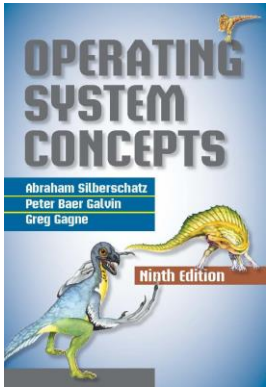
# CS222
# Operating Systems

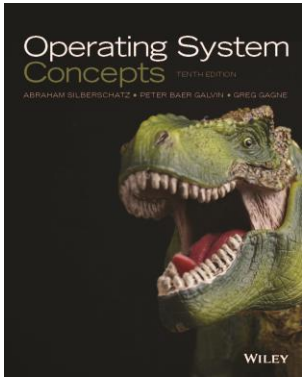
## Lecture 04

(Section 100001)
ผศ. ดร. กษิดิศ ชาญเชี่ยว
ckasidit@tu.ac.th

# Textbook

- Avi Silberschatz, Peter B. Galvin and Greg Gagne; Operating System Concepts, 9th Edition; John Wiley & Sons, Inc; 2012; ISBN 978-1118063330




- Original Slides
- https://www.os-book.com/OS9/slide-dir/index.html
- https://www.os-book.com/OS10/slide-dir/index.html

# Chapter 2

- Operating System Services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific
- Design and Implementation
- Operating System Structure
- Booting an Operating System

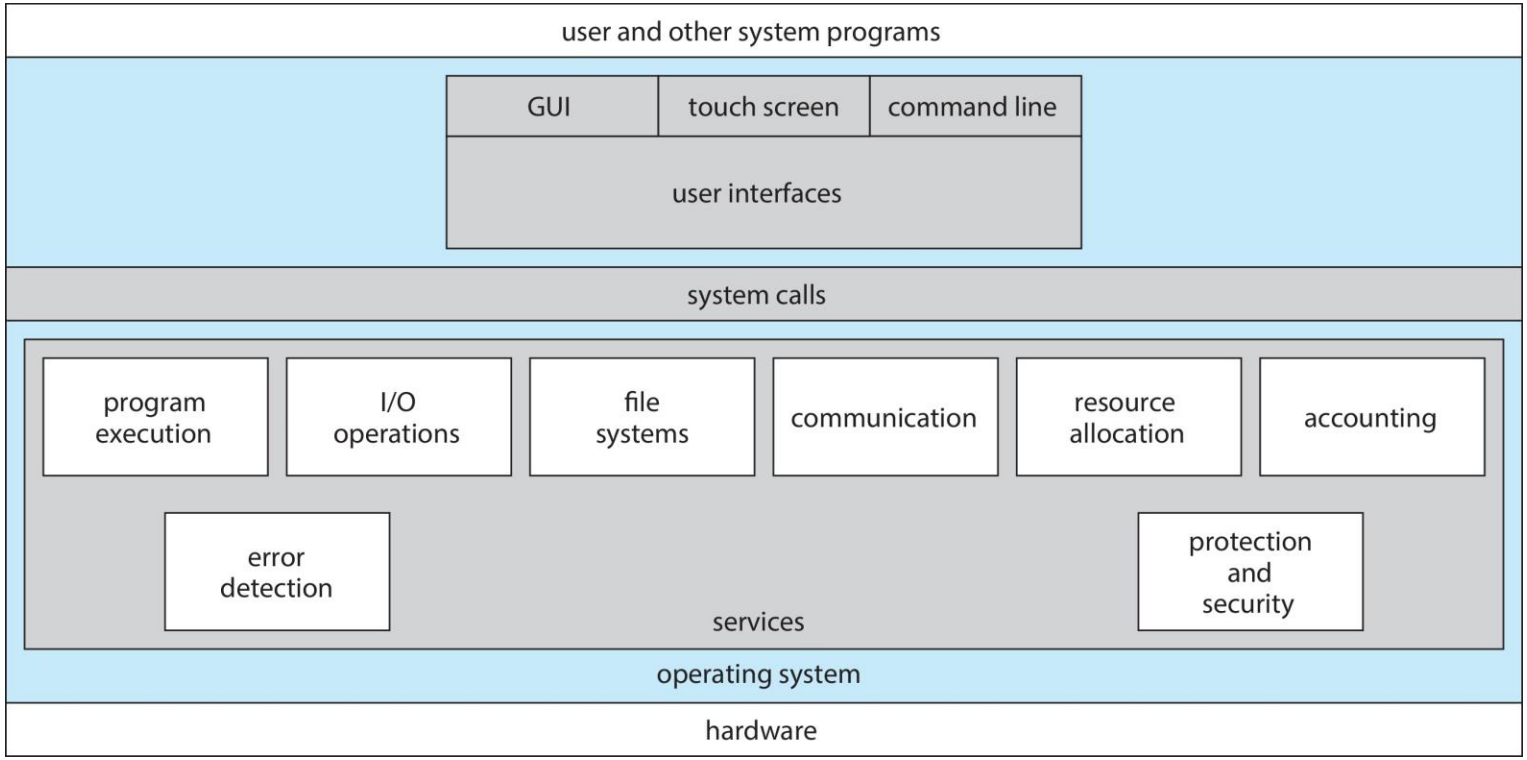
# Objectives

- Identify services provided by an operating system
- Illustrate how system calls are used to provide operating system services
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems
- Illustrate the process for booting an operating system
- Design and implement kernel modules for interacting with a Linux kernel

# Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - Varies between **Command-Line** (**CLI**), **Graphics User Interface** (**GUI**), **touch-screen, Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

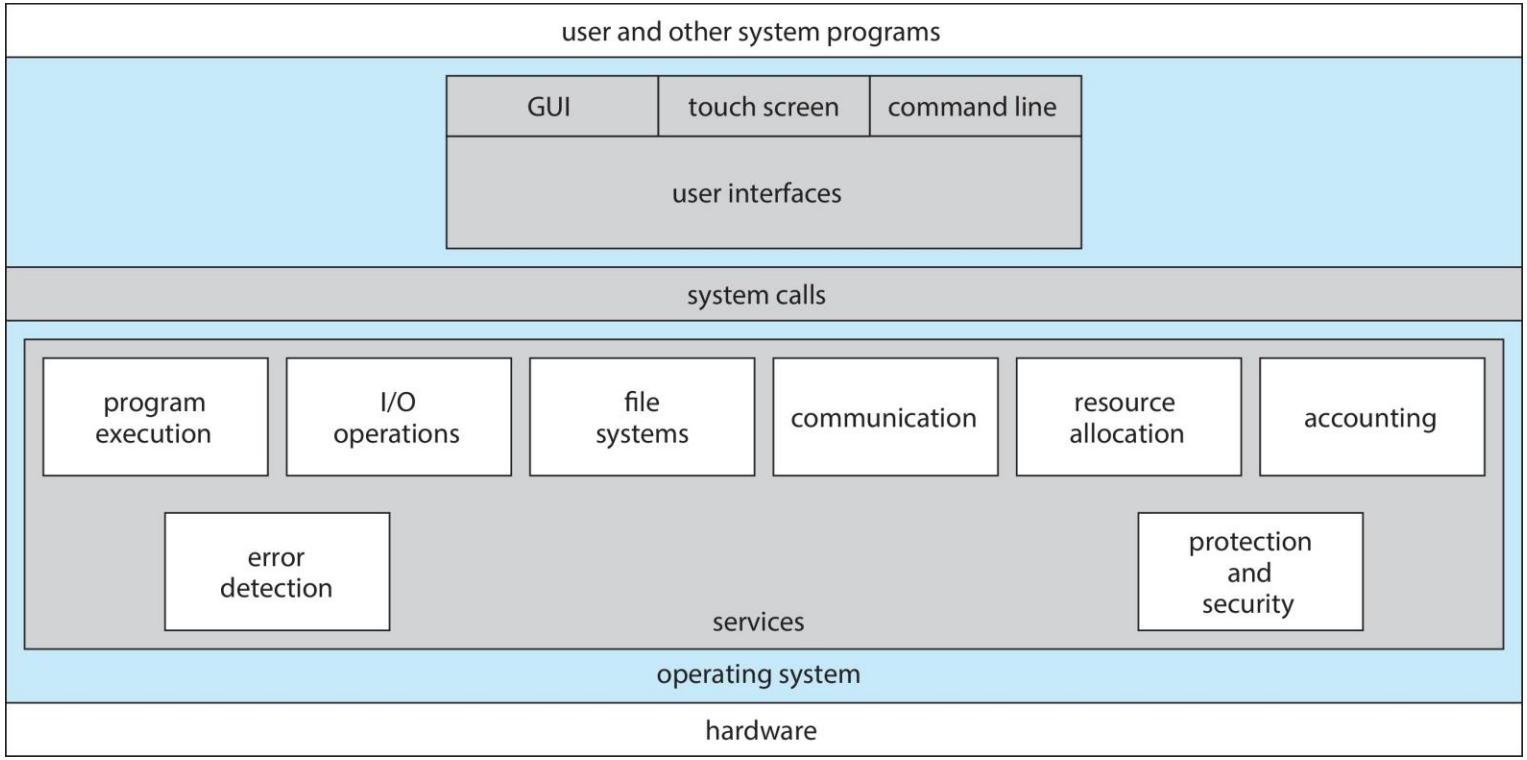
# A View of Operating System Services



# Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - **Error detection** – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

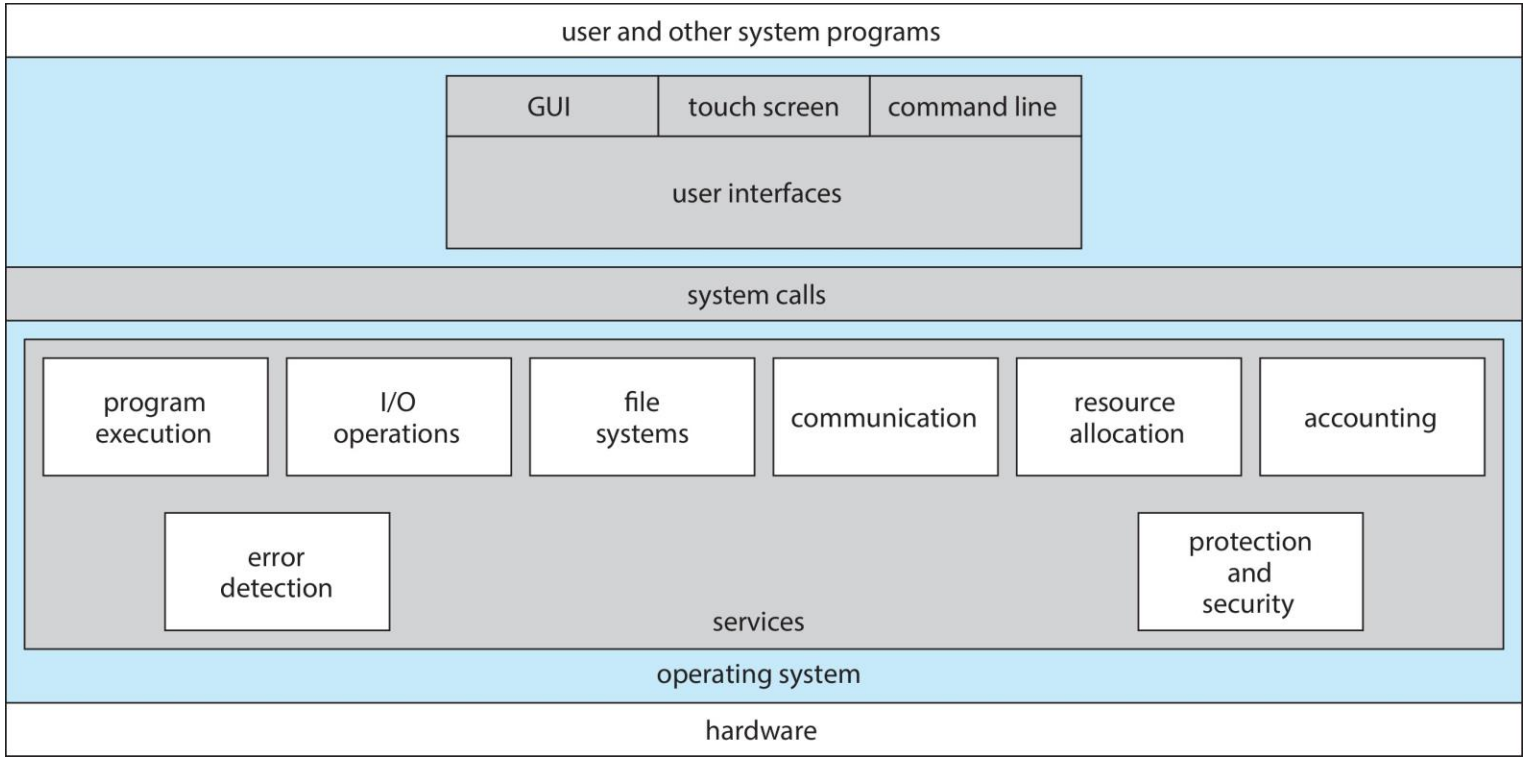
# A View of Operating System Services



# Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Logging -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

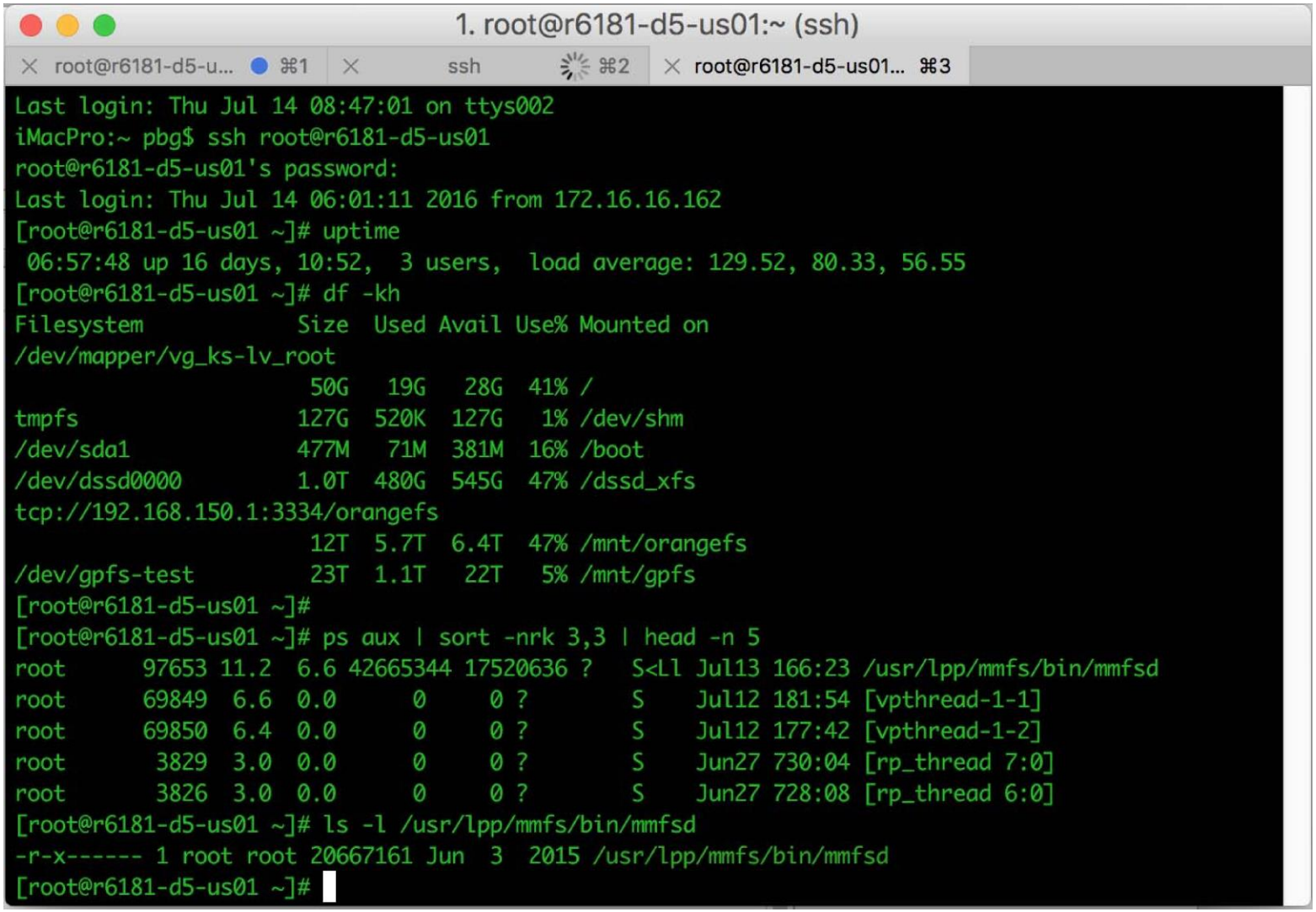
# A View of Operating System Services



# Command Line interpreter

- CLI allows direct command entry
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification

