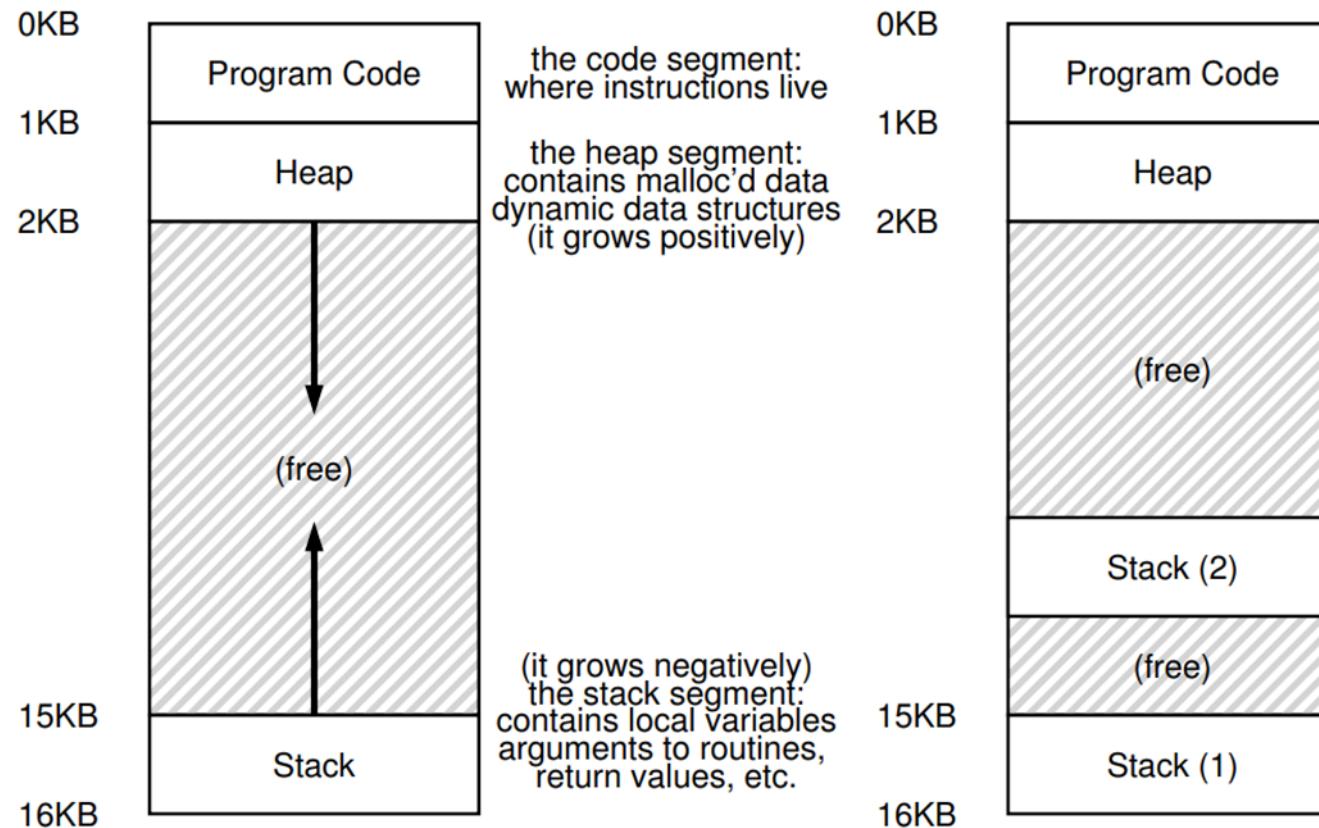
- Thread: a new abstraction for a **single running process**.
- **Multi-threaded** program
 - A multi-threaded program has more than one point of execution.
 - Multiple PCs (Program Counter)
 - They **share** the same address space.

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 must be saved.
 - The register state of T2 must be restored.
 - The **address space** remains the same (i.e., no need to switch page table).

Single-Threaded and Multi-Threaded Address Spaces

- Instead of a single stack in the address space, there will be **one per thread**.



Why Use Threads?

