IT5002

Computer Systems and Applications

Lecture 11

Introduction to Operating Systems

colintan@nus.edu.sg

NUS National University of Singapore
School of Computing

Learning Objectives

- **By the end of this lecture you should be able to:**
  - Describe the onion model.
  - Understand the interaction between layers of the onion model.
  - Describe the options for programming I/O.
  - Give an overview of the key issues in OS design.

What are Operating Systems?

- **An "operating system" is a suite (i.e. a collection) of specialized software that:**
  - Gives you access to the hardware devices like disk drives, printers, keyboards and monitors.
  - Controls and allocate system resources like memory and processor time.
  - Gives you the tools to customize your and tune your system.
- **Examples include LINUX, OS X (a variant of UNIX), Windows 8.**

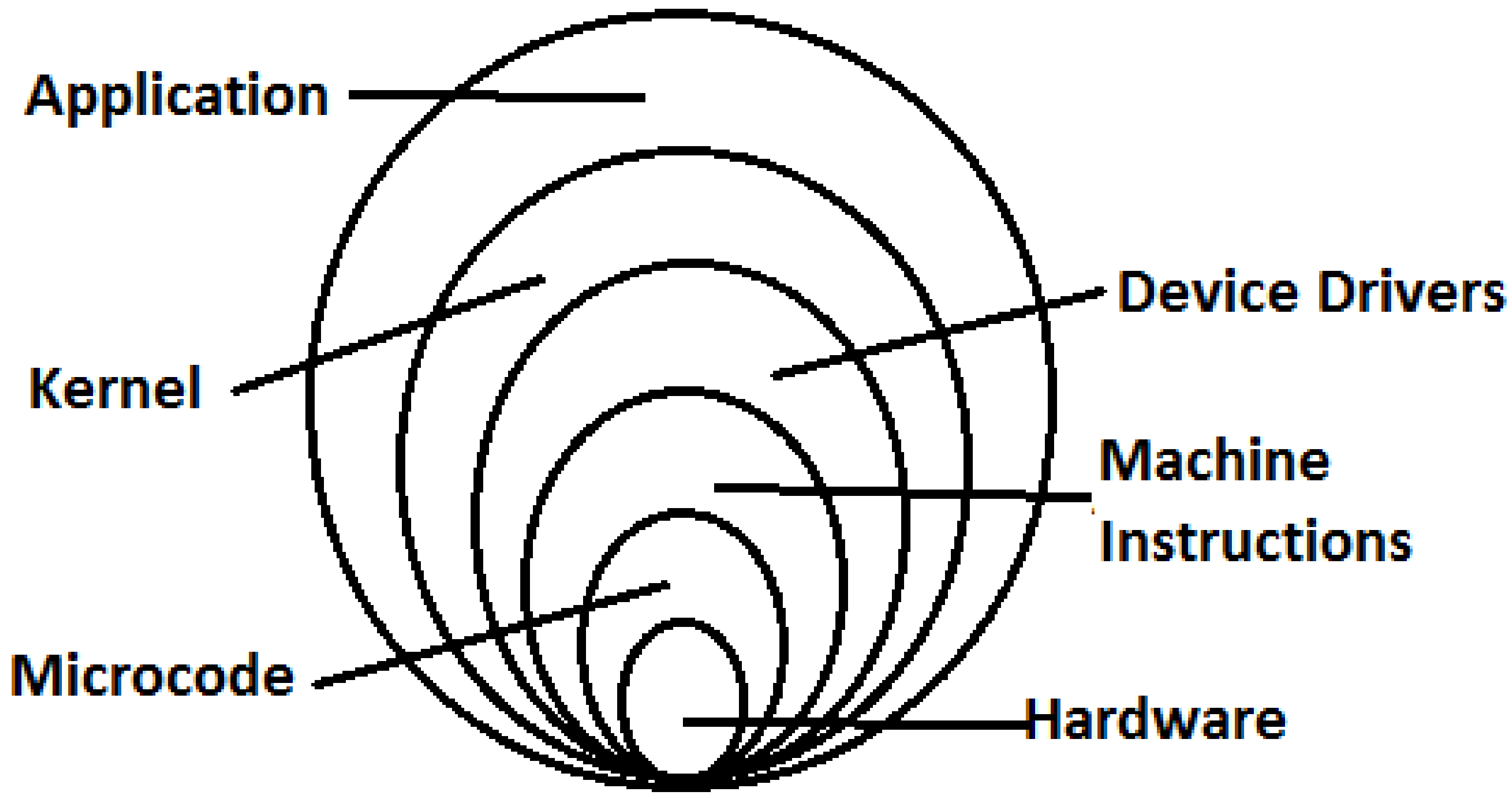
NUS National University of Singapore
School of Computing

What Are Operating Systems?