# Bourne Shell Command Interpreter

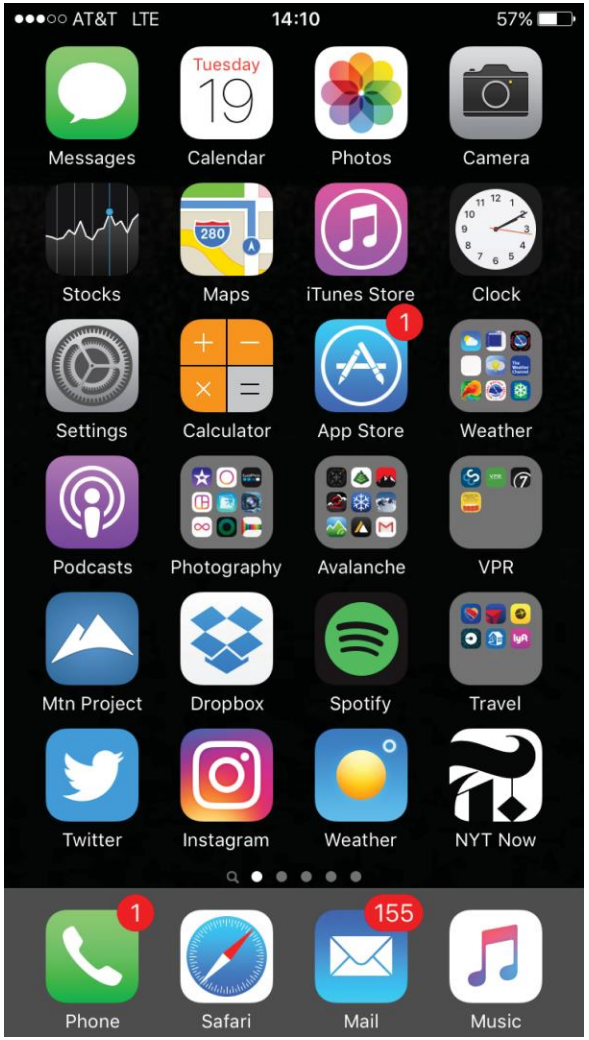


# User Operating System Interface - GUI

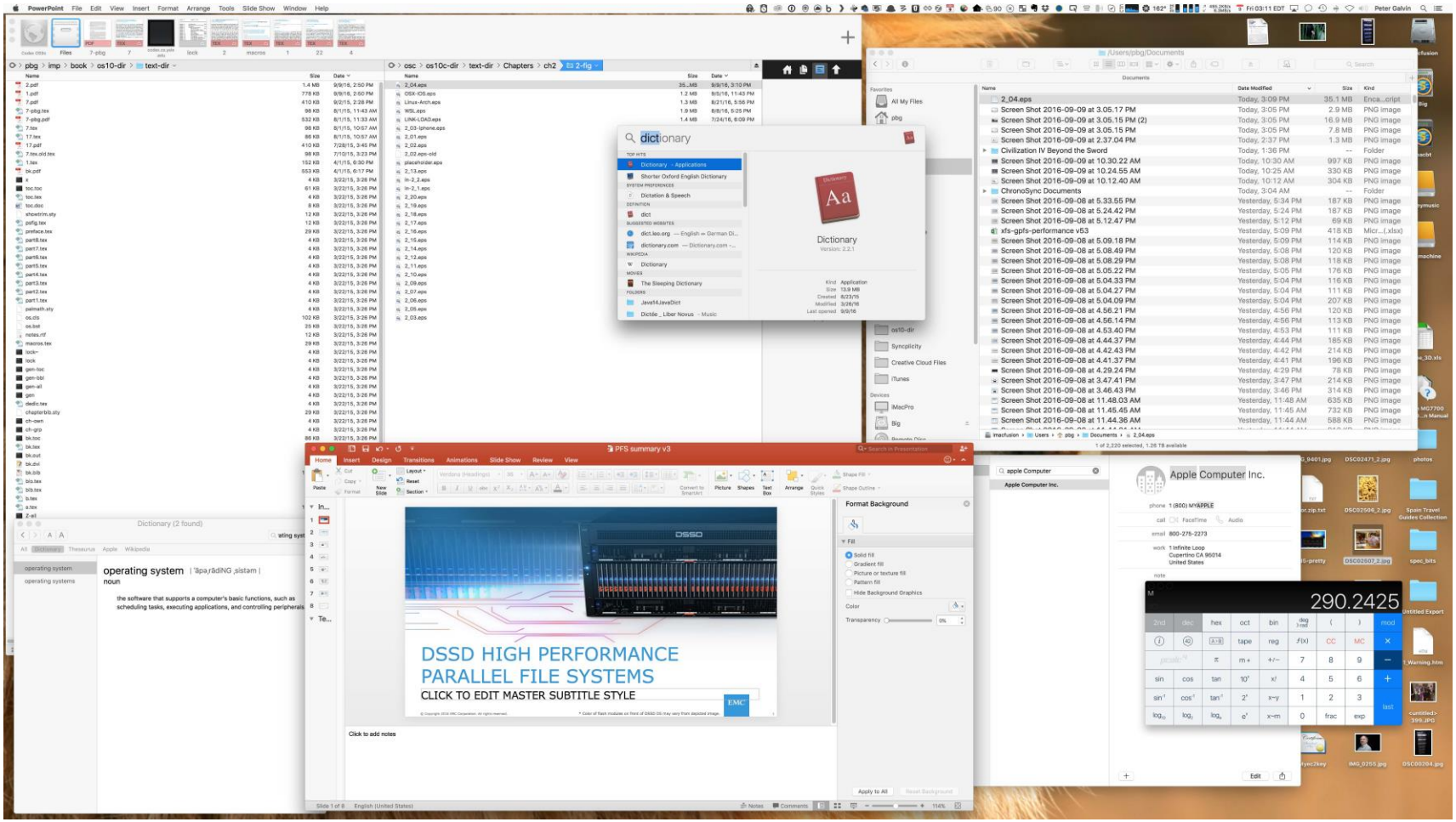
- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

# Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands



# The Mac OS X GUI



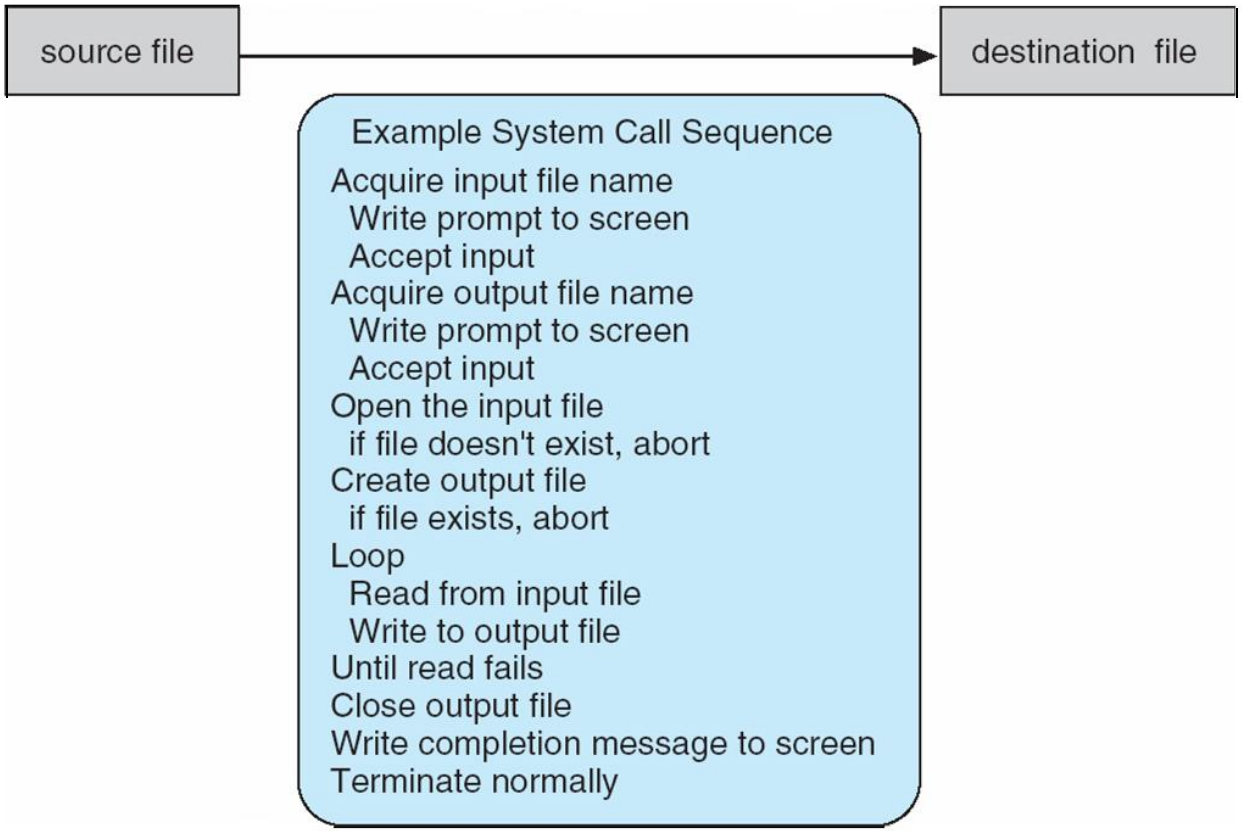
# System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
- Three most common APIs are
  1. Win32 API for Windows,
  2. POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and
  3. Java API for the Java virtual machine (JVM)

Note เราจะมองการใช้ system call จากมุมมองของ Process ที่เรียกใช้ก่อนแล้วจะพิจารณา implementation

# Example of System Calls

- System call sequence to copy the contents of one file to another file



# Example of Standard API

*EXAMPLE OF STANDARD API*

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

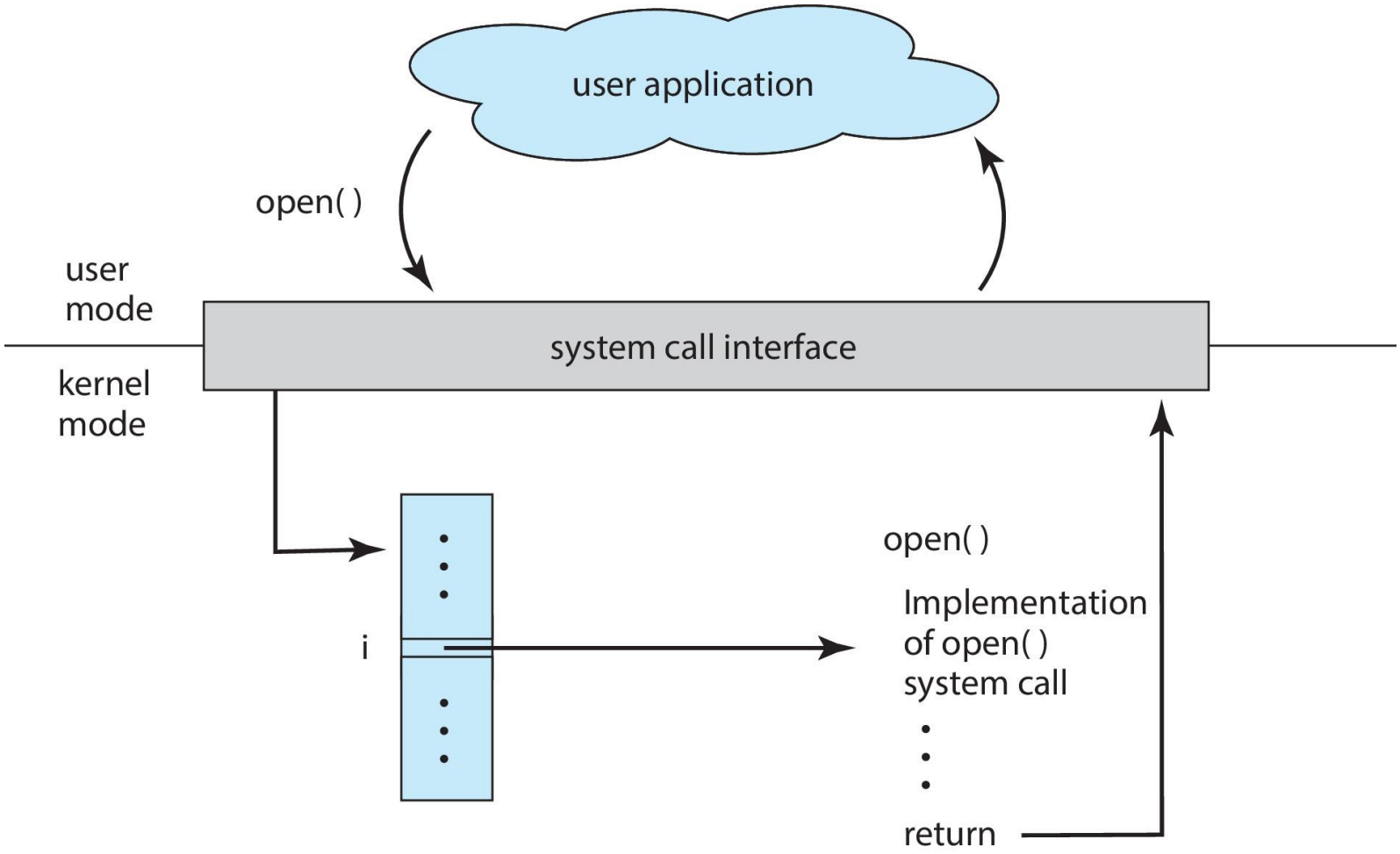
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

# System Call Implementation

- Typically, a number is associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

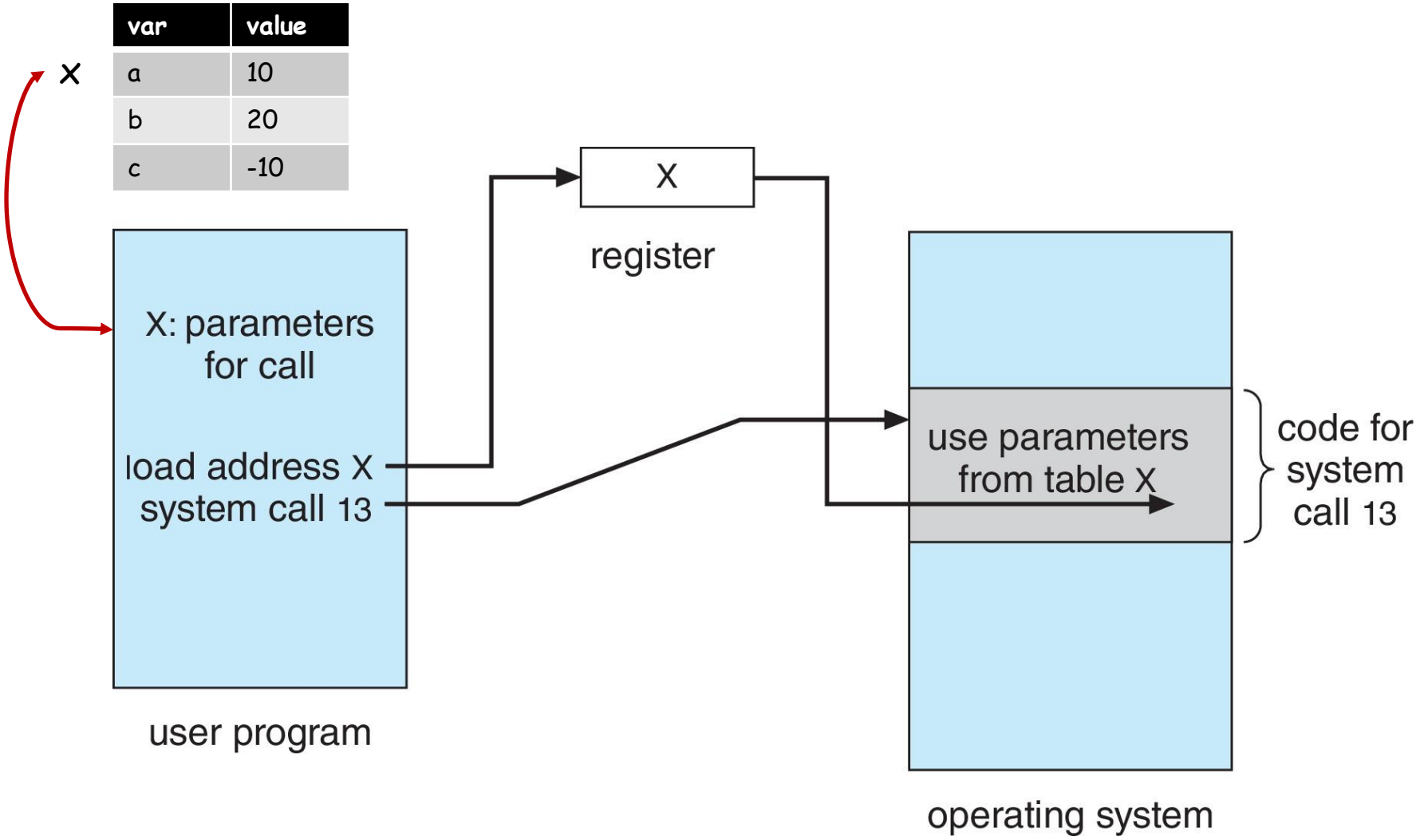
# API – System Call – OS Relationship



# System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest:  pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

# Parameter Passing via Table

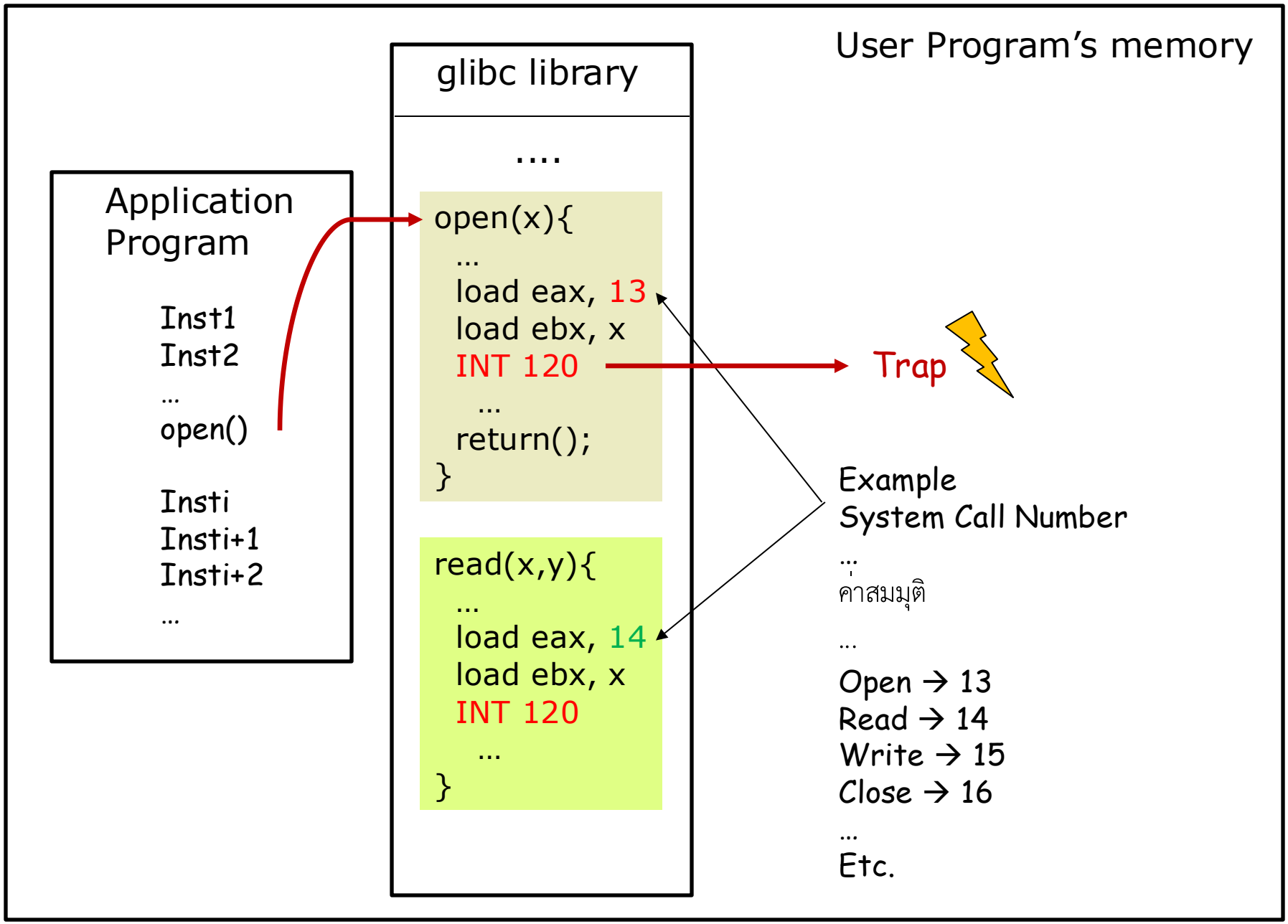


# System Call เปลี่ยนการประมวลผลจากชุดคำสั่งของผู้ใช้ใน User Programs ไปรันชุดคำสั่งของ Kernel ได้อย่างไร

- กระบวนการดังกล่าวใน OS เช่น Linux มีหลายวิธี (ดังในบทที่ 2 ของ Textbook จะ 2)
- ในระบบ Linux มีหลายวิธี แต่จะกล่างถึงสองวิธีใหญ่ ๆ
- **วิธีที่ 1: วิธีการดั้งเดิมเรียกว่า Legacy System Call**
- **วิธีที่ 2: วิธีการ Fast System Call**
- อ่านรายละเอียดเพิ่มเติมได้ที่
- https://blog.packagecloud.io/the-definitive-guide-to-linux-system-calls/#:~:text=The%20Linux%20kernel%20sets%20aside,code%20that%20actually%20does%20this