- **Parallelism**
 - Single-threaded program: the task is straightforward, but slow.
 - Multi-threaded program: natural and typical way to make programs run faster on modern hardware.
 - **Parallelization:** The task of transforming standard single-threaded program into a program that does this sort of work on multiple CPUs.
- Avoid blocking program progress due to slow I/O.
 - Threading enables **overlap** of I/O with other activities within a single program.
 - It is much like **multiprogramming** did for processes across programs.

An Example: Thread Creation (t0.c)

```
1 #include <stdio.h>
2 #include <cassert.h>
3 #include <pthread.h>
4 #include "common.h"
5 #include "common_threads.h"
6
7 void *mythread(void *arg) {
8     printf("%s\n", (char *) arg);
9     return NULL;
10 }
11
12 int main(int argc, char *argv[]) {
13     pthread_t p1, p2;
14     int rc;
15     printf("main: begin\n");
16     Pthread_create(&p1, NULL, mythread, "A");
17     Pthread_create(&p2, NULL, mythread, "B");
18     // join waits for the threads to finish
19     Pthread_join(p1, NULL);
20     Pthread_join(p2, NULL);
21     printf("main: end\n");
22     return 0;
23 }
```

Thread Trace (1)

main	Thread 1	Thread 2
starts running prints “main: begin” create Thread 1 create Thread 2 waits for T1		
	runs prints “A” returns	
waits for T2		
		runs prints “B” returns
prints “main: end”		

Thread Trace (2)

main	Thread 1	Thread 2
starts running prints “main: begin” create Thread 1		
	runs prints “A” returns	
create Thread 2		
		runs prints “B” returns
waits for T1 <i>returns immediately; T1 is done</i> waits for T2 <i>returns immediately; T2 is done</i> prints “main: end”		

Thread Trace (3)

main	Thread 1	Thread 2
starts running prints “main: begin” create Thread 1 create Thread 2		
		runs prints “B” returns
waits for T1		
	runs prints “A” returns	
waits for T2 <i>returns immediately; T2 is done</i> prints “main: end”		

Sharing Data (t1.c)

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include "common.h"
4 #include "common_threads.h"
5
6 static volatile int counter = 0;
7
8 // mythread()
9 //
10 // Simply adds 1 to counter repeatedly, in a loop
11 // No, this is not how you would add 10,000,000 to
12 // a counter, but it shows the problem nicely.
13 //
14 void *mythread(void *arg) {
15     printf("%s: begin\n", (char *) arg);
16     int i;
17     for (i = 0; i < 1e7; i++) {
18         counter = counter + 1;
19     }
20     printf("%s: done\n", (char *) arg);
21     return NULL;
22 }
23
```

Sharing Data (t1.c)

```
24 // main()
25 //
26 // Just launches two threads (pthread_create)
27 // and then waits for them (pthread_join)
28 //
29 int main(int argc, char *argv[]) {
30     pthread_t p1, p2;
31     printf("main: begin (counter = %d)\n", counter);
32     Pthread_create(&p1, NULL, mythread, "A");
33     Pthread_create(&p2, NULL, mythread, "B");
34
35     // join waits for the threads to finish
36     Pthread_join(p1, NULL);
37     Pthread_join(p2, NULL);
38     printf("main: done with both (counter = %d)\n", counter);
39     return 0;
40 }
```

Sharing Data

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

The Heart of the Problem

- counter = counter + 1

```
100    mov 0x8049a1c, %eax  
105    add $0x1, %eax  
108    mov %eax, 0x8049a1c
```

- Assume counter is located at 0x8049a1c.

Race Condition

- Counter is incremented twice; should be 52 but is now 51.

OS	Thread 1	Thread 2	PC	eax	counter
	<i>before critical section</i> mov 8049a1c,%eax add \$0x1,%eax		100 105 108	0 50 51	50 50 50
interrupt <i>save T1</i> <i>restore T2</i>			100	0	50
		mov 8049a1c,%eax add \$0x1,%eax mov %eax,8049a1c	105 108 113	50 51 51	50 50 51
interrupt <i>save T2</i> <i>restore T1</i>			108	51	51
	mov %eax,8049a1c		113	51	51

A Few Terminologies

- **Race condition**
 - The results depend on the timing of the code's execution.
 - Result is **indeterminate**.
- **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**), to guarantee if one thread is executing within the critical section, the others will be prevented from doing so.

The Wish for Atomicity

- If we had a super instruction that executes **atomically**. So it cannot be interrupted mid-instruction.