- **Usually consists of several parts:**
  - Bootloader – First program run by the system on start-up. Loads remainder of the OS kernel.
    - ✓ **On Wintel systems this is found in the Master Boot Record (MBR) on the hard disk.**
  - Kernel – The part of the OS that runs almost continuously.
    - ✓ **The bulk of this course is about how the kernel works.**
  - System Programs – Programs provided by the OS to allow:
    - ✓ **Access to programs.**
    - ✓ **Configuration of the OS.**
    - ✓ **System maintenance, etc.**

NUS National University of Singapore School of Computing

Basic Terminology: The Onion Model.

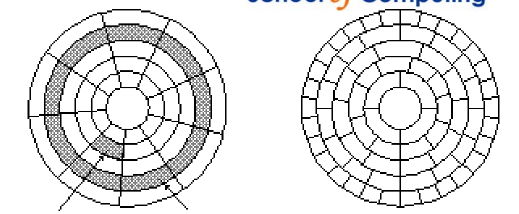


How an OS Works

Bootstrapping

NUS National University of Singapore
School of Computing

How an OS Works
Bootstrapping



- **The OS is not present in memory when a system is "cold started".**
  - When a system is first started up, memory is completely empty.
  - How do we get an operating system into memory?
- **We start first with a bootloader.**
  - Tiny program in the first (few) sector(s) of the hard-disk.
    - ✓**The first sector is generally called the "boot sector" or "master boot record" for this reason.**
  - Job is to load up the main part of the operating system and start it up.

```
7CF7 5E             POP     SI
7CF8 1F             POP     DS
7CF9 8F04           POP     [SI]
7CFB 8F4402         POP     [SI+02]
7CFE CD19           INT     19                 ; If a key was pressed, then
                                               ;   Start over again with
                                               ;   System BOOTSTRAP LOADER.
```

```
7D00 58             POP     AX
7D01 58             POP     AX
7D02 58             POP     AX
7D03 EBE8           JMP     7CED


7D05 8B471A         MOV     AX,[BX+1A]
7D08 48             DEC     AX
7D09 48             DEC     AX
7D0A 8A1E0D7C       MOV     BL,[7C0D]
7D0E 32FF           XOR     BH,BH
7D10 F7E3           MUL     BX
7D12 0306497C       ADD     AX,[7C49]
7D16 13164B7C       ADC     DX,[7C4B]
7D1A BB0007         MOV     BX,0700
7D1D B90300         MOV     CX,0003
7D20 50             PUSH    AX
7D21 52             PUSH    DX
7D22 51             PUSH    CX
7D23 E83A00         CALL    7D60
7D26 72D8           JB      7D00
7D28 B001           MOV     AL,01
7D2A E85400         CALL    7D81                 ; -> Read Sectors
7D2D 59             POP     CX
7D2E 5A             POP     DX
7D2F 58             POP     AX
7D30 72BB           JB      7CED
7D32 050100         ADD     AX,0001
7D35 83D200         ADC     DX,+00
7D38 031E0B7C       ADD     BX,[7C0B]
7D3C E2E2           LOOP    7D20

7D3E 8A2E157C       MOV     CH,[7C15]
7D42 8A16247C       MOV     DL,[7C24]
7D46 8B1E497C       MOV     BX,[7C49]
7D4A A14B7C         MOV     AX,[7C4B]
7D4D EA00007000     JMP     FAR PTR 0070:0000    ; Same as jumping to
                                                 ;  0000:0700 (start DOS).
```