# Legacy System Call

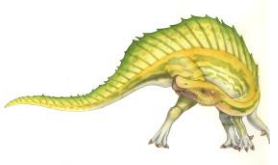
- ใช้การ Trap โดยมีตัวเลข Interrupt Number คือ 120 (0x80)
- Linux Kernel จะกำหนดค่า interrupt vector และ interrupt handler สำหรับจัดการ system call ไว้ตั้งแต่ OS เริ่มต้นการทำงาน
- ยกตัวอย่างเช่น ใน Linux Kernel 3.13 กำหนดใน Interrupt Vector ว่า เมื่อใดถ้ามีสัญญาณ Interrupt หมายเลข 120 ซีพียูก็จะไปรัน Interrupt Handler ชื่อว่า ia32_syscall ซึ่งเป็น รูทีนที่ใช้จัดการ System call ทั้งหมดซึ่ง ถูกเก็บในโค๊ดของ OS kernel ที่ฝังตัวอยู่ในหน่วยความจำของ OS kernel
- การผ่าน พารามิเตอร์ Linux จะใช้ Library ชื่อ glibc ที่ให้บริการ API (ภาษา C) สำหรับเรียกใช้ System Call

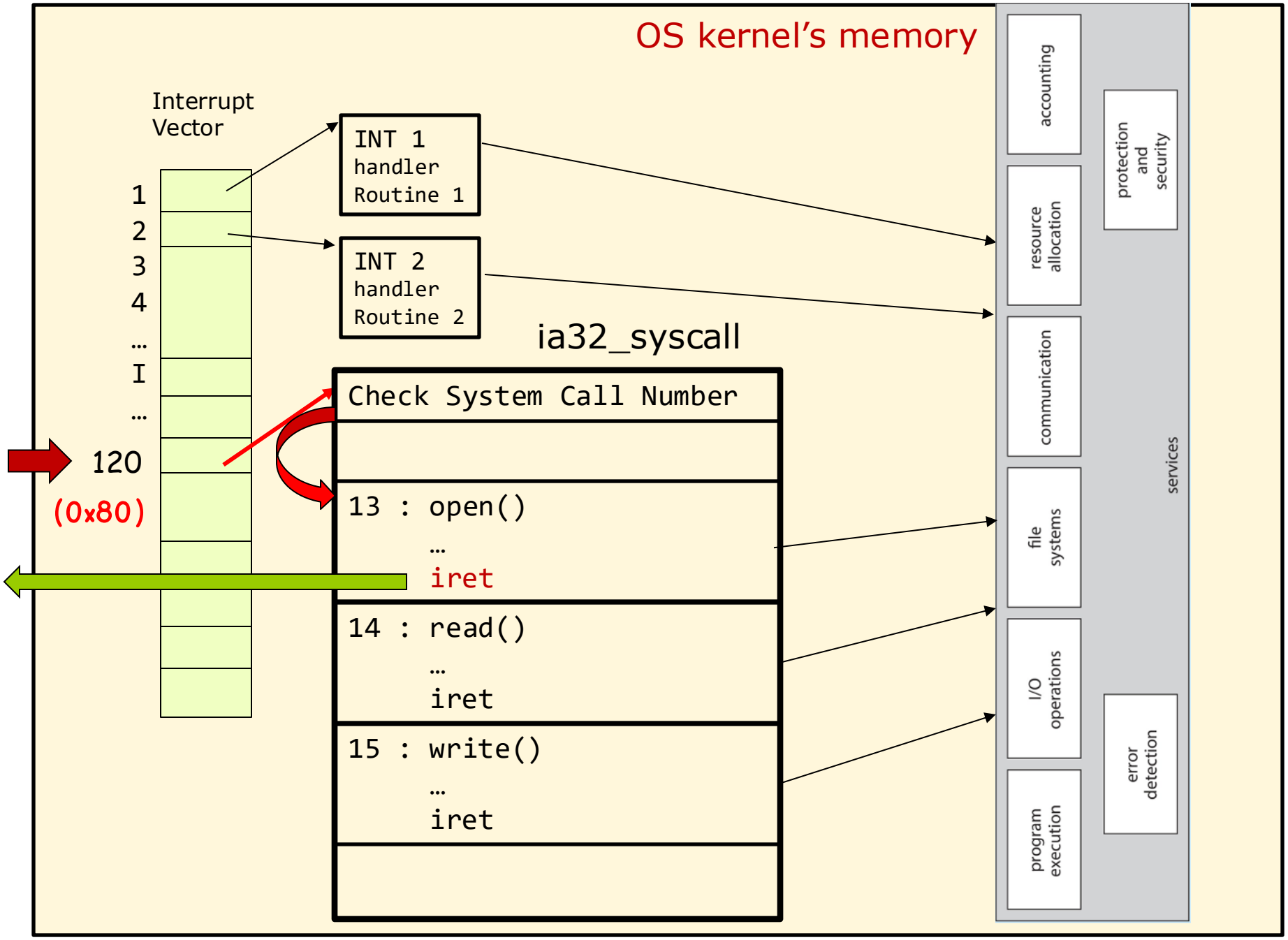

User Program's memory
glibc library
....
Application Program
Inst1
Inst2
…
open()
Insti
Insti+1
Insti+2
…
open(x){
…
load eax, 13
load ebx, x
INT 120
…
return();
}
read(x,y){
…
load eax, 14
load ebx, x
INT 120
…
}
Trap
Example
System Call Number
…
คาสมมุติ
…
Open → 13
Read → 14
Write → 15
Close → 16
…
Etc.

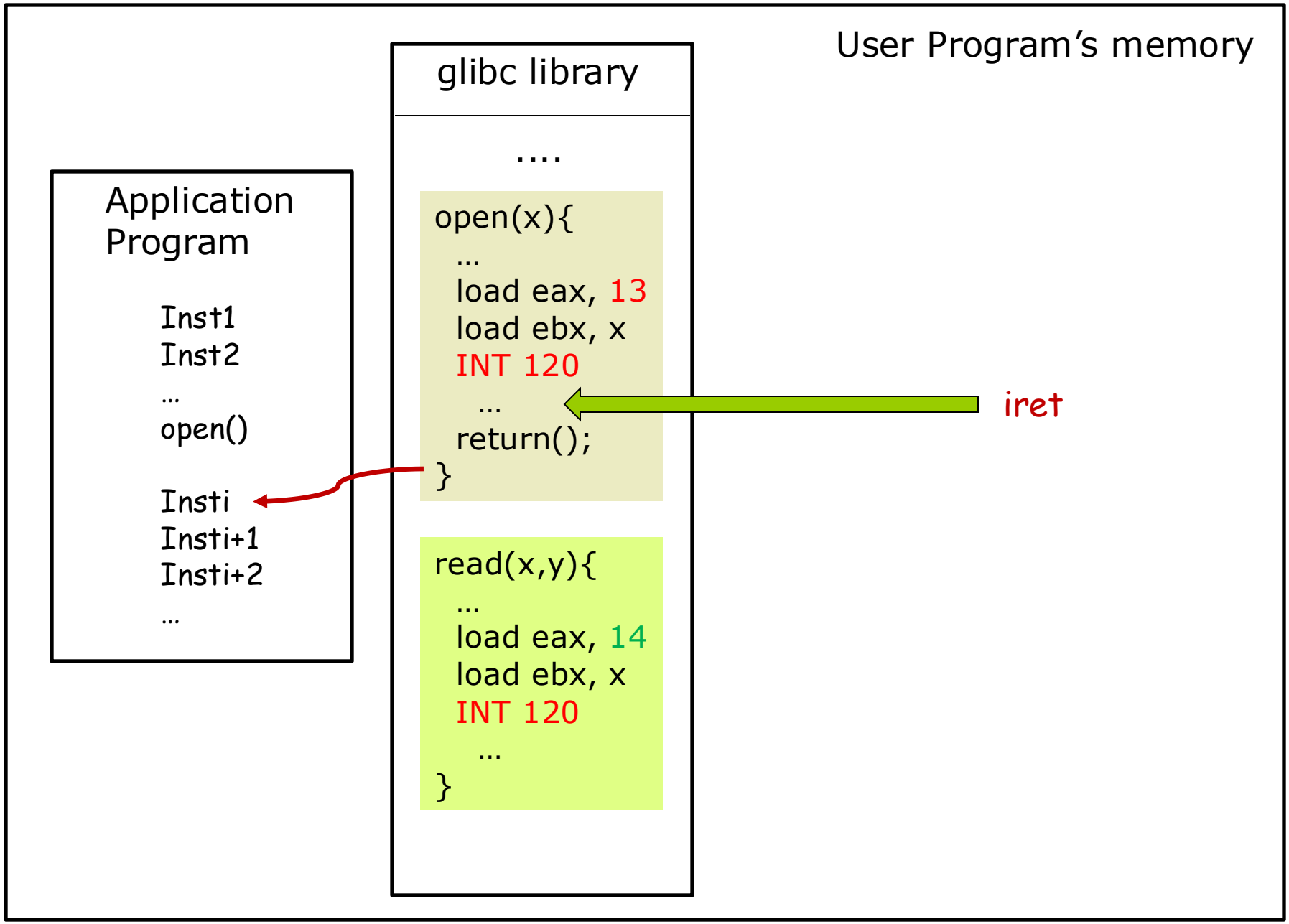
%13

```
asm ("movl %0, %%eax\n"
        "movl %1, %%ebx\n"
     "int $0x80"
   : /* output parameters, we aren't outputting anything, no none */
     /* (none) */
   : /* input parameters mapped to %0 and %1, repsectively */
     "m" (syscall_nr), "m" (exit_status)
   : /* registers that we are "clobbering", unneeded since we are calling exit */
     "eax", "ebx");
```

https://blog.packagecloud.io/the-definitive-guide-to-linux-system-calls/#:~:text=The%20Linux%20kernel%20sets%20aside,code%20that%20actually%20does%20this




OS kernel's memory
Interrupt Vector
1
2
3
4
…
I
…
120
(0x80)
INT 1 handler Routine 1
INT 2 handler Routine 2
ia32_syscall
Check System Call Number
13 : open()
…
iret
14 : read()
…
iret
15 : write()
…
iret
accounting
protection and security
resource allocation
communication
file systems
I/O operations
program execution
error detection
services


User Program's memory
glibc library
....
Application Program
Inst1
Inst2
…
open()
Insti
Insti+1
Insti+2
…
open(x){
…
load eax, 13
load ebx, x
INT 120
…
return();
}
read(x,y){
…
load eax, 14
load ebx, x
INT 120
…
}
iret

# int/iret instruction

- คำสั่ง INT เป็นคำสั่ง user-level instruction ที่ user program สามารถเรียกใช้ได้ เพื่อสร้าง software interrupt หรือ trap เพื่อเรียก system call routine ของ kernel ให้ประมวลผล
- รูปแบบการใช้งานคำสั่ง int คือ
  1. INT N // โดยที่ N คือหมายเลข Interrupt Number
  2. โดย Linux จะกำหนดให้ N = 0x80 ในขณะที่ Windows NT ใช้ N = 0x2e
- รูปแบบการใช้ int สำหรับเรียก system call คือ
  1. กำหนดค่าหมายเลข system call number ใน ”eax” register
  2. INT 0x80 // (0x80 มีค่าเท่ากับ 128)
- INT เป็นคำสั่งที่ user program สามารถเรียกใช้ได้ในขณะที่ซีพียูอยู่ใน user mode **แต่** คำสั่งนี้จะ*ทำให้ซีพียูเปลี่ยนโหมดการประมวลผลเป็น system mode* แล้ว เรียก routine ใน system call table ของ kernel มาประมวลผล

# int/iret instruction

- ก่อนเรียก INT ผู้ใช้ต้องกำนดค่า system call number ใน register **eax** ใน x86
- เมื่อซีพียูรันคำสั่ง “INT 0x80” จะเกิดสัญญาณ Trap หมายเลข 120 ทำให้
  1. ซีพียู copy ค่า register context ทั้งหมดของซีพียู (CPU context) รวมทั้งรีจิสเตอร์ที่เก็บค่า Mode (ซึ่งเป็น User Mode ในตัวอย่างนี้) ไปไว้ในพื้นที่หน่วยความจำของ kernel
  2. ซีพียูเปลี่ยนโหมดเป็น system mode แล้วรัน routine ใน **system call table[eax] ของ kernel (ใช้ค่า ใน eax เป็น index)**
  3. สุดท้าย routine เรียกคำสั่ง **iret** เพื่อ restore CPU context และส่งคืนกลับการประมวล (รวมทั้งเปลี่ยนซีพียูเป็น user mode) ให้ user program ที่ int ปร

# ข้อเสียของ int/iret instruction

- ช้า เนื่องจากต้องเข้าถึง Interrupt Vector Table ก่อน แล้วจึงเข้าถึง system call table
- ยากที่โปรแกรมเมอร์จะอ่านทำความเข้าใจ ปรับปรุง และตรวจสอบ OS เนื่องจากแต่ละใช้ Interrupt vector number สำหรับ system call ที่แตกต่างกัน (ตย. Linux ใช้ 0x80 แต่ Windows NT ใช้ 0x2e เป็นต้น) (ไม่ portable)
- ดูแลรักษาความปลอดภัยได้ยาก เพราะ
  - ผู้โจมตี (Attacker) สามารถแอบสร้าง interrupt vector ที่ไม่ดีไว้ที่ vector number หนึ่งแล้วใช้ Malicious user program ของตนเรียกด้วย INT
  - ผู้โจมตีจงใจเรียก INT N โดยที่ N มากกว่าขนาดของ interrupt vector table แล้วทำให้เกิด buffer overflow

# int/iret instruction

- แหล่งข้อมูลอ้างอิง

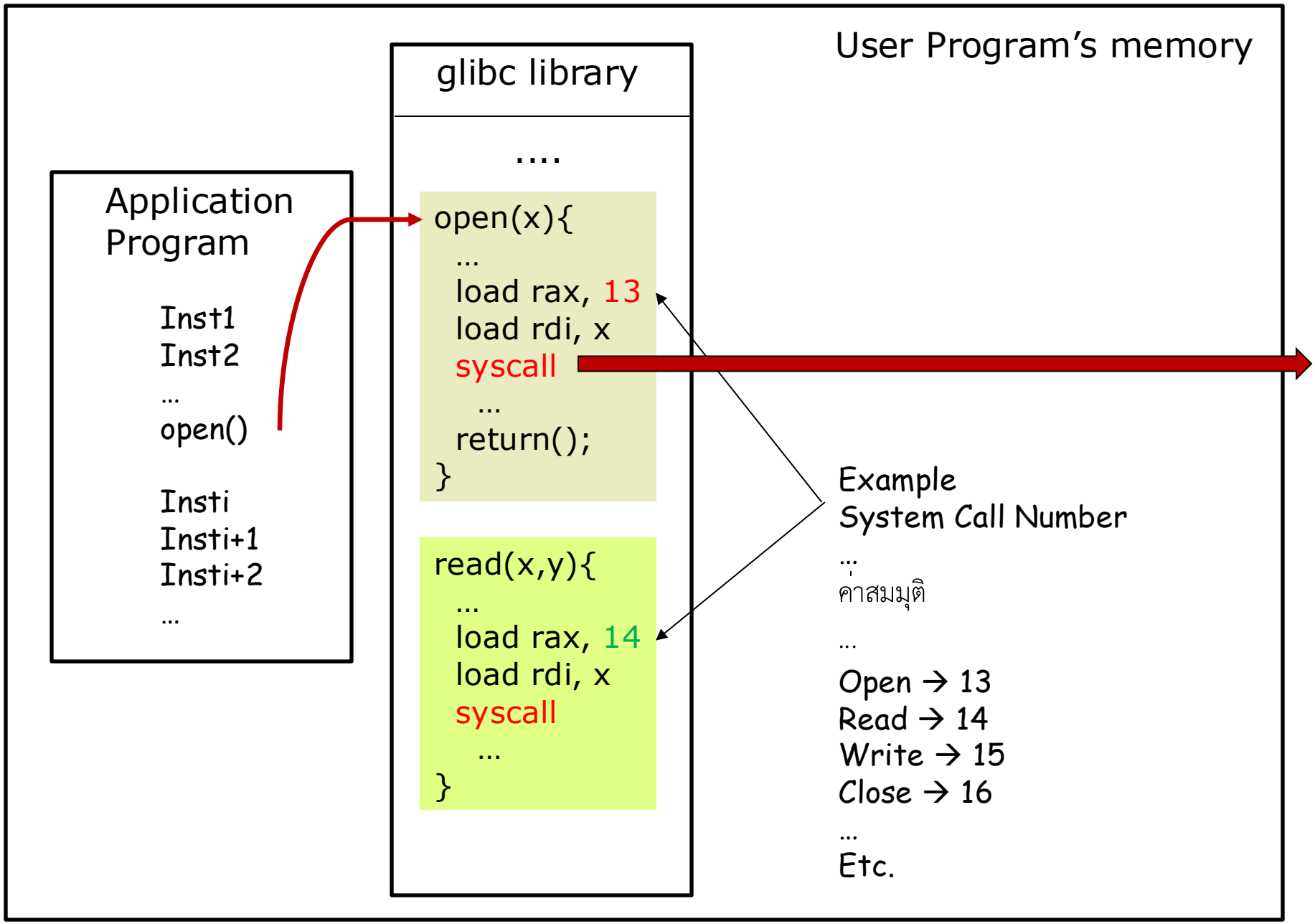
1. https://blog.packagecloud.io/the-definitive-guide-to-linux-system-calls/
2. https://en.wikipedia.org/wiki/INT_(x86_instruction)
3. https://stackoverflow.com/questions/10457296/int-instruction-from-user-space
4. https://www.cs.montana.edu/courses/spring2005/518/Hypertextbook/vijay/project.ppt

# Fast System Call

- ใช้คำสั่งพิเศษ ของ ISA ที่ซีพียูเตรียมไว้ให้ชื่อ “syscall” ที่ User program ธรรมดา (รันเมื่อซีพียูประมวลผลใน user mode) สามารถเรียกใช้ได้
- ผู้ใช้คำสั่งนี้สามารถกำหนดค่า system call number เป็น parameter ให้ system call
- OS เช่น Linux จะต้องกำหนดค่าให้ hardware รู้ไว้ตั้งแค่เริ่มต้นว่าถ้ามีการเรียกใช้คำสั่ง syscall แล้ว ซีพียูจะต้องไปรันชุดคำสั่งสำหรับจัดการ System Call ของ kernel ที่ใด
- เมื่อซีพียูเรียก syscall จะทำให้เกิดการเปลี่ยนโหมดของซีพียูเป็น kernel mode โดยอัตโนมัติ แล้วซีพียูจะไปรันชุดคำสั่งสำหรับจัดการ system call ที่กำหนดไว้
- การผ่าน พารามิเตอร์ Linux จะใช้ Library ชื่อ glibc ที่ให้บริการ API (ภาษา C) สำหรับเรียกใช้ System Call