```
memory-add 0x8049a1c, $0x1
```

- In general, we do not have such instruction.
- Instead, based upon a few useful instructions, we could build a general set of **synchronization primitives** to support atomicity.
- Ensure that any such critical section executes as if it were a single atomic instruction.

```
mov 0x8049a1c, %eax  
add $0x1, %eax  
mov %eax, 0x8049a1c
```

Summary

- A process can be abstracted as a **thread**. **Multi-threaded** program has **more than** one point of execution.
- When a resource is **shared** among threads, **race condition** arises if multiple threads enter the **critical section** at the same time. The result of such program is **indeterminate**.
- **Synchronization primitives**, such as **mutual exclusion** primitives, should be used to guarantee only a single thread ever enters a critical section.

Ch. 27. Thread API

(OSTEP Ch. 27)

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.
 - E.g., stack size, scheduling priority, etc.
- **start_routine**: The function this thread starts 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

- If `start_routine` instead requires another type argument, the declaration would look like this:
- Requires an integer argument:

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

- Returns an integer:

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

Example: Creating a Thread

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 typedef struct {
5     int a;
6     int b;
7 } myarg_t;
8
9 void *mythread(void *arg) {
10     myarg_t *args = (myarg_t *) arg;
11     printf("%d %d\n", args->a, args->b);
12     return NULL;
13 }
14
15 int main(int argc, char *argv[]) {
16     pthread_t p;
17     myarg_t args = { 10, 20 };
18
19     int rc = pthread_create(&p, NULL, mythread, &args);
20     ...
21 }
```

Wait for a Thread to Complete

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

- **thread**: Used to specify which thread to wait for.
- **value_ptr**: A pointer to the expected return value.
 - Because `pthread_join()` routine changes the value of the argument, you need to pass a **pointer** to that value.

Example: Waiting for Thread Completion

```
1 typedef struct { int a; int b; } myarg_t;
2 typedef struct { int x; int y; } myret_t;
3
4 void *mythread(void *arg) {
5     myret_t *rvals = Malloc(sizeof(myret_t));
6     rvals->x = 1;
7     rvals->y = 2;
8     return (void *) rvals;
9 }
10
11 int main(int argc, char *argv[]) {
12     pthread_t p;
13     myret_t *rvals;
14     myarg_t args = { 10, 20 };
15     Pthread_create(&p, NULL, mythread, &args);
16     Pthread_join(p, (void **) &rvals);
17     printf("returned %d %d\n", rvals->x, rvals->y);
18     free(rvals);
19
20 }
```

Be Careful About Returning Values

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

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

- When the variable **oops** returns, it is automatically **deallocated**.
 - Fortunately the compiler gcc will likely complain when you write code like this, which is yet another reason to pay attention to compiler warnings.

Example: Simpler Argument Passing to a Thread

- Just passing in a single value

```
void *mythread(void *arg) {
    long long int value = (long long int) arg;
    printf("%lld\n", value);
    return (void *) (value + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    long long int rvalue;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &rvalue);
    printf("returned %lld\n", rvalue);
    return 0;
}
```

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 (without 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

- 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

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

```
// Keeps code clean; only use if exit() 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 (in general, avoid them).

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

- **trylock** version: Returns failure if the lock is already held
- **timelock** version: Returns after a timeout or after acquiring the lock, whichever happens first.

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

- Thread going to sleep

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
  
Pthread_mutex_lock(&lock);  
while (ready == 0)  
    Pthread_cond_wait(&cond, &lock);  
Pthread_mutex_unlock(&lock);
```

- The wait call **releases** the lock when putting the caller to sleep.
- **Before** returning after being woken, the wait call **re-acquires** the lock.
- Thread waking others up

```
Pthread_mutex_lock(&lock);  
ready = 1;  
Pthread_cond_signal(&cond);  
Pthread_mutex_unlock(&lock);
```