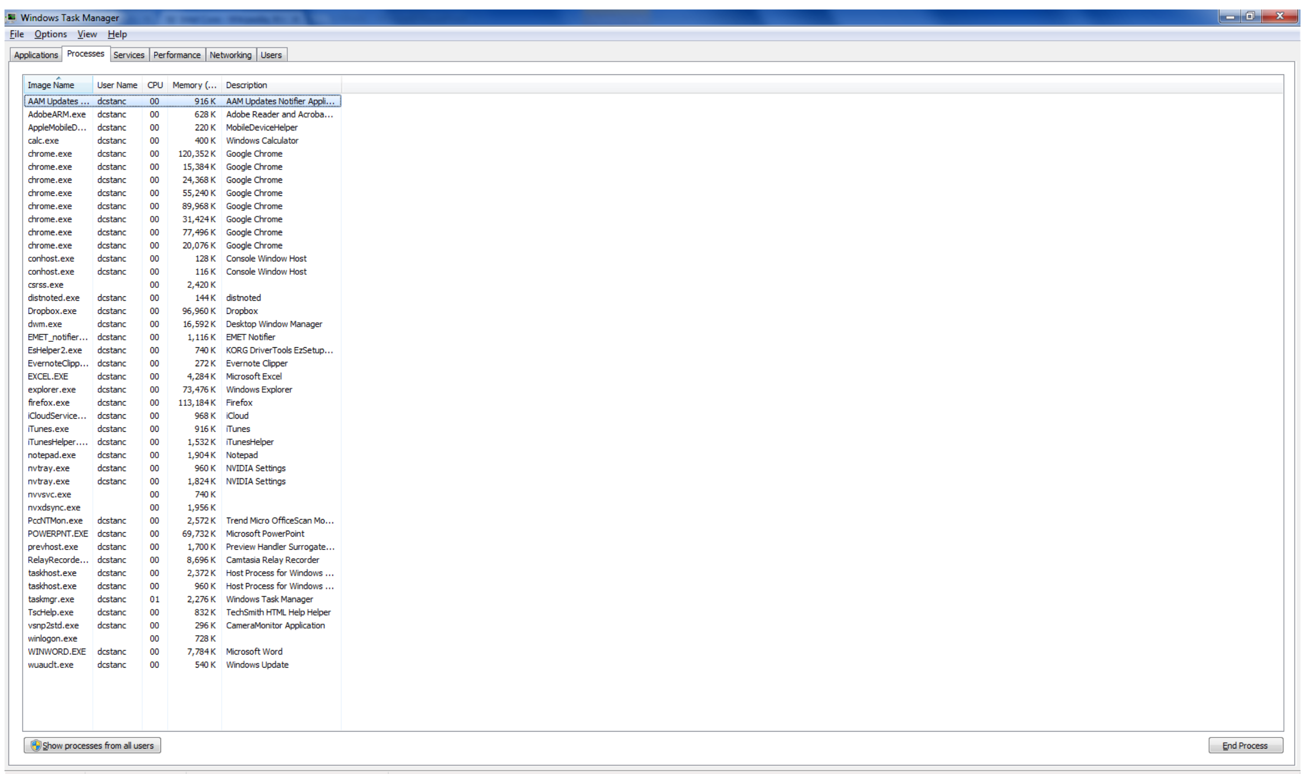
How an OS Works

Process Management

NUS National University of Singapore
School of Computing

How an OS Works
Context Switching

- **A shortage of cores.**
  - A typical system today has two to four "cores".
    - ✓ **CPU units that can execute processes.**
  - We have much more than 4 processes to run.



How an OS Works
Context Switching

- **To manage, we share a core very quickly between multiple processes.**
- **Key points:**
  - Entire sharing must be transparent.
  - Processes can be suspended and resumed arbitrarily.
    - ✓ **I.e. it is not usually possible to build in this "sharing" into a process.**
- **Solution:**
  - Save the "context" of the process to be suspended.
  - Restore the "context" of the process to be (re)started.

How an OS Works
Scheduling

- **So we see that a single system may have multiple processing units ("cores"), but there will generally be many more processes than cores.**
  - We settled this problem with context switching.
- **BUT how do we choose which process to run if several want to run?**
  - Should interactive processes be chosen over background processes?
    - ✓ **Obviously not all the time because background processes will starve**
  - Are some processes more important than others?
  - This is the issue of scheduling.