# Fast System Call

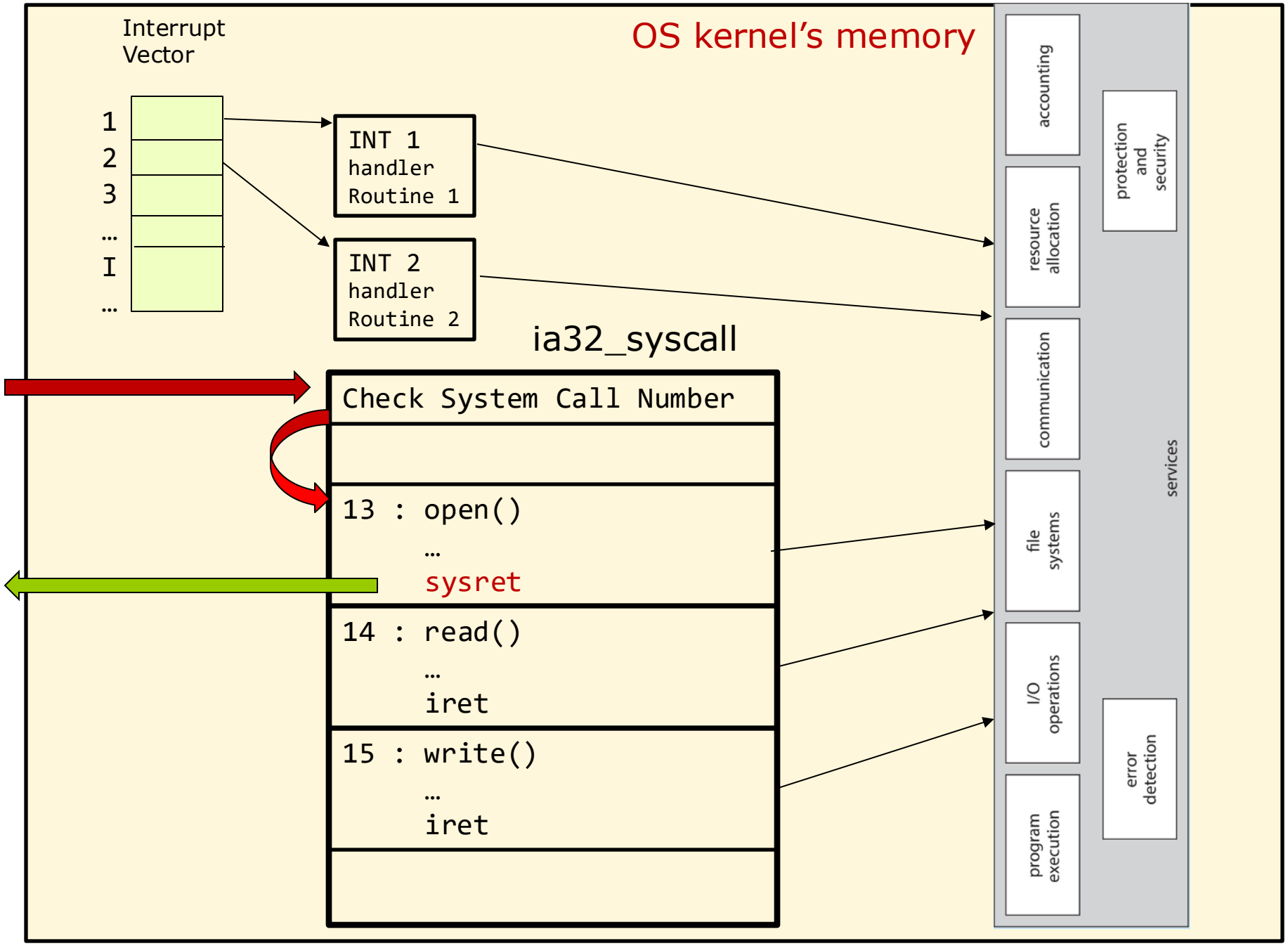
- วิธี Fast System Call มีประสิทธภ Legacy System Call เนื่องจากทุกครั้งที่มีการเรียก system call ไม่เสียเวลาเข้าถึงข้อมูลใน Interrupt Vector เพื่อหาตำแหน่งของ Handler สำหรับจัดการ System Call ในหน่วยความจำของ Kernel
- ในวิธี Fast System call ซีพียูจะรู้ทันทีเมื่อ User Program รันคำสั่ง “syscall” ว่าจะต้องไปรันชุดคำสั่งของ Handler สำหรับจัดการ System Call ที่ตำแหน่งหน่วยความจำของ kernel ใดเข้ามาประมวลผล

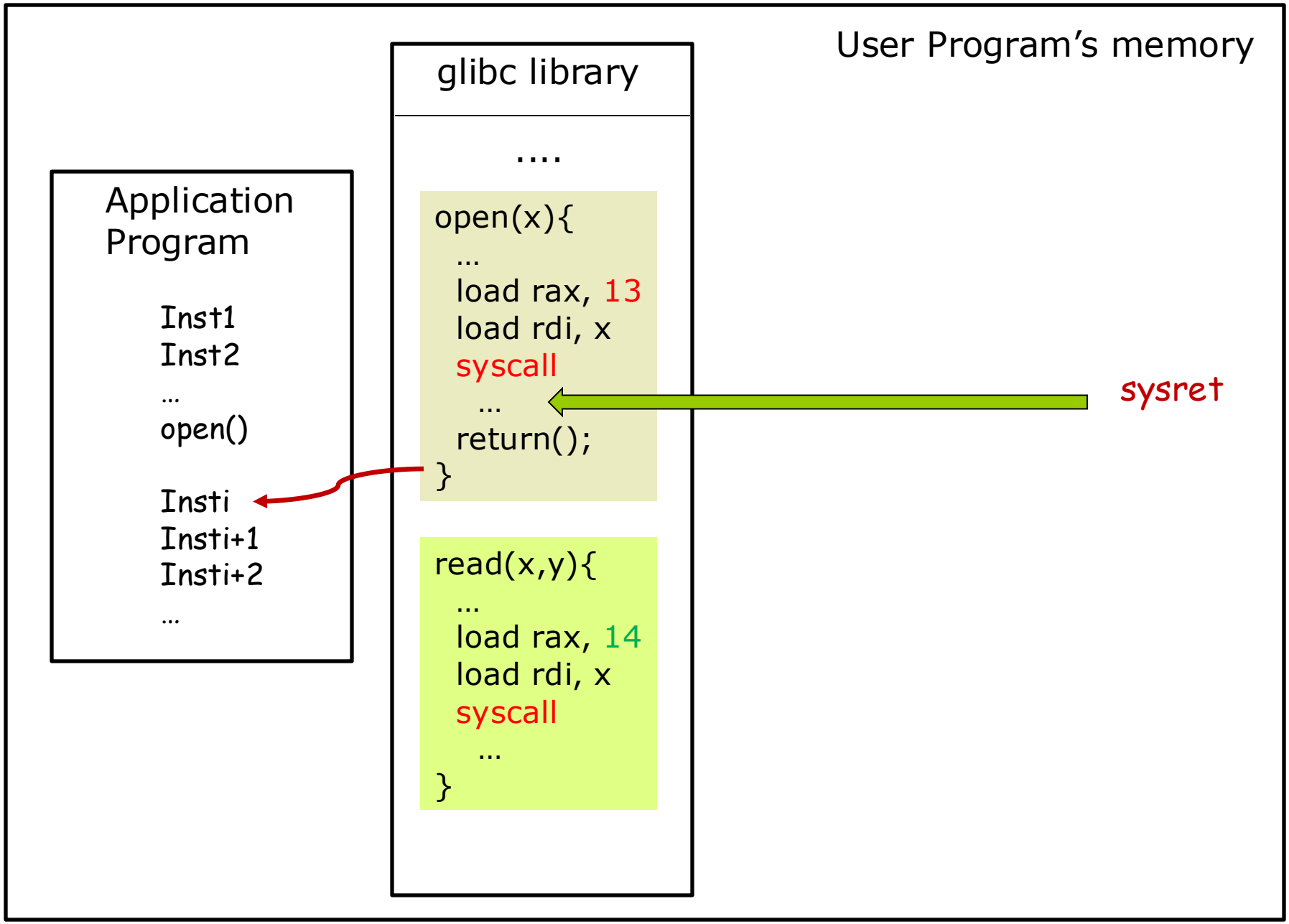

User Program's memory
glibc library
....
Application Program
Inst1
Inst2
…
open()
Insti
Insti+1
Insti+2
…
open(x){
…
load rax, 13
load rdi, x
syscall
…
return();
}
read(x,y){
…
load rax, 14
load rdi, x
syscall
…
}
Example
System Call Number
…
คาสมมุติ
…
Open → 13
Read → 14
Write → 15
Close → 16
…
Etc.

%13

```
asm ("movq %0, %%rax\n"
     "movq %1, %%rdi\n"
     "syscall"
   : /* output parameters, we aren't outputting anything, no none */
     /* (none) */
   : /* input parameters mapped to %0 and %1, repsectively */
     "m" (syscall_nr), "m" (exit_status)
   : /* registers that we are "clobbering", unneeded since we are calling exit */
     "rax", "rdi");
}
```

https://blog.packagecloud.io/the-definitive-guide-to-linux-system-calls/#:~:text=The%20Linux%20kernel%20sets%20aside,code%20that%20actually%20does%20this


OS kernel's memory
Interrupt Vector
1
2
3
…
I
…
INT 1 handler Routine 1
INT 2 handler Routine 2
ia32_syscall
Check System Call Number
13 : open()
…
sysret
14 : read()
…
iret
15 : write()
…
iret
accounting
resource allocation
communication
file systems
I/O operations
program execution
protection and security
error detection
services


User Program's memory
glibc library
....
open(x){
…
load rax, 13
load rdi, x
syscall
…
return();
}
read(x,y){
…
load rax, 14
load rdi, x
syscall
…
}
Application Program
Inst1
Inst2
…
open()
Insti
Insti+1
Insti+2
…
sysret

# syscall/sysret instruction

- คำสั่ง syscall เป็นคำสั่ง user-level instruction ที่ user program สามารถเรียกใช้ได้เพื่อสร้าง software interrupt หรือ trap เพื่อเรียก system call routine ของ kernel ให้ประมวลผล
- รูปแบบการใช้คำสั่ง syscall คือ
  1. กำหนดค่าหมายเลข system call number ใน "rax" register
  2. syscall
- syscall เป็นคำสั่งที่ user program เรียกใช้ได้ใน user mode แต่ คำสั่งนี้จะทำให้ซีพียูเปลี่ยนโหมดการประมวลผลเป็น system mode แล้ว เรียก routine ใน system call table ของ kernel มาประมวลผล
- *เร็วกว่า INT และทำให้ดูแลรักษาโค้ดของโอเอสได้ง่ายกว่า*

ปรับ hardware
เพื่อ Software

# syscall/sysret instruction

- ก่อนเรียก syscall ผู้ใช้ต้องกำนดค่า system call number ใน register rax ใน x86
- เมื่อซีพียูประมวลผล syscall จะเกิดสัญญาณ trap ทำให้

1. ซีพียู copy ค่า register context ทั้งหมดของซีพียู (CPU context) รวมทั้งรีจิสเตอร์ที่เก็บค่า Mode (ซึ่งเป็น User Mode ในตัวอย่างนี้) ไปไว้ในพื้นที่หน่วยความจำของ kernel
2. ซีพียูเปลี่ยนโหมดเป็น system mode แล้วไปรัน system call routine ใน system call table[rax] ของ kernel (ใช้ค่า ใน rax เป็น index)
3. สุดท้าย routine เรียกคำสั่ง **sysret** เมื่อมันจบการประมวลผลเพื่อคืนกลับการประมวล (รวมทั้งเปลี่ยนซีพียูเป็น user mode) ให้ user program ที่เรียก syscall ป ร

# syscall/sysret instruction

- แหล่งข้อมูลอ้างอิง
  - https://blog.packagecloud.io/the-definitive-guide-to-linux-system-calls/

## คำถาม

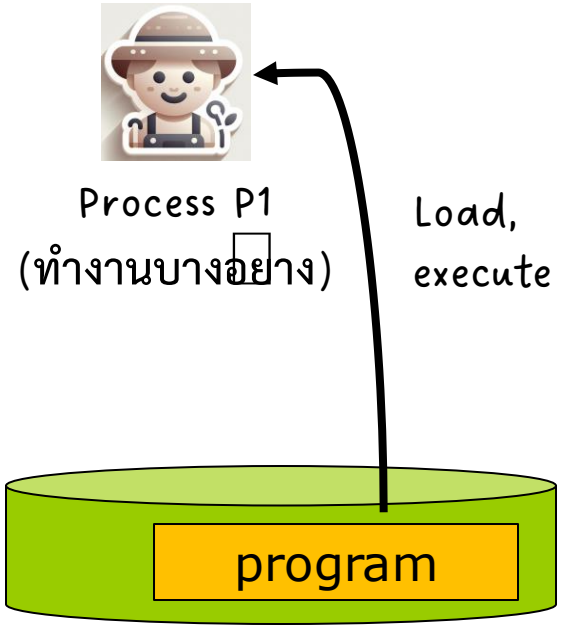
- ในการเรียก System Call หนึ่งครั้งจาก User โปรแกรม แล้วการทำงาน return กลับไปที่ user program เลย (เช่น getpid() ใน Linux) ดังตัวอย่างที่ผ่านมา เกิด CPU Context Switching ขึ้นกี่ครั้ง?

# การรีเทิร์นจาก sys call ขึ้นอยู่กับชนิดของ handler

- ถ้า system call ใช้เวลาน้อย (เช่นไม่ต้องรอ I/O เป็นเวลานาน) OS ก็จะทำงานให้เสร็จ และ return การมำงานให้ process ที่เรียกใช้ system call ทำงานต่อ
- แต่ถ้า system call ใช้วลามาก OS (เช่น block รอ I/O เป็นเวลานาน) OS ก็จะนำ Process ใหม่เข้ามาใช้งานซีพียู แล้วค่อยกลับไปรัน process ที่เรียกใช้ system call ในเวลาที่เหมาะสม
- เปรียบเหมือนเวลาพนักงาน ร้านสะดวกซื้อ เอาของไปเวฟให้เรา ถ้านานเขาก็จะให้เรารอ แล้วคิดเงินให้ลูกค้าคนอื่นไปก่อน

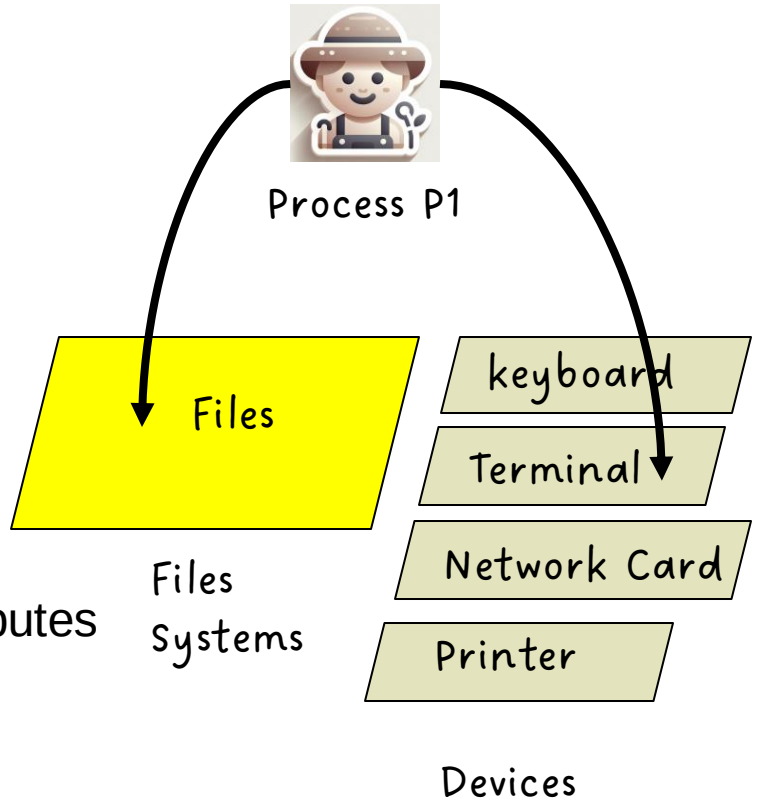
# Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes



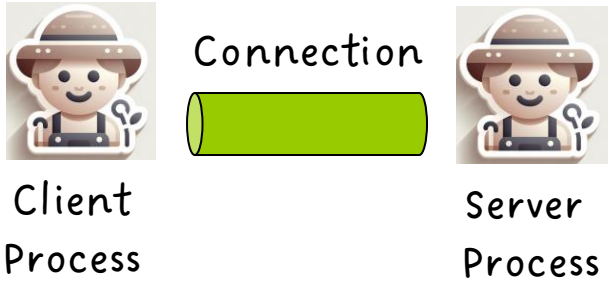
# Types of System Calls (Cont.)

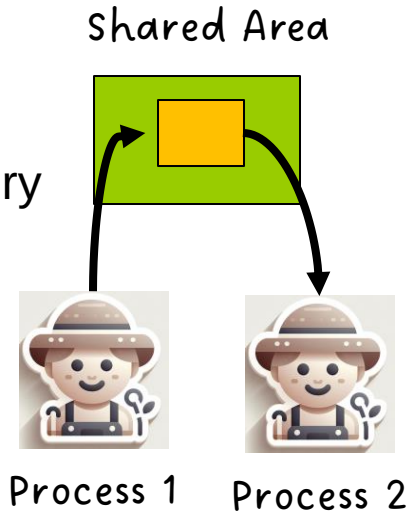
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices



# Types of System Calls (Cont.)

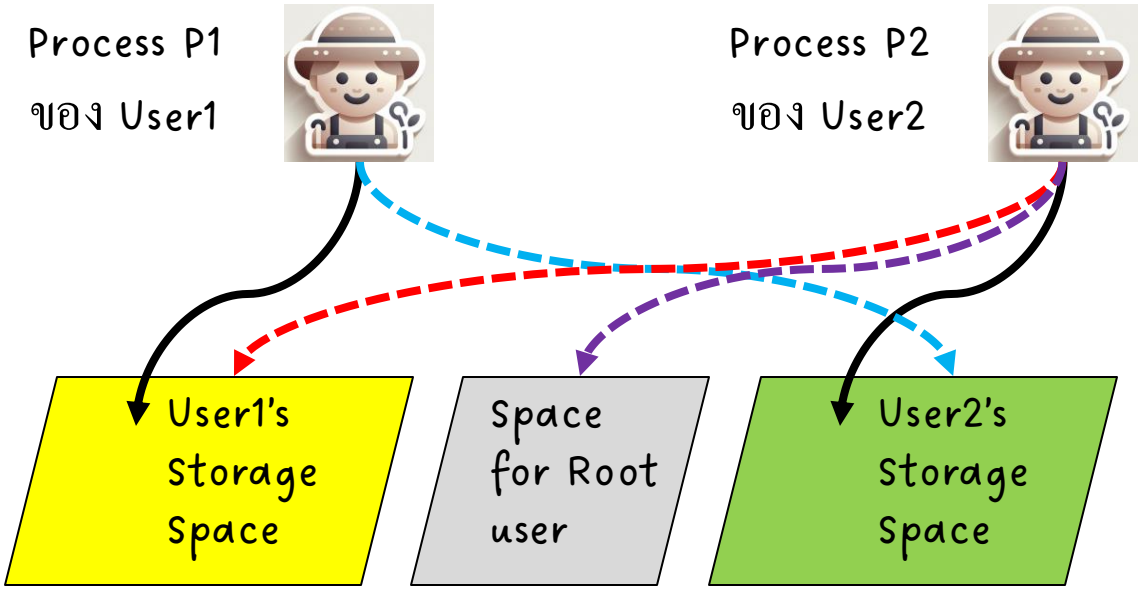
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices



# Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access



# Examples of Windows and Unix System Calls

*EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS*

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

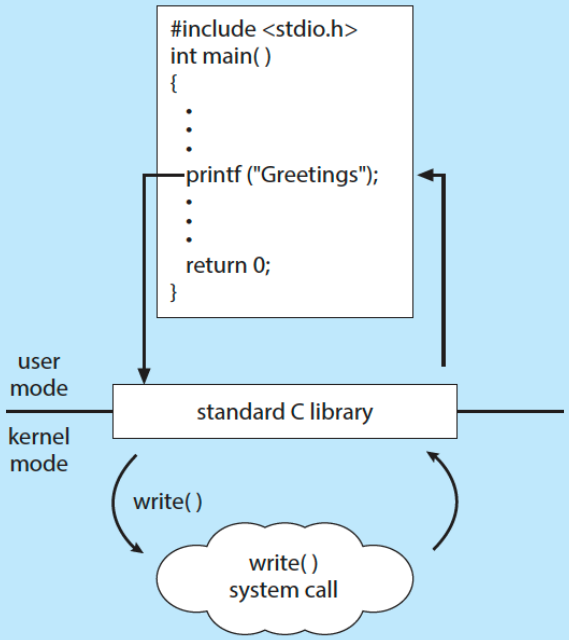
| | Windows | Unix |
|---|---|---|
| **Process control** | `CreateProcess()` | `fork()` |
| | `ExitProcess()` | `exit()` |
| | `WaitForSingleObject()` | `wait()` |
| **File management** | `CreateFile()` | `open()` |
| | `ReadFile()` | `read()` |
| | `WriteFile()` | `write()` |
| | `CloseHandle()` | `close()` |
| **Device management** | `SetConsoleMode()` | `ioctl()` |
| | `ReadConsole()` | `read()` |
| | `WriteConsole()` | `write()` |
| **Information maintenance** | `GetCurrentProcessID()` | `getpid()` |
| | `SetTimer()` | `alarm()` |
| | `Sleep()` | `sleep()` |
| **Communications** | `CreatePipe()` | `pipe()` |
| | `CreateFileMapping()` | `shm_open()` |
| | `MapViewOfFile()` | `mmap()` |
| **Protection** | `SetFileSecurity()` | `chmod()` |
| | `InitlializeSecurityDescriptor()` | `umask()` |
| | `SetSecurityDescriptorGroup()` | `chown()` |

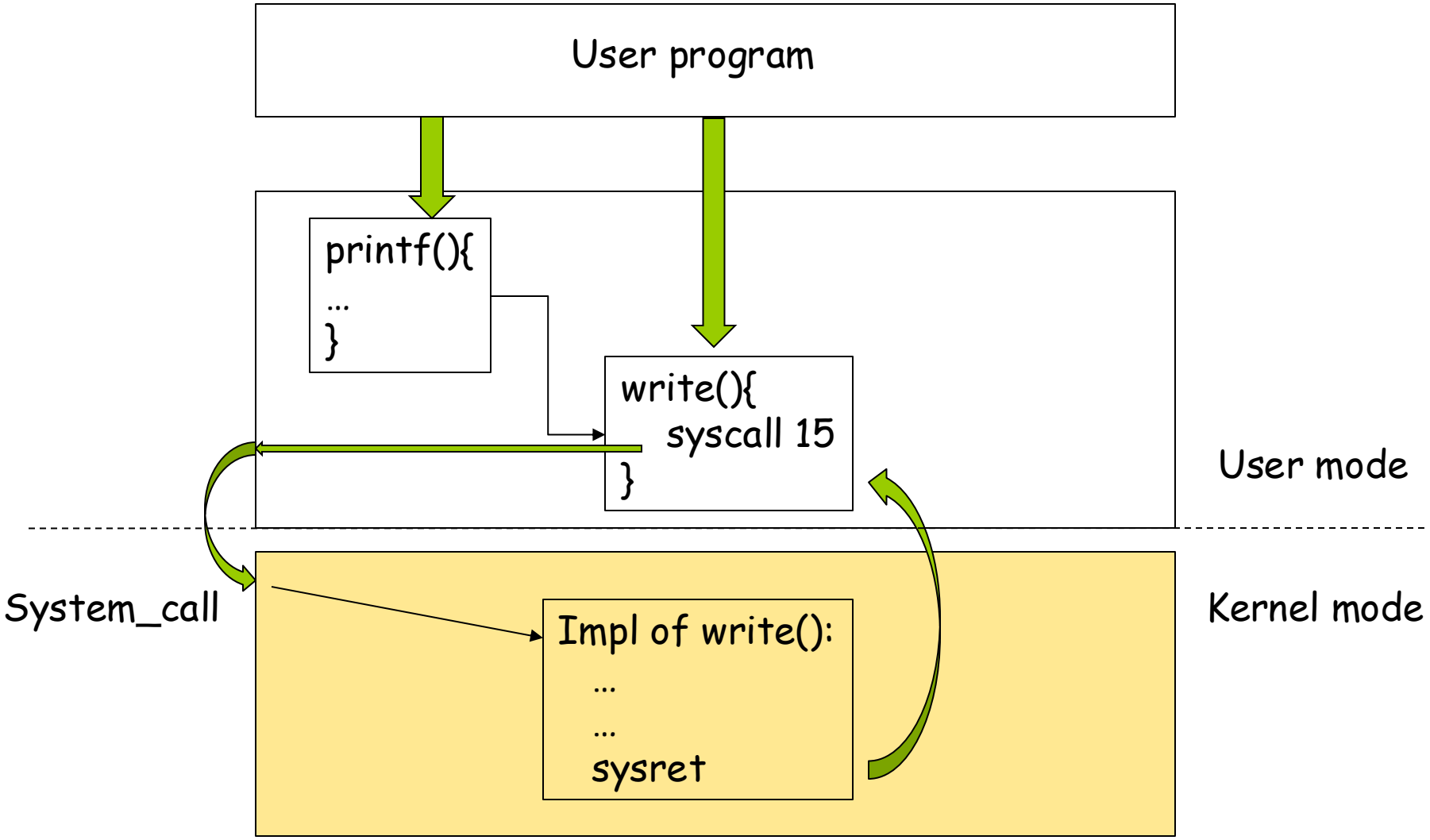
# Standard C Library Example

- C program invoking printf() library call, which calls write() system call

*THE STANDARD C LIBRARY*

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:




User program
printf(){
...
}
write(){
syscall 15
}
User mode
System_call
Impl of write():
...
...
sysret
Kernel mode

# การใช้ strace ดู system call ที่ Process ใช้

- คำสั่ง strace <cmd> เป็นคำสั่งที่แสดงการเรียกใช้ system call ทั้งหมด เมื่อโปรแกรม <cmd> รัน
- $ sudo apt install strace
- $ strace ls

```
kasidit@enterprise:~$ strace ls
execve("/usr/bin/ls", ["ls"], 0x7ffe0f6dd350 /* 32 vars */) = 0
brk(NULL)                                = 0x55d5923e6000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffdda683ce0) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5acb086000
access("/etc/ld.so.preload", R_OK)       = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=99859, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 99859, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5acb06d000
close(3)                                 = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=166280, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 177672, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5acb041000
mprotect(0x7f5acb047000, 139264, PROT_NONE) = 0
mmap(0x7f5acb047000, 106496, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0
mmap(0x7f5acb061000, 28672, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x20000) = 0
mmap(0x7f5acb069000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x
mmap(0x7f5acb06b000, 5640, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0
close(3)                                 = 0
```

# System Services

- โปรแกรมให้บริการของระบบ System Services หรือ System Programs หรือ Utility Programs -- โปรแกรม cp ls chmod ... etc ที่ นศ ใช้ที่ไม่ใช่ App Prog
- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a file
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- Most users' view of the operating system is defined by system programs, not the actual system calls

# System Services (Cont.)

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a **registry** - used to store and retrieve configuration information

# System Services (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

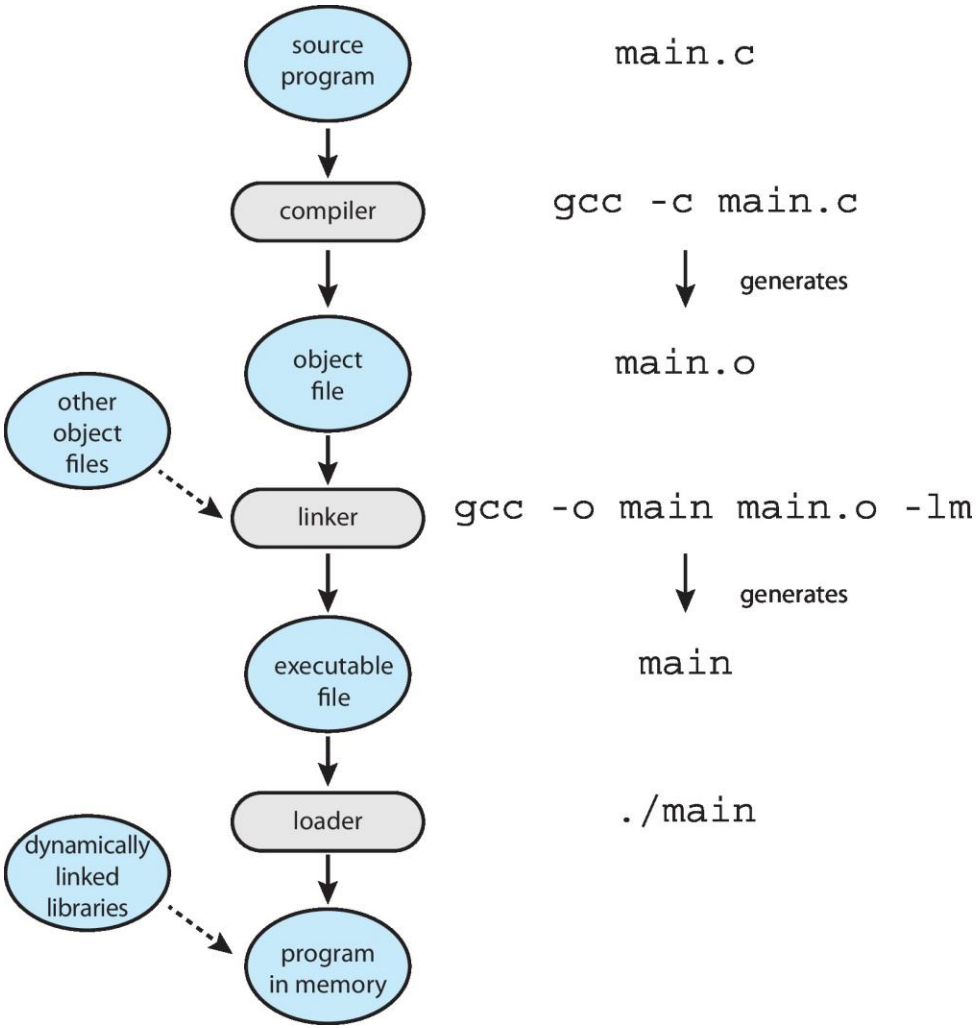
# System Services (Cont.)

- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as **services**, **subsystems**, **daemons**
- **Application programs**
  - Don’t pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke

# Linkers and Loaders

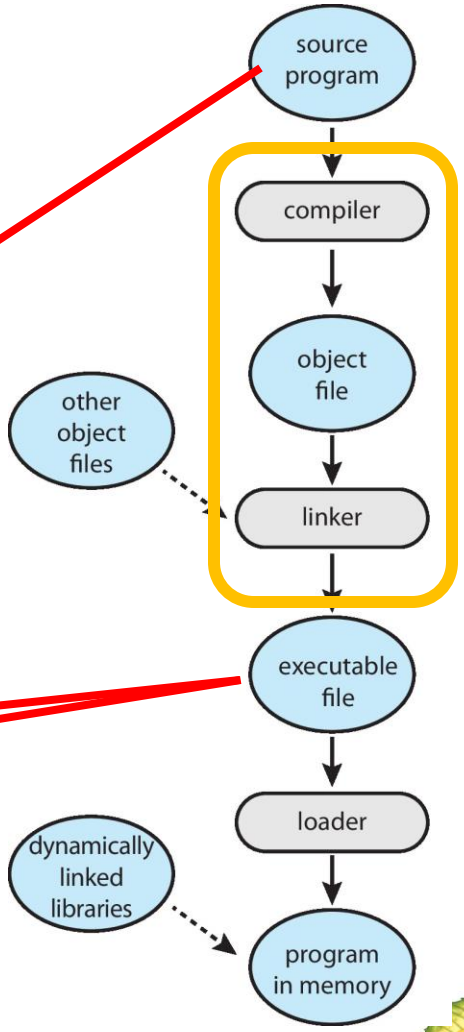
- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
  - Also brings in libraries
- Program resides on secondary storage as binary executable
- Must be brought into memory by **loader** to be executed
  - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses
- Modern general purpose systems don't link libraries into executables
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)
- Object, executable files have standard formats, so operating system knows how to load and start them

# The Role of the Linker and Loader



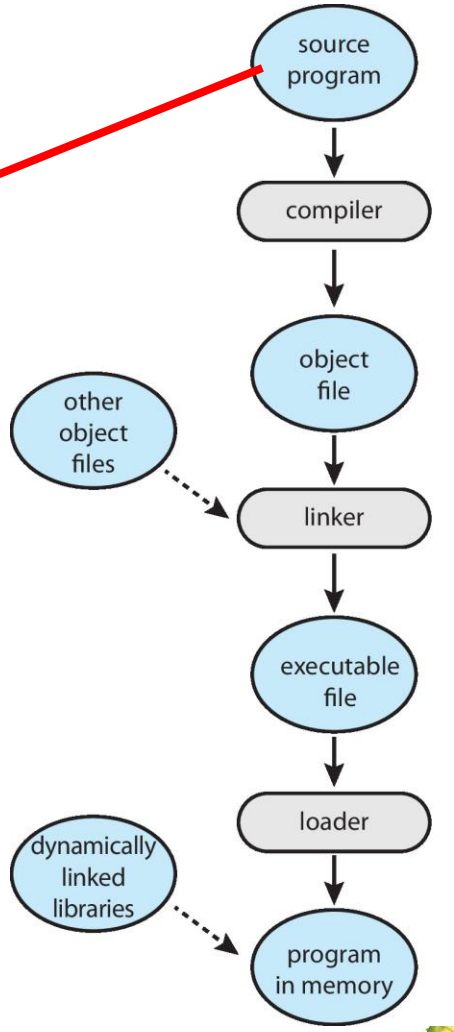
# ติดตั้ง C compiler และ compile C program แบบไฟล์เดียว

```
$ sudo apt install gcc
(or apt install build-essentials)
$ nano hello.c
#include <stdio.h>
main(){
  printf("hello world :)\n");
}
$ gcc hello.c
$ ./a.out
$ gcc -o hello hello.c
$ ./hello
```



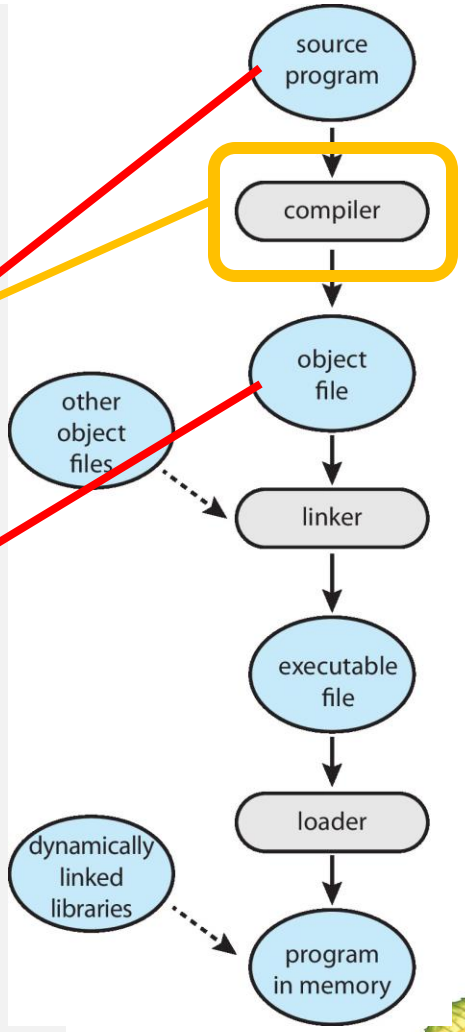

# การคอมไพลแบบแยกไฟล์ object code และ link โปรแกรม

```
$ nano hello1.h
$ cat hello1.h
#define CONSTANT1 100
$
```



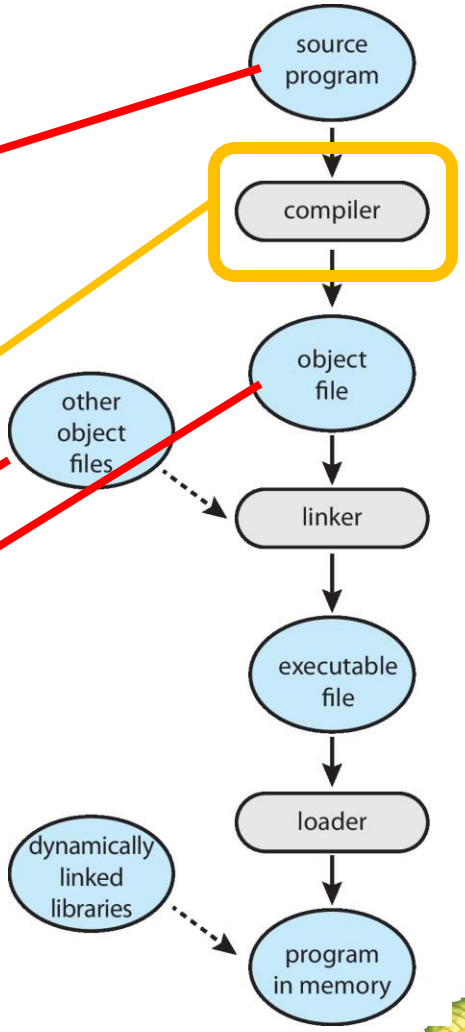

# การคอมไพล object code “hellosub1.o”(1)

```
$ mkdir myprograms
$ cd myprograms
$ nano hellosub1.c
#include <stdio.h>
#include “hello1.h”
void fun1(void){
  printf("hello1 %d\n”, CONSTANT1);
}
$ gcc –c hellosub1.c
$ ls
.. hellosub1.o
```



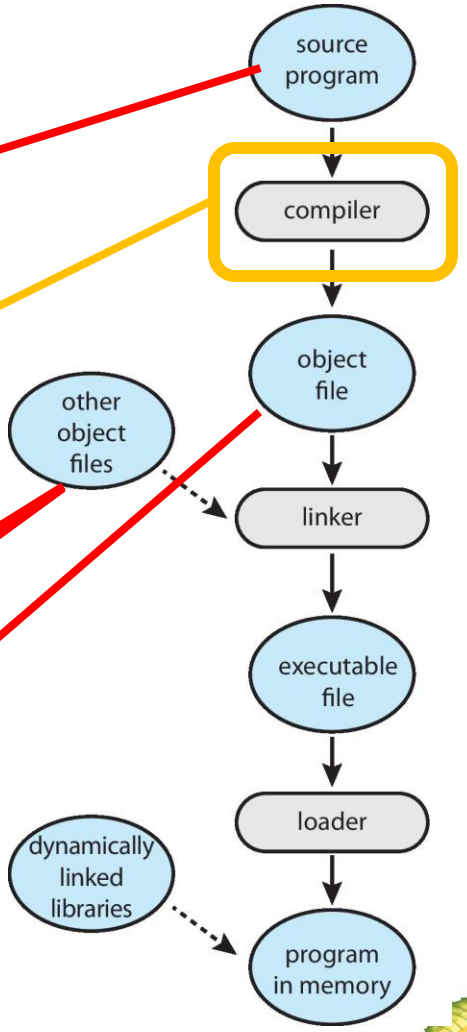

# การคอมไพล object code “hellosub2.o”

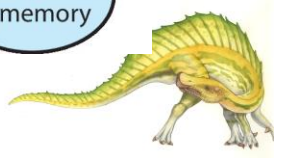
```
$ nano hellosub2.c
#include <stdio.h>

void fun2(void){
  printf("hello2\n");
}
$ gcc -c hellosub2.c
$ ls
.. hellosub1.o
.. hellosub2.o
```