Condition Variables

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

```
Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &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

- Don't ever do the following.
 - Thread going to sleep

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

- Thread waking others up

```
ready = 1;
```

- It performs poorly in many cases. → Spinning for a long time just wastes CPU cycles.
- It is error prone. → In a study, roughly half the uses of these ad hoc synchronization were buggy! → Use condition variables instead.

Compiling and Running

- To compile them, you must include the header `pthread.h`.
- Also 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
```

Summary

- The basics of the pthread library are introduced, including **thread creation**, building mutual exclusion via **locks**, and signaling and waiting via **condition variables**.

COMP 4736

Introduction to Operating Systems

11. Concurrency II
(OSTEP Ch. 30 & 32)

Ch. 30. Condition Variables

(OSTEP Ch. 30)

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

```
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); // child
11    // XXX how to wait for child?
12    printf("parent: end\n");
13    return 0;
14 }
```

- What we would like to see

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

Parent Waiting For 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); // child
13    while (done == 0)
14        ; // spin
15    printf("parent: end\n");
16    return 0;
17 }
```

- It 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 its state, can wake one of those waiting threads and thus allow them to continue.
- Three items are needed
 - Condition variable
 - State variable (e.g., done)
 - Lock (e.g., mutex)

Definition and Routines

- To declare a condition variable.

```
pthread_cond_t c;
```

- Proper initialization is required.
- The POSIX operation 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 releases the lock and puts the calling thread to sleep **atomically**.
 - When the thread wakes up, it must re-acquire the lock **before** returning.

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
```

Parent Waiting For Child: Use a Condition Variable

```
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
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:**

- Creates the child thread and continues running itself.
- Calls into `thr_join()` to wait for the child thread to complete.
 - Acquires the lock.
 - Checks if the child is done.
 - Puts itself to sleep by calling `wait()`.
 - Releases the lock.

- **Child:**

- Prints the message “child”.
- Calls `thr_exit()` to wake up the parent thread.
 - Grabs the lock.
 - Sets the state variable `done`.
 - Signals the parent thus waking it.

Parent Waiting: No State Variable

```
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 }
```

- 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.

Parent Waiting: No Lock

```
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 into 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 (v1)

```
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 (v1)

```
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    while (1) {
11        int tmp = get();
12        printf("%d\n", tmp);
13    }
14 }
```

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

Producer/Consumer: Single CV And If Statement

```
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex);                      // c1
21         if (count == 0)                                  // c2
22             Pthread_cond_wait(&cond, &mutex);           // c3
23         int tmp = get();                                // c4
24         Pthread_cond_signal(&cond);                  // c5
25         Pthread_mutex_unlock(&mutex);                // c6
26         printf("%d\n", tmp);
27     }
28 }
```

- 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.

Thread Trace: Broken Solution (v1)

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

Broken Solution (v1)

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

Producer/Consumer: Single CV and While

```
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex);                      // c1
21         while (count == 0)                                // c2
22             Pthread_cond_wait(&cond, &mutex);           // c3
23         int tmp = get();                                // c4
24         Pthread_cond_signal(&cond);                   // c5
25         Pthread_mutex_unlock(&mutex);                 // c6
26         printf("%d\n", tmp);
27     }
28 }
```

- 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 (v2)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	
	Sleep	c1	Run		Ready	0	
	Sleep	c2	Run		Ready	0	
	Sleep	c3	Sleep		Ready	0	
	Sleep		Sleep	p1	Run	0	
	Sleep		Sleep	p2	Run	0	
	Sleep		Sleep	p4	Run	1	Buffer now full
	Ready		Sleep	p5	Run	1	T_{c1} awoken
	Ready		Sleep	p6	Run	1	
	Ready		Sleep	p1	Run	1	
	Ready		Sleep	p2	Run	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Run		Sleep		Sleep	1	Recheck condition
c4	Run		Sleep		Sleep	0	T_{c1} grabs data
c5	Run		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Run		Ready		Sleep	0	
c1	Run		Ready		Sleep	0	
c2	Run		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	
	Sleep	c2	Run		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

```
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 Correct Put And Get Routines

- To enable more concurrency and efficiency. → Add more buffer slots.
 - Reduce context switches (for single producer and consumer).
 - Allow concurrent production or consuming to take place (for multiple producers and/or consumers).