How an OS Works

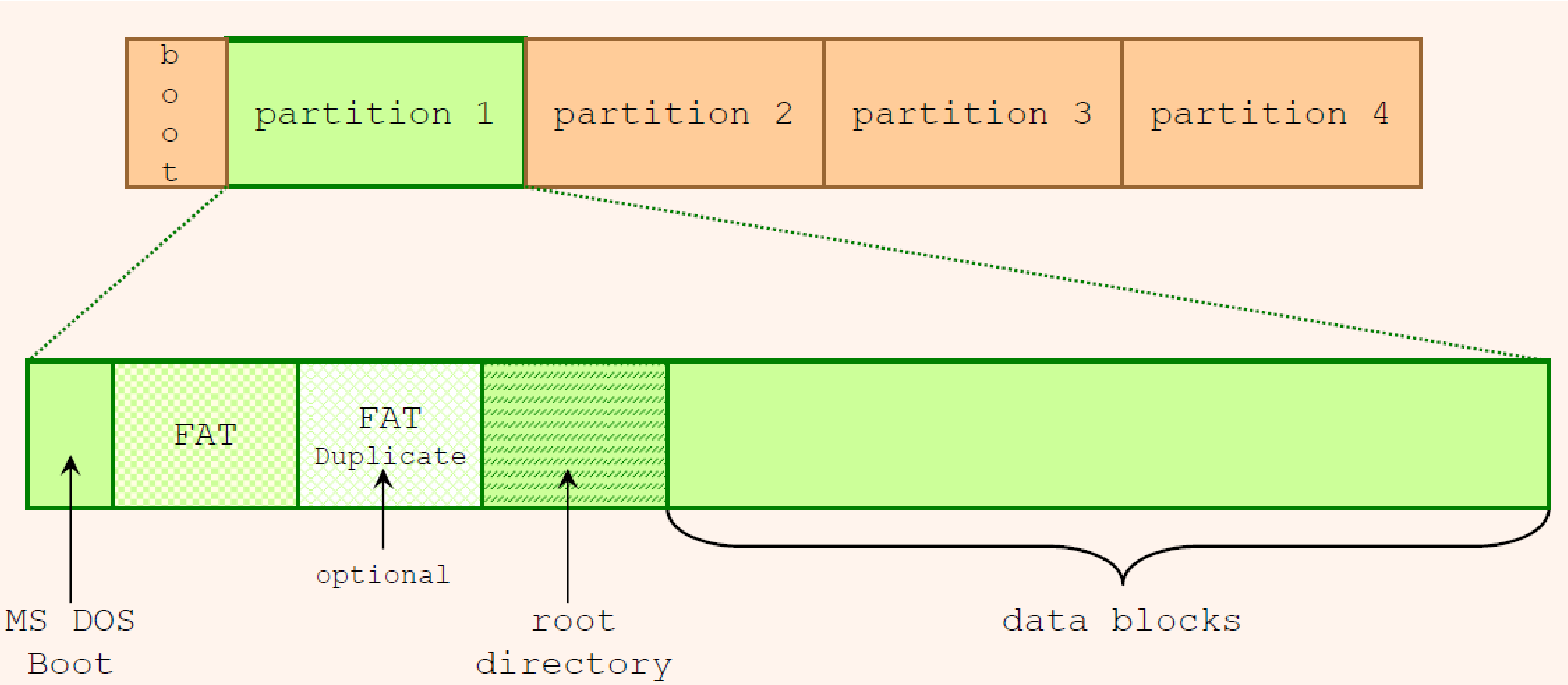
Organizing Data On Your Computer

School of Computing

How an OS Works
File Systems

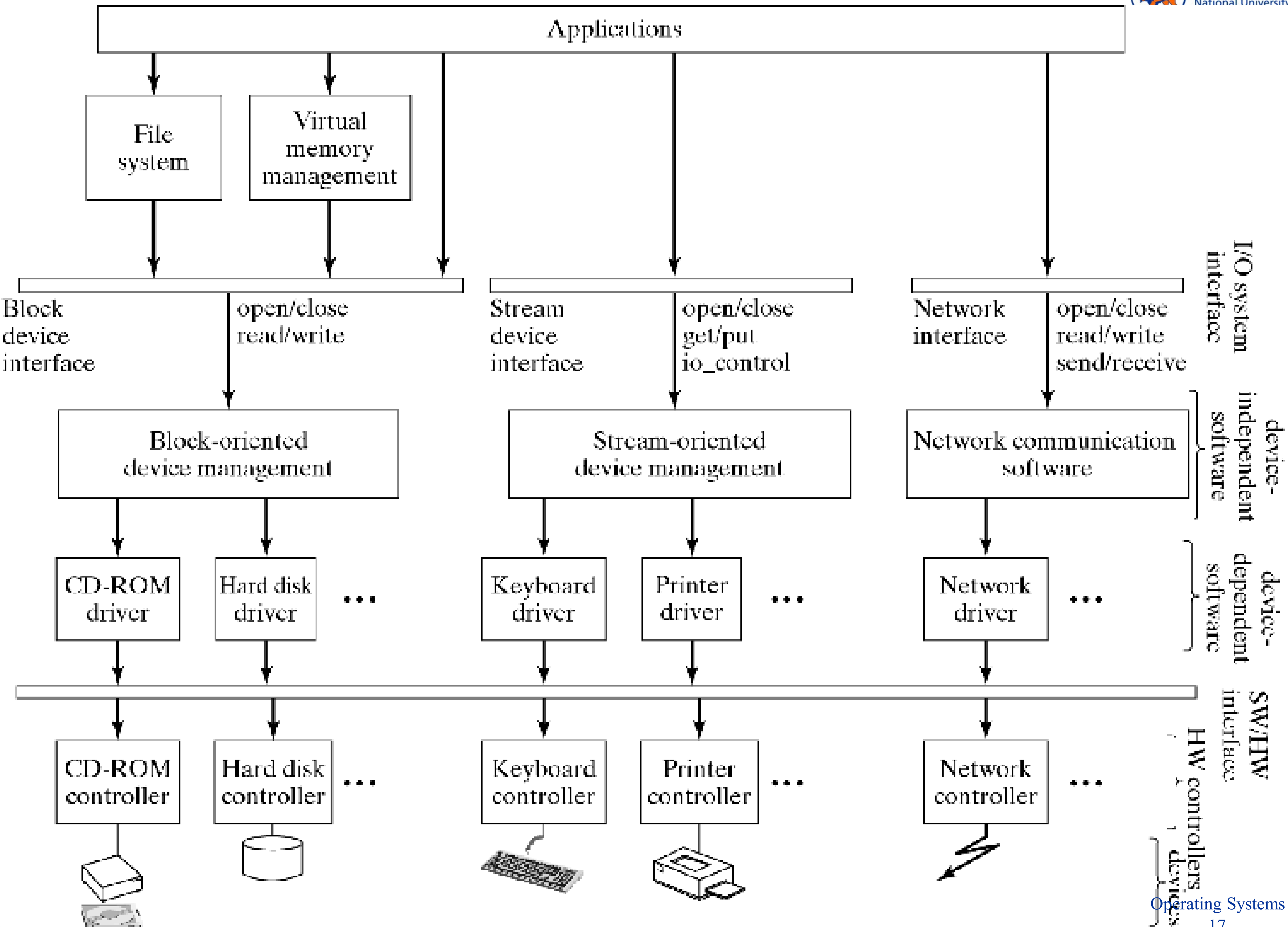
- **An OS must support persistent storage.**
  - This is storage whose contents do not disappear when the system is turned off.
  - E.g.:
    - ✓ **Executable program files.**
    - ✓ **Database files.**
    - ✓ **…**
- **The primary way to do this is through a "file system" on persistent storage devices like hard disks.**
  - A set of data structures on disk and within the OS kernel memory to organize persistent data.