# การคอมไพล object code “hellomain.o”

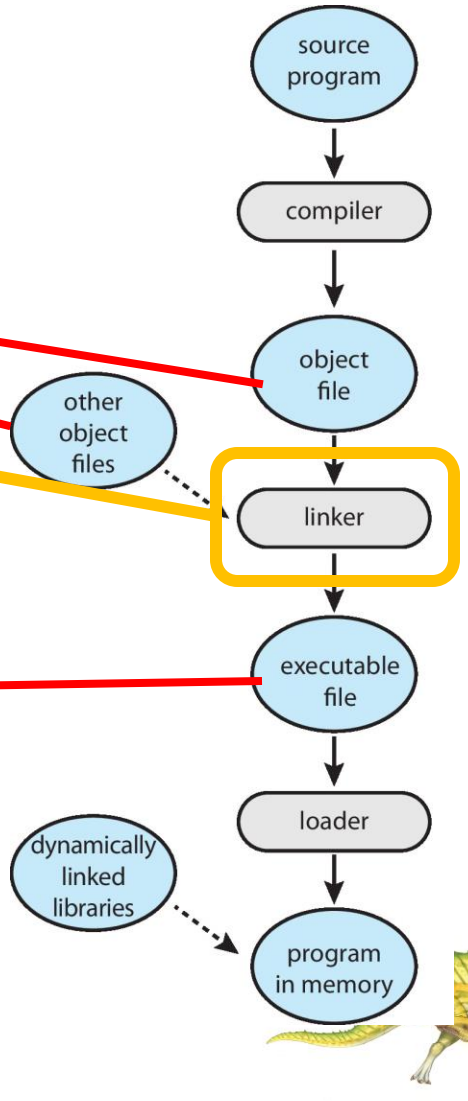
```
$ nano hellomain.c
#include <stdio.h>
extern void fun1(void);
extern void fun2(void);

int main(){
  fun1();
  fun2();
}
$ gcc –c hellomain.c
$ ls –l
.. hellosub1.o
.. hellosub2.o ..hellomain.o
```



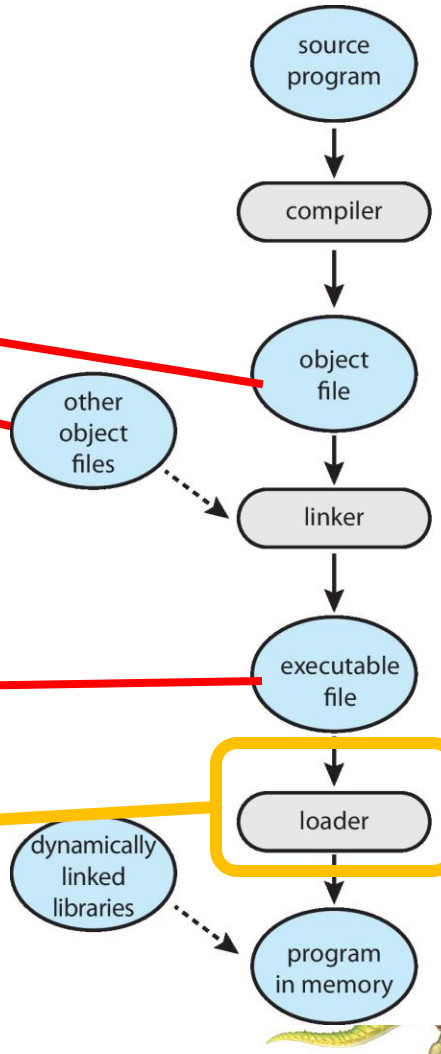
# การlink โปรแกรม (4)

```
$ gcc -o hello hellomain.o \
> hellosub1.o hellosub2.o
$ ls -l
.. hellmain.o hellosub1.o
.. hellosub2.o
.. hello
$ file *
$ ./hello
```



# การ load โปรแกรม (4)

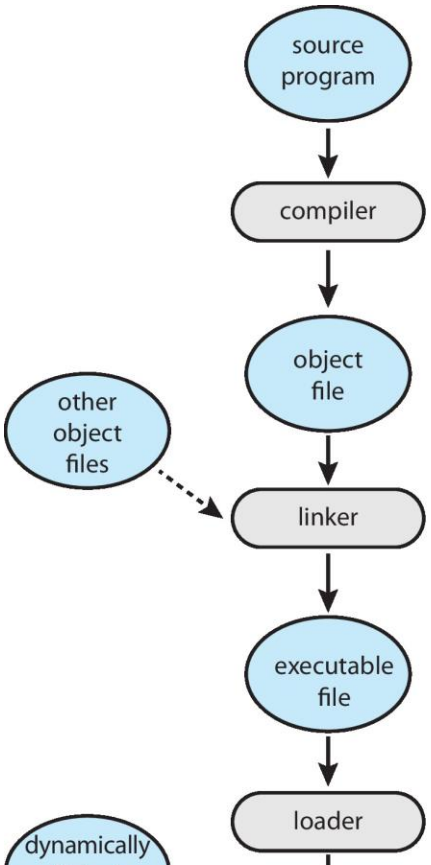
```
$ gcc -o hello hellomain.o \
> hellosub1.o hellosub2.o
$ ls -l
.. hellmain.o hellosub1.o
.. hellosub2.o
.. hello
$ file *
$ ./hello
```



# การ execute และ dynamic link โปรแกรม (4)

```
$ ./hello
```

- เกิด Process hello ในระบบ
- เมื่อ hello จะเรียกใช้ printf() ซึ่งมีcode ใน library stdio
- มันจะโหลดเนื้อหาของ stdio library เข้าสู่ memory แล้ว Process hello จะเรียกใช้ printf()



# Why Applications are Operating System Specific

- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own **unique system calls**
  - Own file formats, etc.
- แต่ก็มี..... Apps can be multi-operating system
  - Written in **interpreted language** like Python, Ruby, and interpreter available on multiple operating systems
  - App written in **language that includes a VM** containing the running app (like Java)
  - Use **standard** language (like C), compile separately on each operating system to run on each (เปรียบเทียบกับ มาตรฐาน html, TCP/IP, มาตรฐานอุตสาหกรรม, etc.)
- **Application Binary Interface** (**ABI**) is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.

# Policy and Mechanism

- **Policy**: **What** needs to be done?
  - Example: Interrupt after every 100 seconds
- **Mechanism**: **How** to do something?
  - Example: timer
    - ดังเมื่อนับถอยหลังถึง 0 หรือ ดังเมื่อเทียบเวลาปัจจุบันตรงกับที่กำหนดไว้ก่อนหน้า
- Important principle: separate policy from mechanism
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.
  - Example: change 100 to 200

# Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

# Operating System Structure

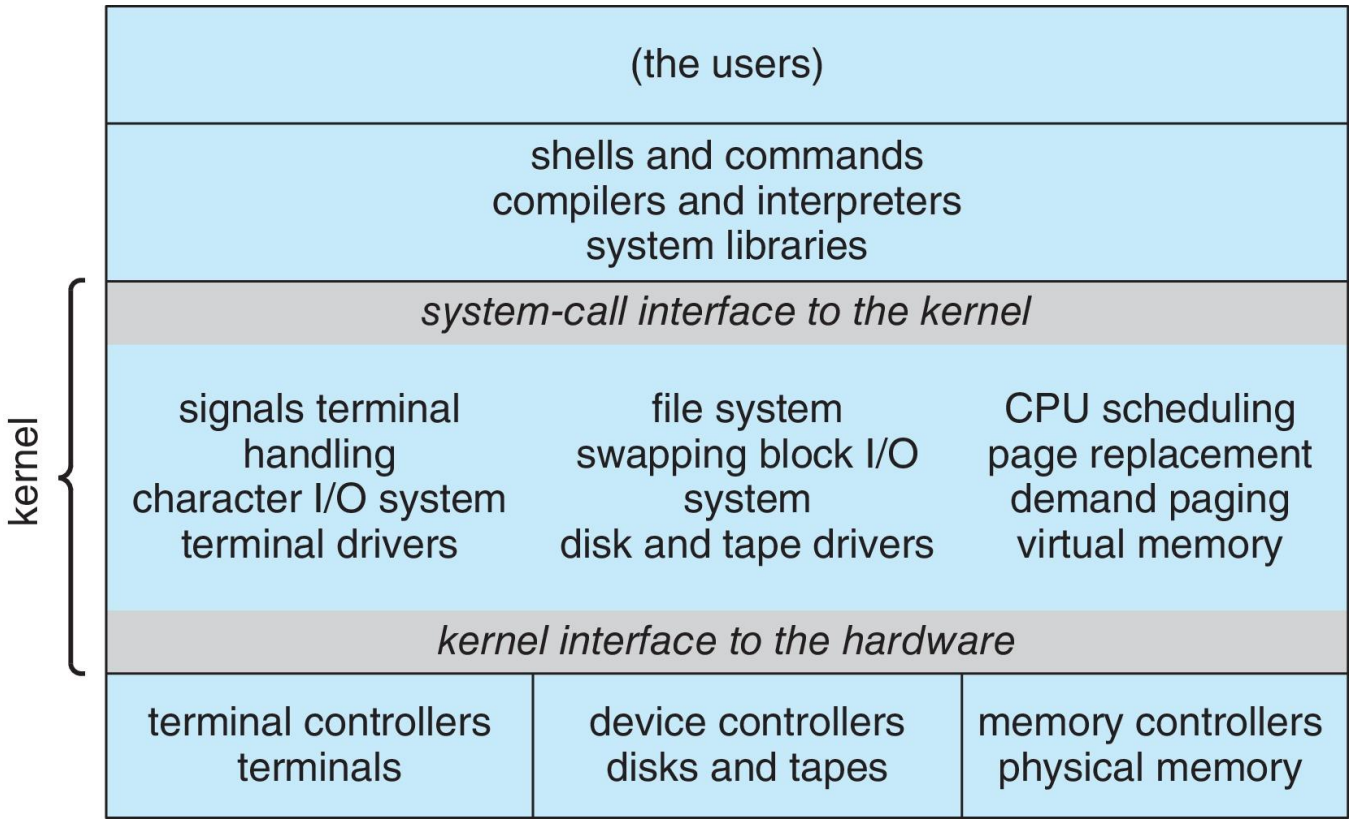
- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Layered – an abstraction
  - Microkernel – Mach

# Monolithic Structure – Original UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
    - ดูภาพ ถัดไป

# Traditional UNIX System Structure

Beyond simple but not fully layered

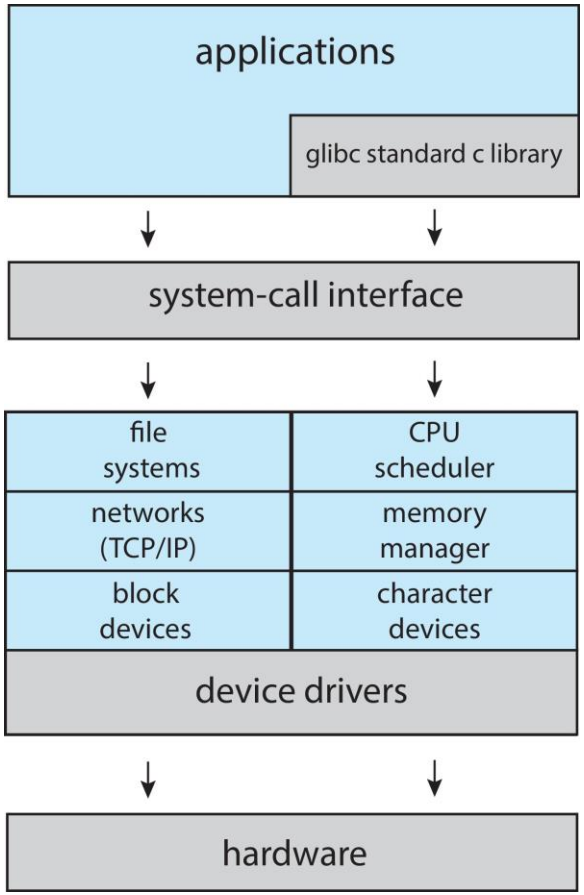


# Linux System Structure

Monolithic plus modular design

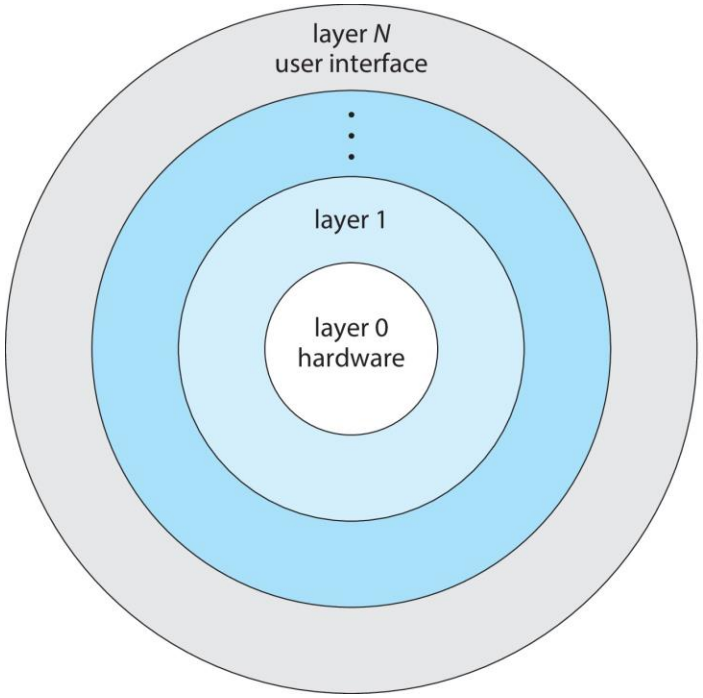
ปรับปรุงเปลี่ยนแปลงได้ง่าย
เช่น สามารถเพิ่มลด

- device driver
- Kernel module



# Layered Approach

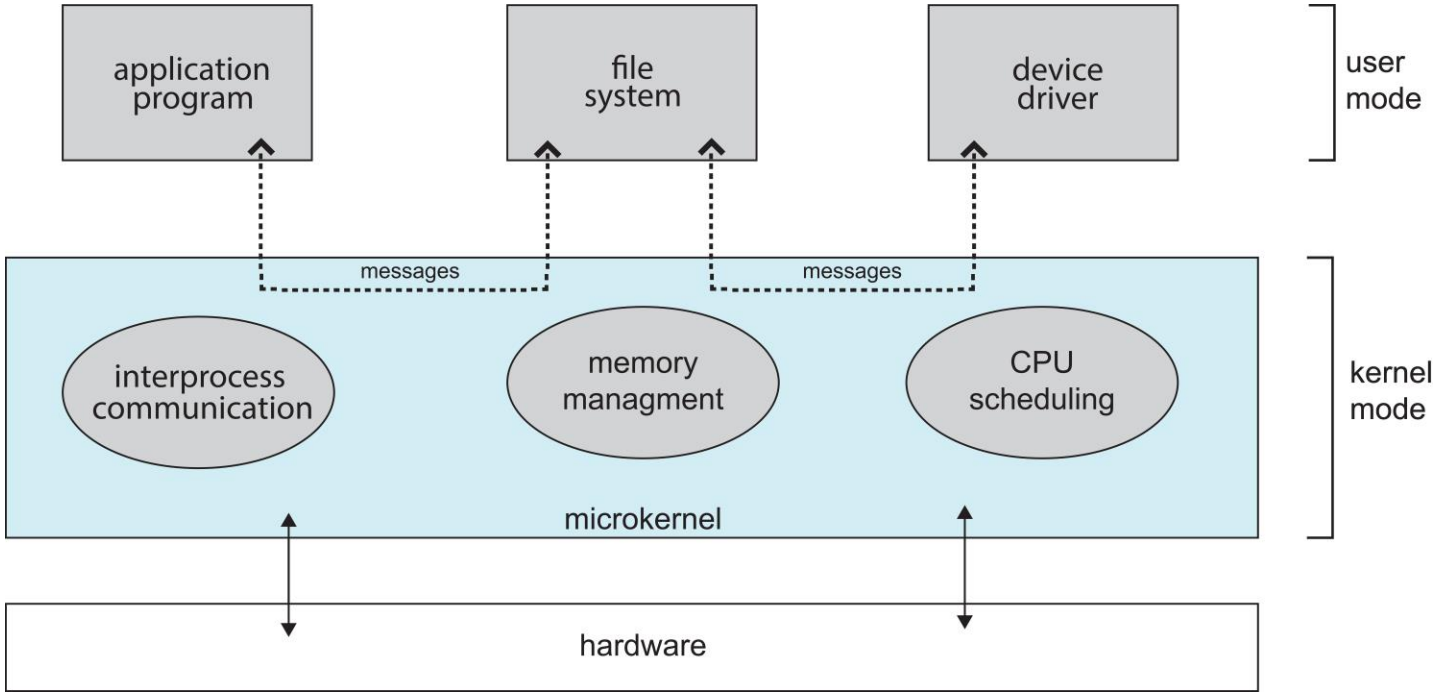
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



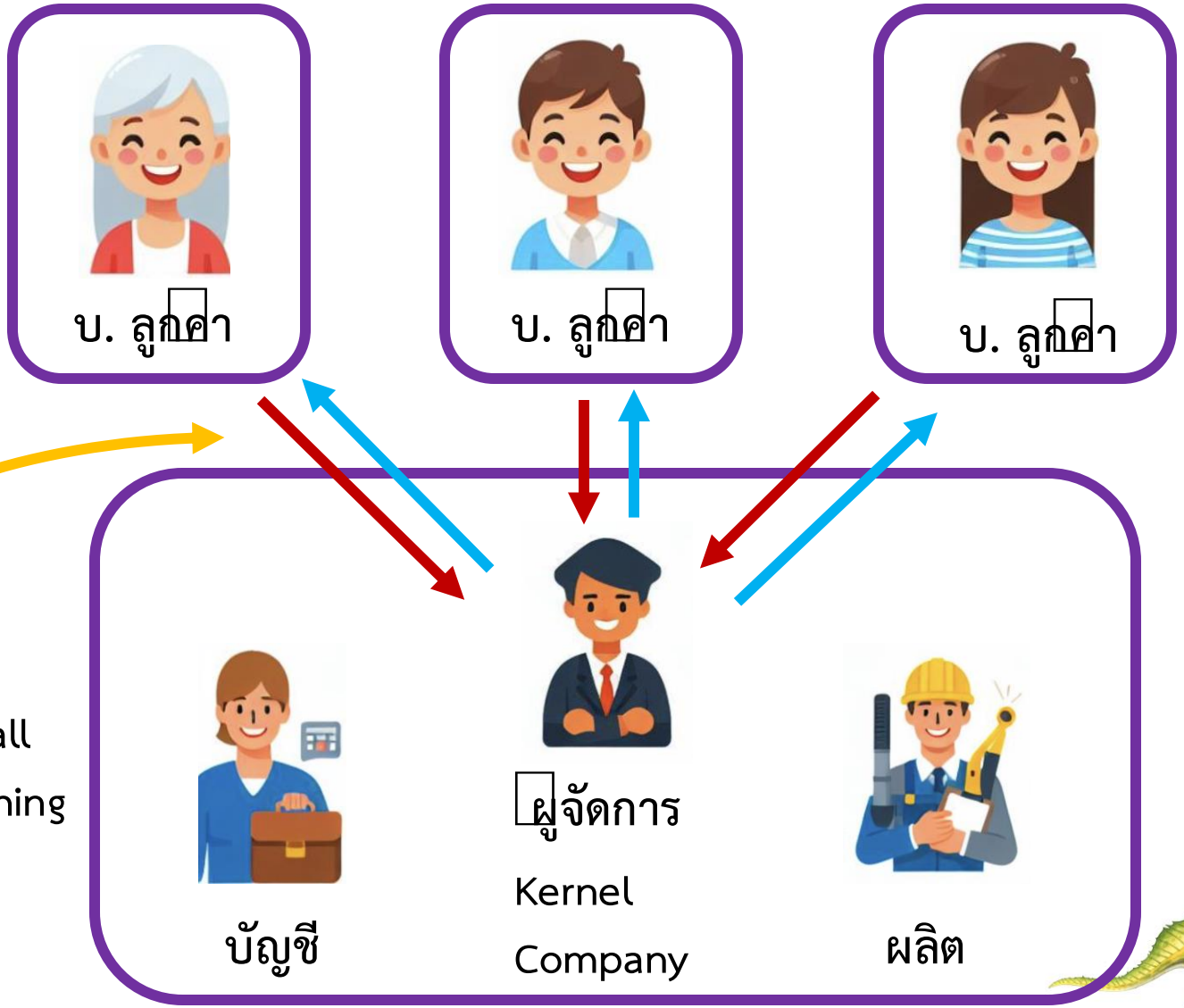
# Microkernels

- Moves as much from the kernel into user space
- **Mach** is an example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