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

The Correct Producer/Consumer Synchronization

```
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);                                    // p4
11        Pthread_cond_signal(&fill);                 // p5
12        Pthread_mutex_unlock(&mutex);                // p6
13    }
14 }
15
```

The Correct Producer/Consumer Synchronization

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

- A producer only sleeps if **all buffers** are currently **filled** (p2).
- Similarly, a consumer only sleeps if **all buffers** are currently **empty** (c2).

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: An Example

```
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

- 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.

Summary

- Another important synchronization primitive beyond locks, **condition variables**, is introduced.
- Allowing threads to **sleep** when some program **state** is not as desired.

Ch. 32. Common Concurrency Problems

(OSTEP Ch. 32)

Common Concurrency Problems

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

- Two classes of concurrency bugs to be discussed.
- **Non-deadlock** bugs (majority)
 - **Atomicity-violation** bugs
 - **Order-violation** bugs
- **Deadlock** bugs

Atomicity-Violation Bugs

- The desired **serializability** among multiple memory accesses is violated (i.e. a code region is intended to be **atomic**, but the atomicity is **not enforced** during execution).
- Simple Example found in MySQL:
- Two different threads access the field `proc_info` in the struct `thd`.

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

Atomicity-Violation Bugs

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

```
1 pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3 Thread 1::
4 pthread_mutex_lock(&proc_info_lock);
5 if (thd->proc_info) {
6     fputs(thd->proc_info, ...);
7 }
8 pthread_mutex_unlock(&proc_info_lock);
9
10 Thread 2::
11 pthread_mutex_lock(&proc_info_lock);
12 thd->proc_info = NULL;
13 pthread_mutex_unlock(&proc_info_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 Thread 1::  
2 void init() {  
3     mThread = PR_CreateThread(mMain, ...);  
4 }  
5  
6 Thread 2::  
7 void mMain(...) {  
8     mState = mThread->State;  
9 }
```

Order-Violation Bugs

- 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
```

Order-Violation Bugs

```
18 Thread 2::  
19 void mMain(...) {  
20     ...  
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

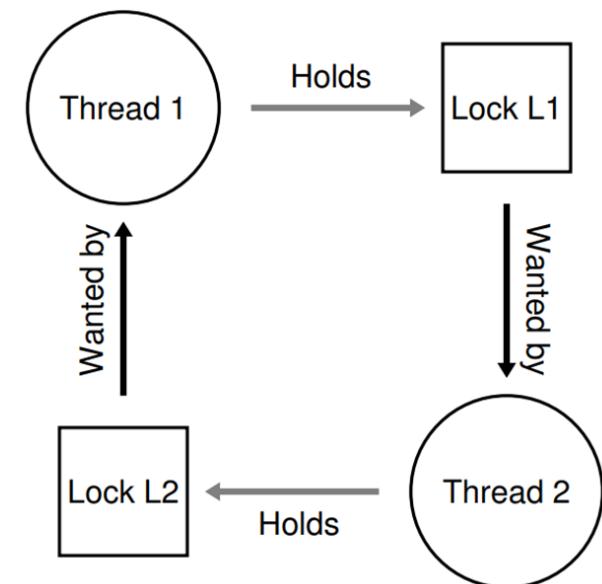
Thread 1:

```
pthread_mutex_lock(L1);  
pthread_mutex_lock(L2);
```

Thread 2:

```
pthread_mutex_lock(L2);  
pthread_mutex_lock(L1);
```

- Deadlock does not **necessarily** occur; rather, it **may** occur, if
 - Thread 1 is holding a lock L1 and waiting for another one, L2.
 - Thread 2 that holds lock L2 is waiting for L1 to be released.
- The presence of a **cycle** in the graph is indicative of the **deadlock**.



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?

- Java Vector class and the method AddAll()