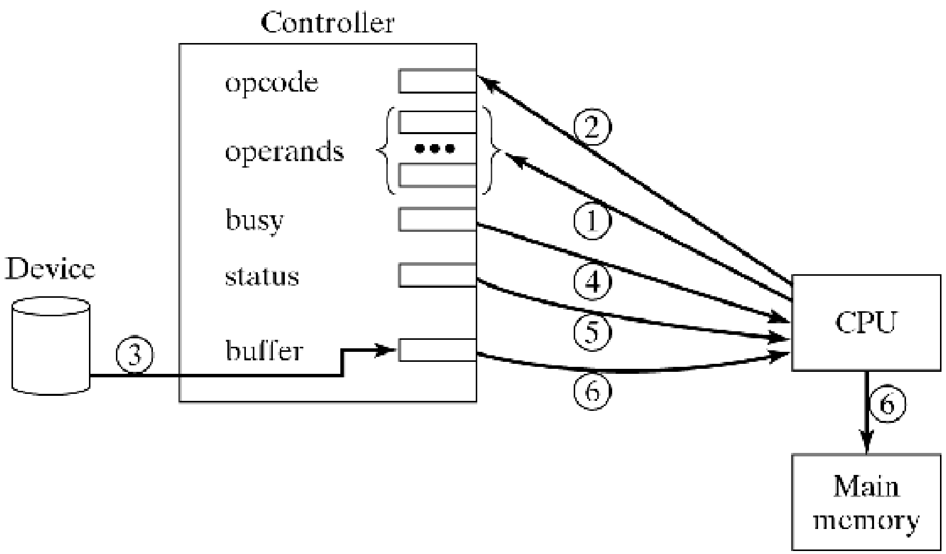
How an OS Works
File Systems

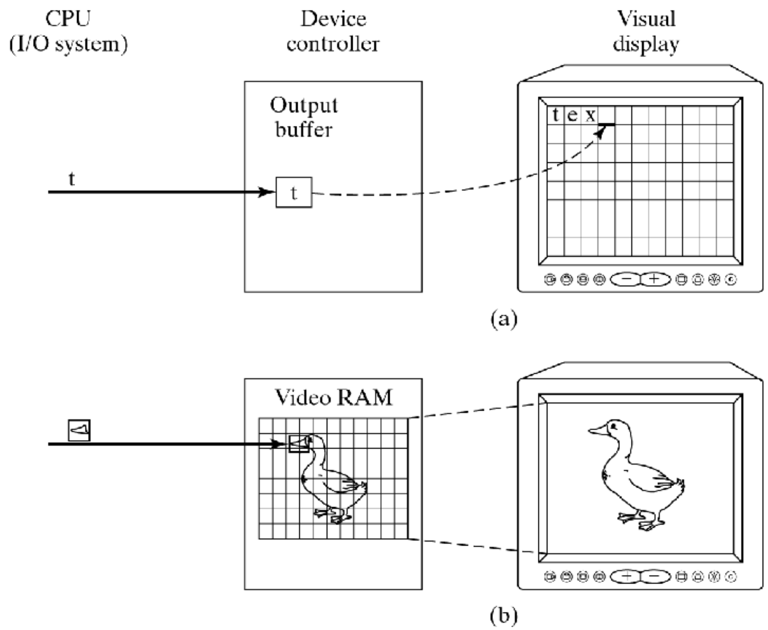


How an OS Works

Interfacing to Hardware


Applications
File system
Virtual memory management
Block device interface
open/close read/write
Stream device interface
open/close get/put io_control
Network interface
open/close read/write send/receive
I/O system interface
Block-oriented device management
Stream-oriented device management
Network communication software
device-independent software
CD-ROM driver
Hard disk driver
Keyboard driver
Printer driver
Network driver
device-dependent software
SW/HW interface
CD-ROM controller
Hard disk controller
Keyboard controller
Printer controller
Network controller
HW controllers


Controller
opcode
operands
busy
status
buffer
Device
CPU
Main memory
1
2
3
4
5
6
6


CPU (I/O system)
Device controller
Visual display
Output buffer
t
t
t e x
(a)
Video RAM
(b)