# Microkernel System Structure



# Monolithic System




ถ้าเปรียบการขอใช้บริการแต่ละครั้ง เหมือน 1 System Call จะมี Context Switching รวมกี่ครั้ง

# Microkernel System

บ. ลูกค้า

บ. ลูกค้า

บ. ลูกค้า

บ. บัญชี

บ. ผลิต

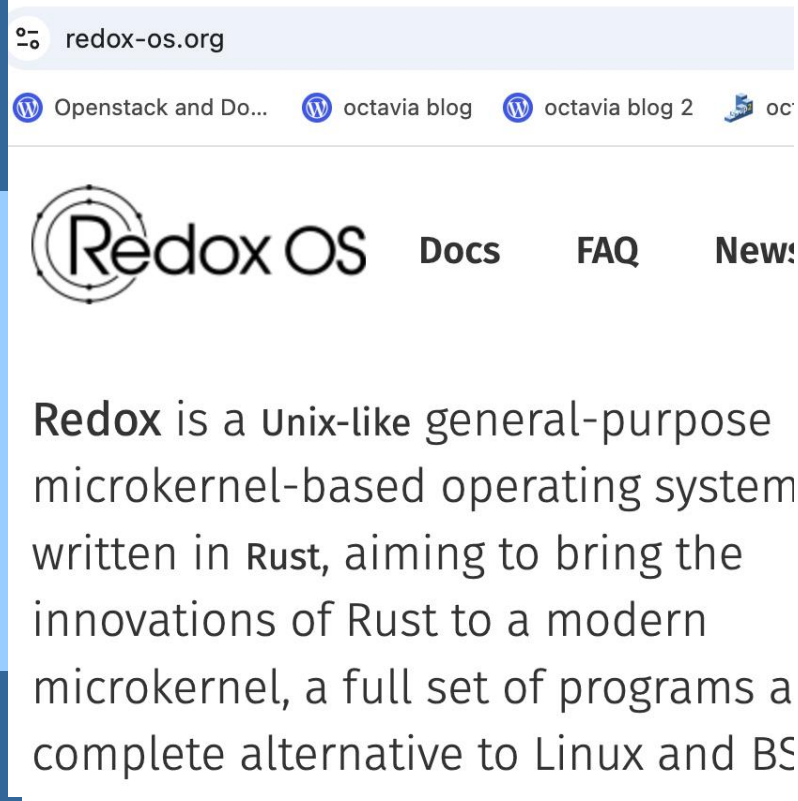
เพิ่มการขอใช้บริการข้ามบริษัท

เพิ่ม System Call

เพิ่ม context switching

ผู้จัดการ Micro Kernel Company

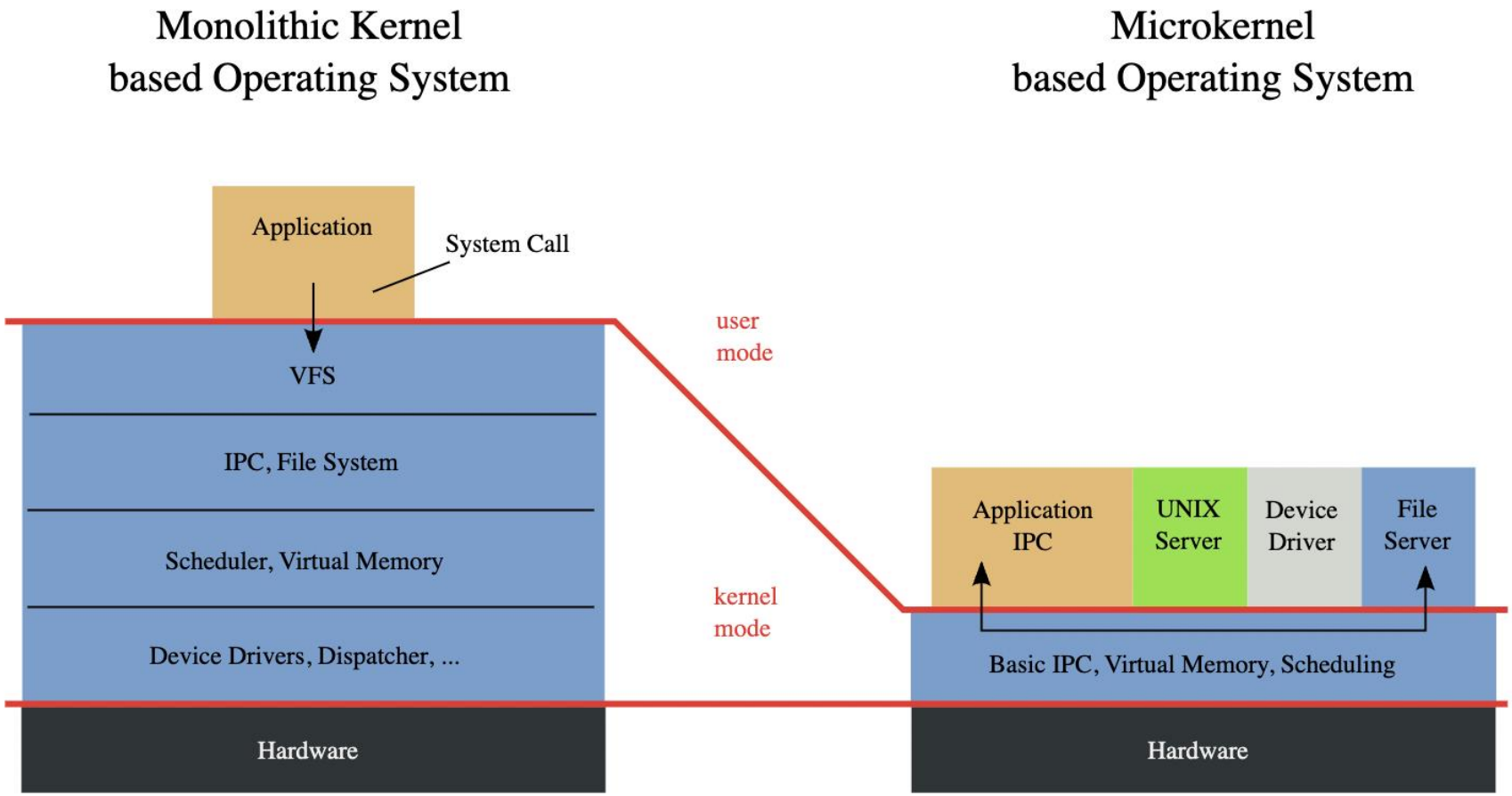
# Example: Redox OS




- Inspired by Plan 9, Minix, seL4, BSD and Linux
- Implemented in Rust
- Microkernel Design
- Includes optional GUI - Orbital
- Partial POSIX compatibility
- Source compatibility with Linux/BSD programs

- MIT Licensed
- Supports Rust Standard Library
- Drivers run in Userspace
- Includes common Unix/Linux tools
- Custom C library written in Rust (relibc)
- See Redox in Action

# Example: Redox OS




# Modules

- เป็นอีกทางเลือกหนึ่ง เสริมให้ Monolithic มีความยืดหยุ่น extensible มากขึ้น
- Many modern operating systems implement **loadable kernel modules** (**LKMs**)
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
  - OS ยังเป็น Monolithic แต่ใช้วิธีเพิ่ม Loadable Module (หรือ Object) เมื่อต้องการเพิ่มความสามารถของ Kernel
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc.
- ปัญหา: ขึ้นอยู่กับ kernel version และตัว Module ต้องถูกปรับตาม kernel version ที่เปลี่ยนไป (Module ไม่ได้เป็นส่วนหนึ่งของ kernel ดังนั้นผู้ maintain Module ต้องปรับและทดสอบกับ kernel version ใหม่เสมอ

# Boot process

1. รันโปรแกรม bootstrap หรือ boot loader เพื่อหา OS kernel ใน disk
2. โหลด OS kernel เข้าสู่หน่วยความจำ RAM
3. Kernel กำหนดค่าเริ่มต้นให้ hardware
4. Kernel **mount** root file system

# System Boot

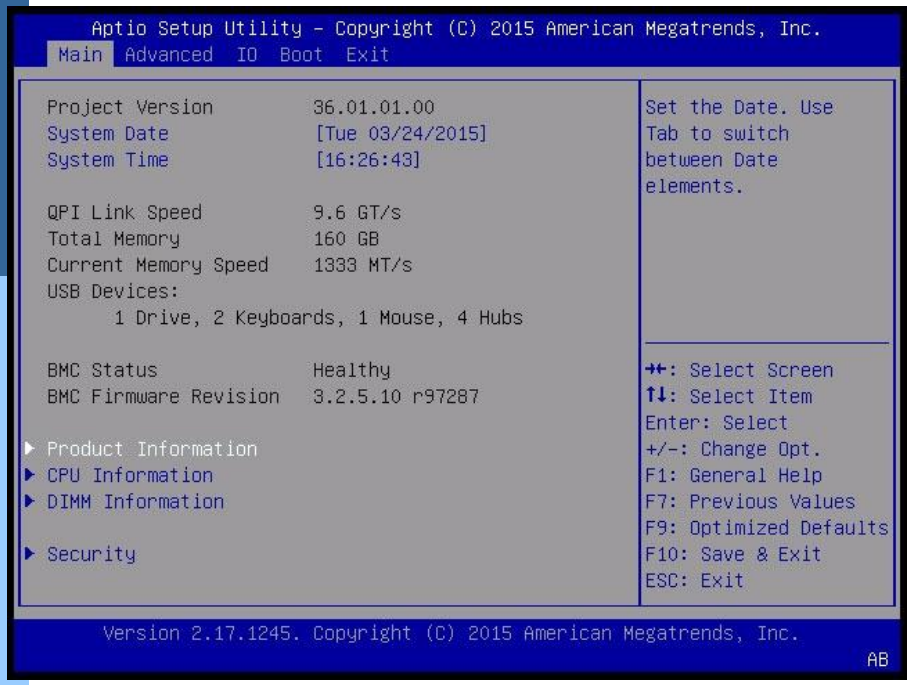
- When power initialized on system, execution starts at a fixed memory location
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, **BIOS**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
  - Modern systems replace BIOS with **Unified Extensible Firmware Interface** (**UEFI**)
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**
- Boot loaders frequently allow various boot states, such as single user mode

# Tasks of the boostrap program

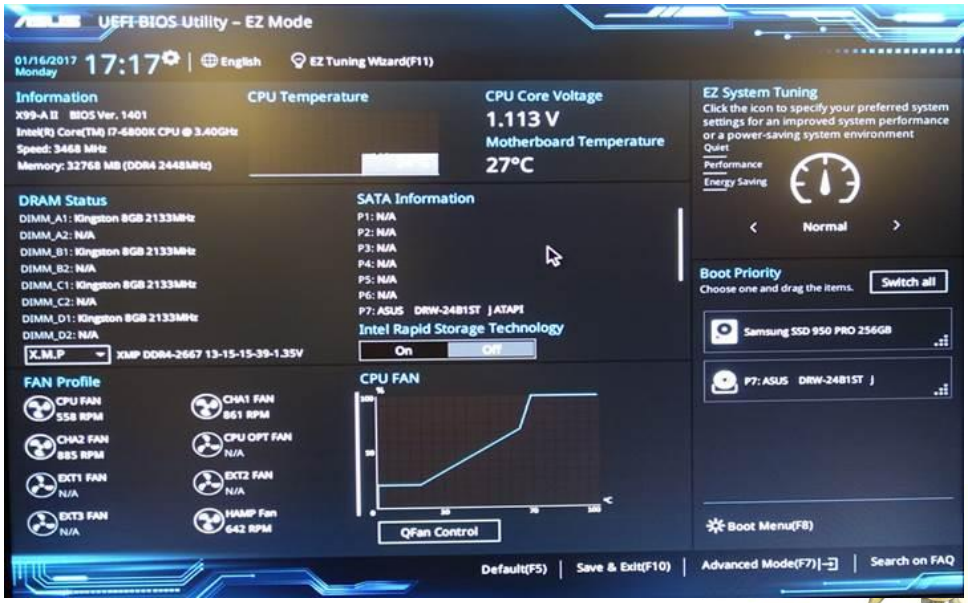
## Bootstrap Loader:

- เช็คความถูกต้องของฮาร์ดแวร์ ตรวจสอบ CPU memory I/O devices
- Set ค่าเริ่มต้นของ registers ของ CPU และ device controllers และค่าใน memory
- โหลดและเริ่มต้นรัน OS และ mount root file system

# BIOS VS UEFI




http://www.davidapps.net/ArcGISpc/OS/help/configure_the_asus_uefi_bios_for_os_install.htm

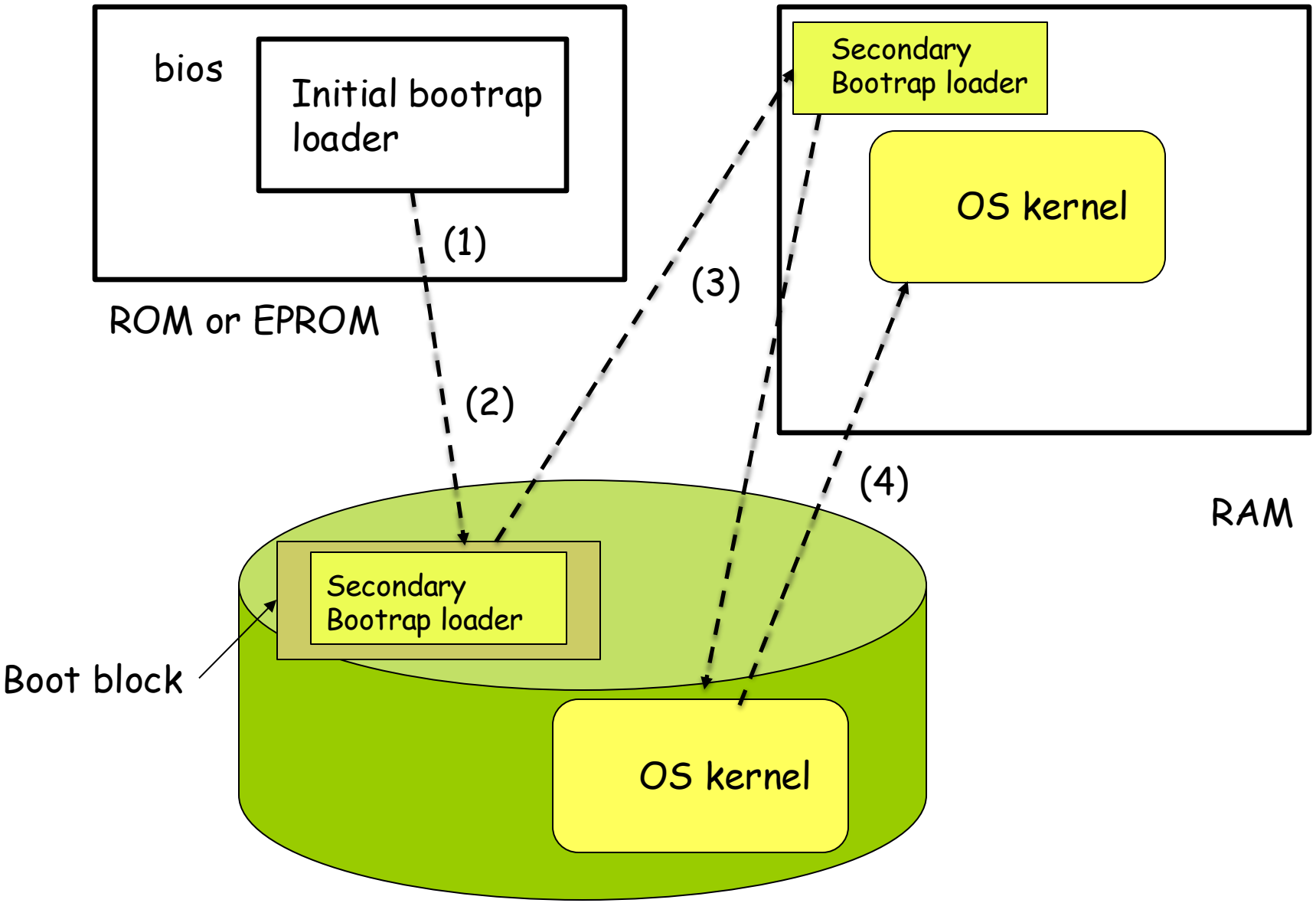



https://docs.oracle.com/cd/E56388_01/html/E56396/gnvlw.html

# Boot แบบ Legacy ด้วย BIOS

- สมัยก่อน เราไม่มี SSD ดังนั้น ระบบคอมพิวเตอร์ถูกออกแบบให้ซีพียูรันโปรแกรมเริ่มต้นเมื่อเราเปิดเครื่องให้เริ่มทำงาน จาก Memory Chip เรียกว่า ROM (Read Only Memory) ซึ่งเป็นหน่วยความจำแบบ Non-Volatile ที่เก็บโปรแกรมขนาดเล็กเรียกว่า BIOS (Basic Input/Output System) ไว้ในนั้น (สามาตุที่ต้องเป็นโปรแกรมขนาดเล็กเพราะ ROM เก็บข้อมูลได้น้อย
- โปรแกรม BIOS จะประมวลผลดังนี้
  1. เช็คความถูกต้องของฮาร์ดแวร์ เช่น ตรวจสอบ CPU memory I/O devices และ กำหนดค่าเริ่มต้นของ registers ของ CPU และ device controllers และค่าใน memory
  2. ตรวจสอบว่าใน Master Boot Record (จุดเริ่มต้นของ Disk/Storage) มีโปรแกรมสำหรับช่วยโหลด (หรือเลือกโหลด OS ในกรณี Disk มีหลาย OS หรือไม่ ถ้าไม่มีก็ โหลด OS เข้าสู่ RAM และ mount root file system
  3. ถ้ามีโปรแกรมช่วยโหลด ก็โหลดโปรแกรมนั้นเข้าสู่หน่วยความจำแล้ว ให้ซีพียูประมวลผลโปรแกรมนั้น โปรแกรมจะให้ผู้ใช้เลือกชนิดหรือเวอร์ชั่นของ OS ที่ต้องการ
  4. โปรแกรมช่วยโหลดจะโหลด OS เข้าสู่ RAM และ mount root file system ต่อไป