```
Vector v1, v2;
```

```
Thread 1::
```

```
v1.AddAll(v2);
```

```
Thread 2::
```

```
v2.AddAll(v1);
```

- To be multi-thread safe, **locks** for both the vector being added to (v1) and the parameter (v2) need to be acquired.
- Potential for deadlock, all in a way that is quite hidden from the calling application.

Conditions for Deadlock

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

Condition	Descriptions
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 or 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 pthread_mutex_lock(prevention);      // begin acquisition
2 pthread_mutex_lock(L1);
3 pthread_mutex_lock(L2);
4 ...
5 pthread_mutex_unlock(prevention);    // end
```

- 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.
- `pthread_mutex_trylock()`
 - Can be 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     pthread_mutex_lock(L1);  
3     if (pthread_mutex_trylock(L2) != 0) {  
4         pthread_mutex_unlock(L1);  
5         goto top;  
6     }
```

Prevention—No Preemption

- **livelock**
 - Two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks.
 - Both systems are running through the code sequence over and over again.
 - Not a deadlock, but progress is not being made.
- Solution:
 - Add a random delay before looping back and trying the entire thing over again.

Prevention—Mutual Exclusion

- Lock-free and 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; // failure  
7 }
```

Prevention—Mutual Exclusion

- To **atomically** increment a value by a certain amount, using compare-and-swap.

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

- Instead of acquiring a lock, doing the update, and then releasing it...
- We repeatedly try to update the value to the new amount and use the compare-and-swap to do so.
- No deadlock can arise, though livelock is still possible.

Prevention—Mutual Exclusion

- Inserts at the head of a list (using locks):

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

- Inserts at the head of a list (using compare-and-swap):

```
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

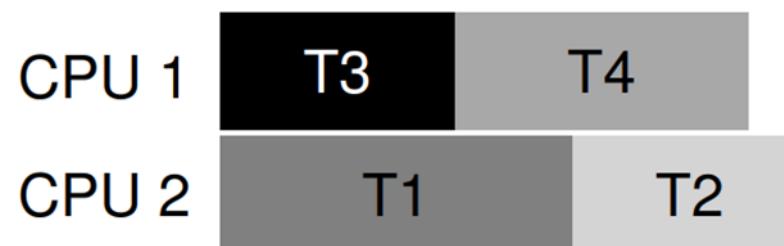
- **Deadlock Avoidance**
 - Get the information about the locks various threads might grab during their execution.
 - Schedule the threads in a way to **guarantee** no deadlock can occur.
- In some scenarios, deadlock avoidance is preferable.
- Problem: Global knowledge is required.

Deadlock Avoidance Example

- 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 thus compute that as long as **T1** and **T2** are **not run at the same time**, no deadlock could ever arise.

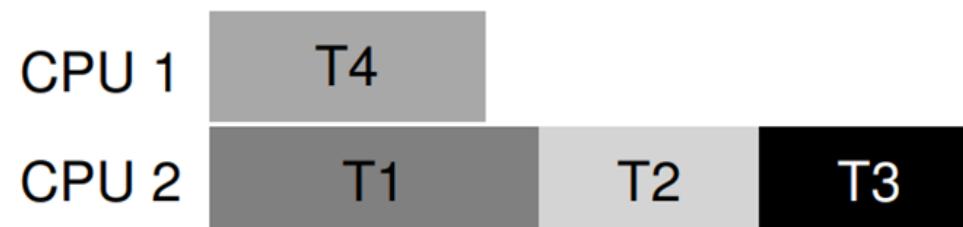


Deadlock Avoidance Example

- 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 (e.g., once a year), you would just reboot it.
- Many database systems employ **deadlock detection** and **recovery** techniques.
 - A deadlock detector runs **periodically**.
 - Building a resource graph and checking it for **cycles**.
 - In the event of a cycle (deadlock), the system need to be **restarted**.

Summary

- The types of bugs that occur in concurrent programs are studied.
- Non-deadlock bugs, including **atomicity violations** and **order violations**, are surprisingly common but often are easier to fix.
- For deadlocks, the best solution in practice is to be **careful** and develop a **lock acquisition order**.
- **Wait-free** approaches also have promise.