NUS National University of Singapore
School of Computing

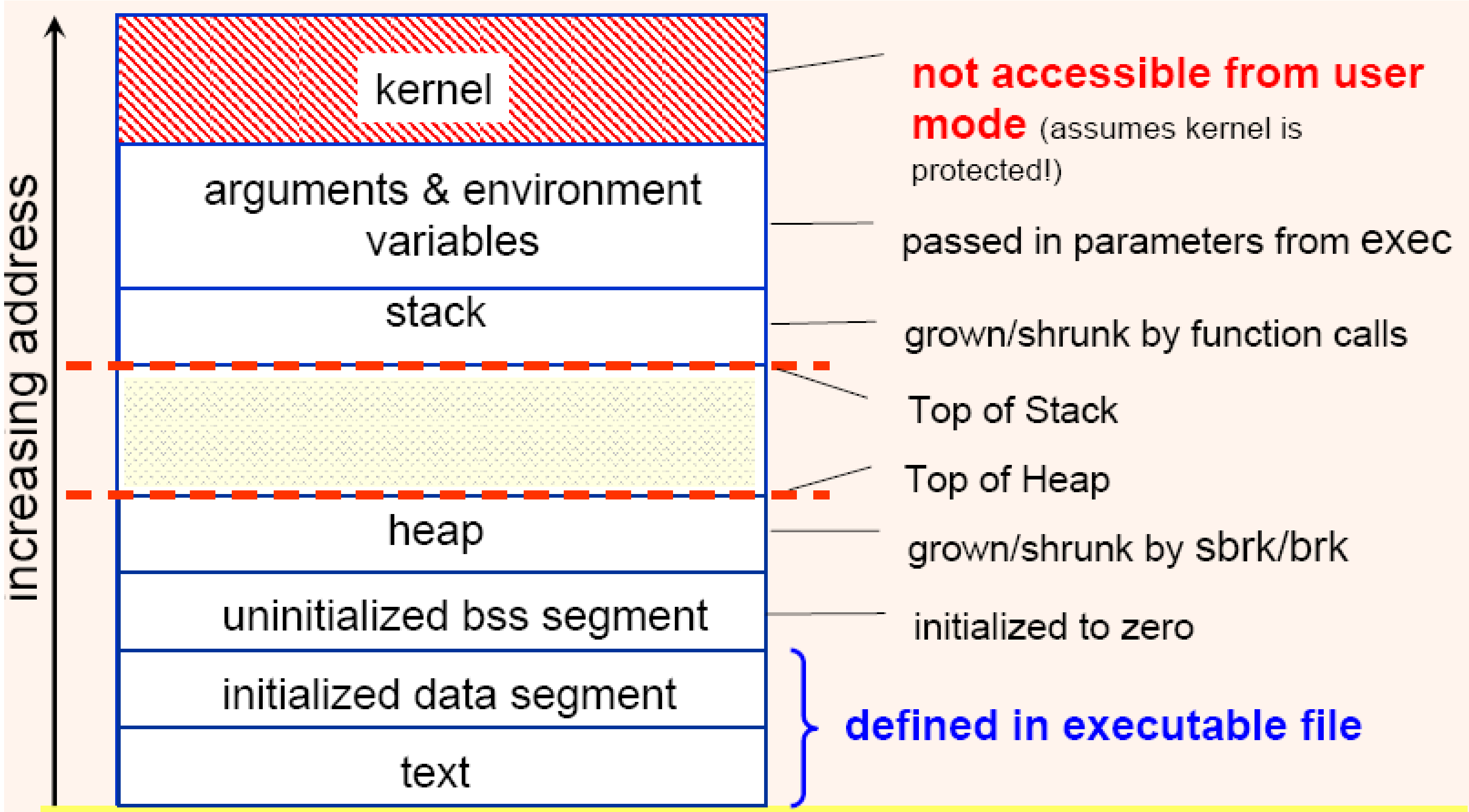
How an OS Works

Managing Memory

How an OS Works
Memory Management

- **All programs require memory to work.**
  - Memory to store instructions
  - Memory to store data.
- **The operating system must try to provide memory requested by a program.**
  - Note:
    - ✓ **We are not just talking about memory given to a program at start-up.**
    - ✓ **Programs can also ask for (and release) memory dynamically using new, delete, malloc and free.**

How an OS Works
Memory Management

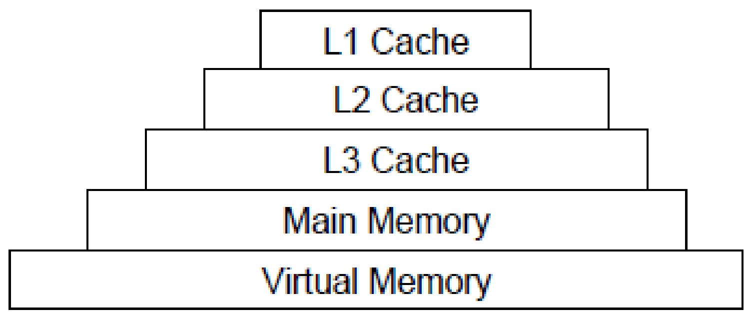


Note: unused portion (eg. between heap & stack) is not meant to be usable, e.g. segmentation fault happens when illegal memory is accessed

NUS National University of Singapore School of Computing

How an OS Works
Virtual Memory Management

- **For cost/speed reasons memory is organized in a hierarchy:**



- **The lowest level is called “virtual memory” and is the slowest but cheapest memory.**
  - Actually made using hard-disk space!
  - Allows us to fit much more instructions and data than memory allows!