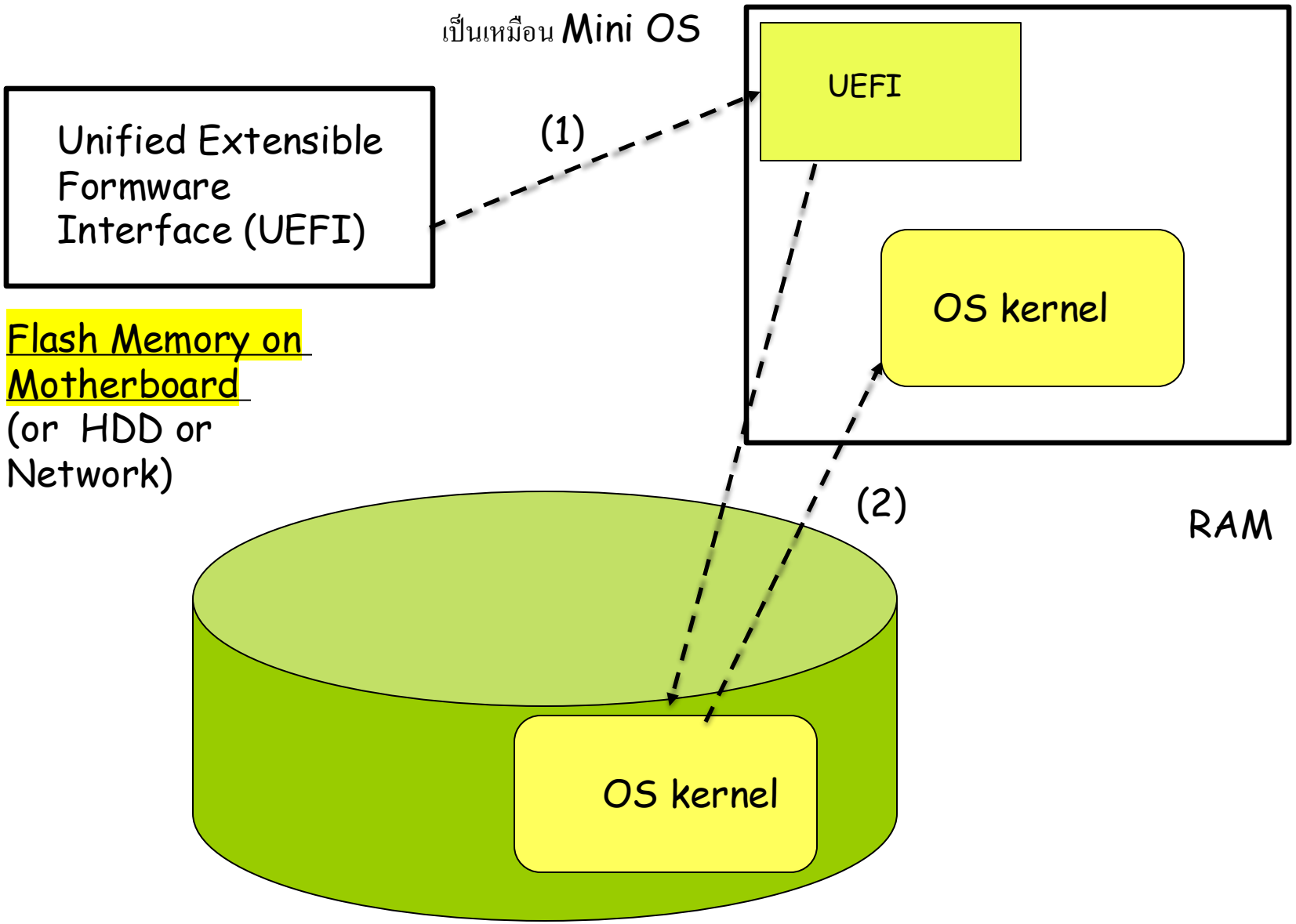
# Legacy


bios
Initial bootrap loader
(1)
ROM or EPROM
(2)
(3)
Secondary Bootrap loader
OS kernel
(4)
RAM
Secondary Bootrap loader
Boot block
OS kernel

# Boot ด้วย UEFI

- Unified Extensible Firmware Interface (UEFI) เป็นโปรแกรม Boot Strap Loader ที่ถูกเก็บใน SSD ของระบบคอมพิวเตอร์ ระบบคอมพิวเตอร์สมัยใหม่ถูกออกแบบให้ซีพียูเริ่มต้นการทำงานโดยโหลดโปรแกรม UEFI จาก SSD บน motherboard (หรือบน HDD หรือ Network ก็ได้) เข้าสู่ RAM แล้วประมวลผลโปรแกรมนั้น เนื่องจาก SSD สามารถมีความจุมาก โปรแกรม UEFI จึงมีขนาดใหญ่และรวมโปรแกรม Driver สำหรับใช้งานอุปกรณ์ต่างๆไว้ด้วยจึงสามารถรัน GUI และทำงานที่ซับซ้อนได้มากกว่า BIOS
- หลักจากถูกโหลดเข้าสู่ RAM ซีพียูจะรันโปรแกรม UEFI เพื่อทำต่อไปนี้
  1. เช็คความถูกต้องของฮาร์ดแวร์ เช่น ตรวจสอบ CPU memory I/O devices และกำหนดค่าเริ่มต้นของ registers ของ CPU และ device controllers และค่าใน memory
  2. โหลด OS จาก Master Boot Record (จุดเริ่มต้นของ Disk/Storage) เข้าสู่ RAM และ mount root file system

# UEFI

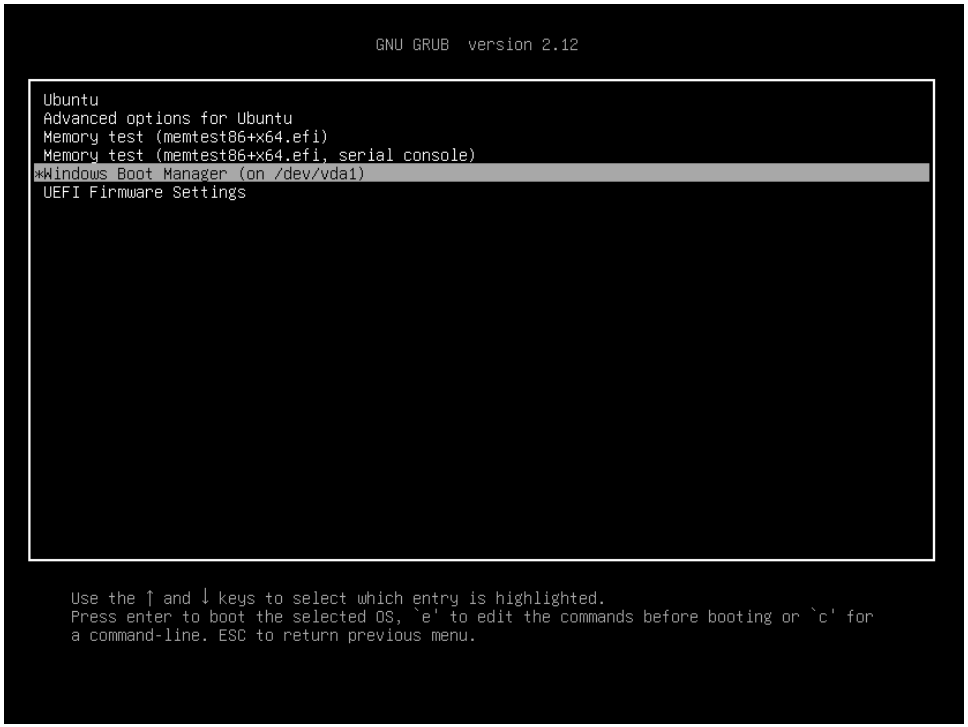

เป็นเหมือน Mini OS
Unified Extensible Formware Interface (UEFI)
(1)
UEFI
OS kernel
Flash Memory on Motherboard
(or HDD or Network)
(2)
RAM
OS kernel

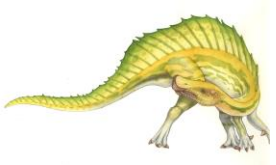
# Grand unified bootloader (Grub)

- Grand unified bootloader (GRUP) เป็น bootstrap program ของ Linux ที่อนุญาตให้
  1. ผู้ใช้เลือก OS และ
  2. กำหนดค่าพารามีเตอร์ ให้กับ OS program ได้ (เปลี่ยนแปลงค่าพารามี อ ์ ไ ด ้ )
- GRUP มีทั้ง version ที่ทำงานร่วมกับ BIOS และ UEFI
- เป็น Secondary Bootstrap Loader

# ตัวอย่าง

- https://sysguides.com/dual-boot-windows-11-and-ubuntu#google_vignette

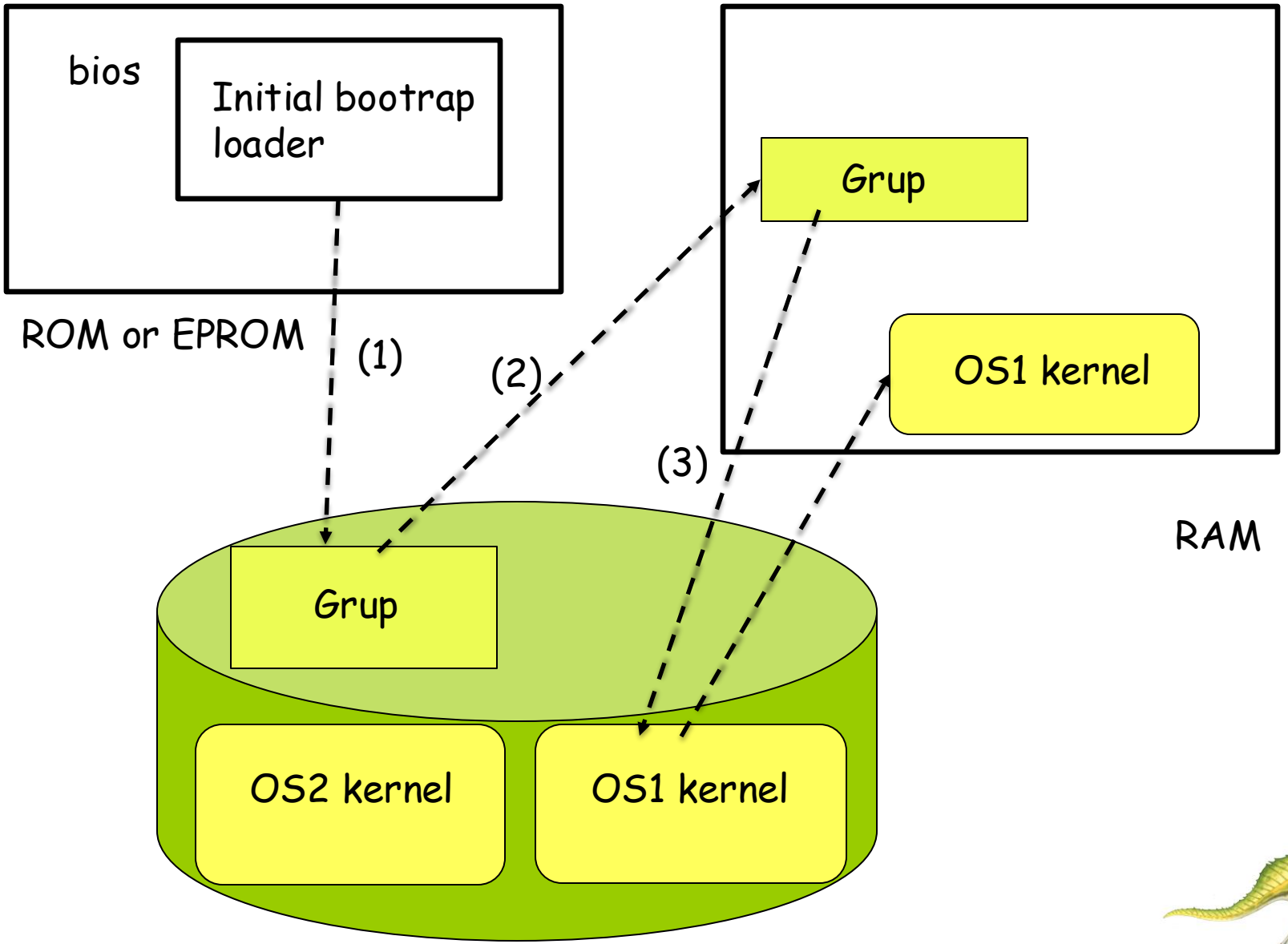



# Boot แบบ Legacy ด้วย BIOS-Grup

- เราใช้งาน BIOS ร่วมกับ Grup ได้
- โปรแกรม BIOS จะประมวลผลดังนี้
  1. เช็คความถูกต้องของฮาร์ดแวร์ เช่น ตรวจสอบ CPU memory I/O devices และ กำหนดค่าเริ่มต้นของ registers ของ CPU และ device controllers และค่าใน memory
  2. โหลด Grub จาก Master Boot Record (จุดเริ่มต้นของ Disk/Storage) เข้าสู่ RAM แล้วประมวลผล Grub
  3. Grub จะ ผู้ใช้เลือกชนิดหรือเวอร์ชั่นของ OS ที่ต้องการ (OS1 หรือ OS2 ในภาพ) และ
  4. โหลด OS เข้าสู่ RAM และ mount root file system

# BIOS-grup

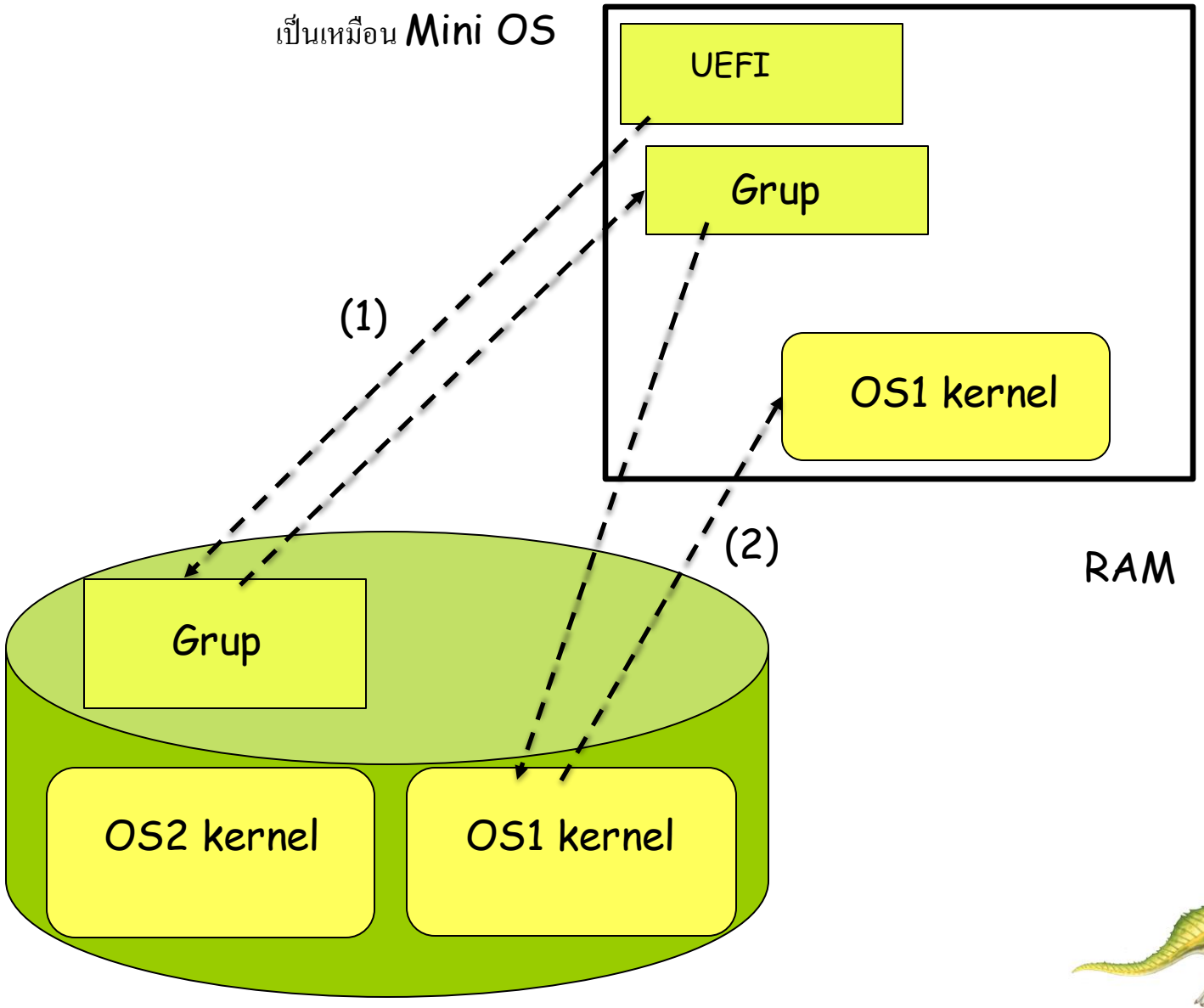

bios
Initial bootrap loader
ROM or EPROM
(1)
(2)
(3)
Grup
OS1 kernel
RAM
Grup
OS2 kernel
OS1 kernel

# Boot ด้วย UEFI-Grup

- เราใช้งาน UEFI ร่วมกับ Grup ได้
- ซีพียูโหลด UEFI เข้าสู่ RAM แล้วโปรแกรม UEFI จะประมวลผลดังนี้
  1. เช็คความถูกต้องของฮาร์ดแวร์ เช่น ตรวจสอบ CPU memory I/O devices และ กำหนดค่าเริ่มต้นของ registers ของ CPU และ device controllers และค่าใน memory
  2. โหลด Grub จาก Master Boot Record (จุดเริ่มต้นของ Disk/Storage) เข้าสู่ RAM แล้วประมวลผล Grub
  3. Grub จะ ผู้ใช้เลือกชนิดหรือเวอร์ชั่นของ OS ที่ต้องการ (OS1 หรือ OS2 ในภาพ) และ
  4. โหลด OS เข้าสู่ RAM และ mount root file system

# UEFI-grup

เป็นเหมือน Mini OS




# Advantages of UEFI over BIOS

- เป็นเหมือน OS เล็กๆของ mainboard ที่โหลดเข้าสู่หน่วยความจำและประมวลผลระบบคอมพิวเตอร์เพื่อดูแลระบบและโหลดโอเอสจาก disk (ทำอะไรๆได้มากกว่า BIOS)
- UEFI มาแทน BIOS และให้บริการแบบ BIOS ได้ด้วย
- เร็วสามารถโหลดโอเอสได้เร็วในขั้นตอนเดียว
- แต่มีความยืดหยุ่น ปรับรูปแบบการโหลดได้ตามความต้องการหรือ policy ของผู้ใช้
- จาก https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface

## Advantages [ edit ]

The interface defined by the EFI specification includes data tables that contain platform information, and boot and runtime services that are available to the OS loader and OS. UEFI firmware provides several technical advantages over a traditional BIOS system:[18]

- Ability to boot a disk containing large partitions (over 2 TB) with a GUID Partition Table (GPT)[19][a][20]
- Flexible pre-OS environment, including network capability, GUI, multi language
- 32-bit (for example IA-32, ARM32) or 64-bit (for example x64, AArch64) pre-OS environment
- C language programming
- Modular design
- Backward and forward compatibility

# สรุป

- โครงสร้างของ OS
- รายละเอียดของ System Call ชนิดของ System Call
- บริการของ OS
- แนะนำส่วนประกอบอื่นที่สำคัญของ OS