How an OS Works

Securing Data

How an OS Works
Security

- **Here security means controlling access to various resources.**
  - Data (files)
    - ✓**Encryption techniques**
    - ✓**Access control lists**
  - Resources
    - ✓**Access to the hardware (biometric, passwords, etc)**
    - ✓**Memory access**
    - ✓**File access**
    - ✓**Etc.**

Writing an OS

- Historically OS written in assembler/machine code

- Modern OS: written in C/C++
  - good match to hardware (e.g. bit manipulation)
  - large parts of it become portable to other architectures

- Separation: machine independent vs machine dependent

- Implementation is difficult
  - Efficient hardware management requires complex algorithms
  - Large size
  - Difficult to modularize
  - Difficult to test/debug

Writing an OS

- **ARDOS (http://www.bitbucket.org/ctank/ardos-ide)**
  - Very small operating system built for Arduino
  - Only context switching and hardware setup code is machine dependent.
    - ✓ **Context switching code in assembly.**
    - ✓ **Hardware setup code in C but manipulates hardware registers specific to the ATmega328P.**
  - Task management, communication code etc. is completely portable.

Writing an OS

```
// Sets up SP to point to the thread stack
#define portSetStack()\
asm volatile(\
        "OUT __SP_L__, %A0      \n\t"\
        "OUT __SP_H__, %B0      \n\t": : "r" (pxCurrentTCB))

// Loads the starting address of the thread function onto the stack and
// puts in the passed parameter into R25 and R24 as expected by the function.

#if OSCPU_TYPE==AT168 || OSCPU_TYPE==AT328
        #define portPushRetAddress()\
        asm volatile(\
                "mov r0, %A0    \n\t"\
                "push r0        \n\t"\
                "mov r0, %B0    \n\t"\
                "push r0        \n\t"\
                "mov R25, %B1   \n\t"\
                "mov R24, %A1   \n\t": : "r" (pxFuncPtr), "r" (pxFuncArg))
#elif OSCPU_TYPE==AT1280 || OSCPU_TYPE==AT2560
        #define portPushRetAddress()\
        asm volatile(\
                "mov r0, %A0    \n\t"\
                "push r0        \n\t"\
                "mov r0, %B0    \n\t"\
                "push r0        \n\t"\
                "mov r0, %C0    \n\t"\
                "push r0        \n\t"\
                "mov R25, %B1   \n\t"\
                "mov R24, %A1   \n\t": : "r" (pxFuncPtr), "r" (pxFuncArg))
#endif

// Error handling
```

Writing an OS

```
void configureTimer()
{
        // Set fast PWM, OC2A and OC2B disconnected.
        TCCR2A=0b00000011;
        TCNT2=0;

        // Enable TOV2
        TIMSK2|=0b1;

}

void startTimer()
{
        // Start timer giving frequency of approx 1000 Hz
        TCCR2B=0b00000100;
        sei();
}
```

NUS National University of Singapore
School of Computing

Writing an OS

```
void OSScheduler()
{
        // Remove first item from queue
        unsigned char _nextRun=procPeek(&_ready);

        // Check to see that it is a proper process
        if(_nextRun != 255)
#if OSSCHED_TYPE==OS_PRIORITY
        if(_tasks[_nextRun].prio < _tasks[_running].prio || _forcedSwap)
#endif
        {
                _nextRun=procDeq(&_ready);
                if(_running!=255 && _nextRun !=  _running)
                {
                        _tasks[_running].sp=pxCurrentTCB;

                        // Push to READY queue if not blocked
                        if(!(_tasks[_running].status & _OS_BLOCKED))
                                procEnq(_running, _tasks, &_ready);
                }

                pxCurrentTCB=_tasks[_nextRun].sp;
                _running=_nextRun;

        }

}
```

Writing an OS (BSD Unix)

Machine independent

- 162 KLOC
- 80% of kernel
- headers, init, generic interfaces, virtual memory, filesystem, networking+protocols, terminal handling

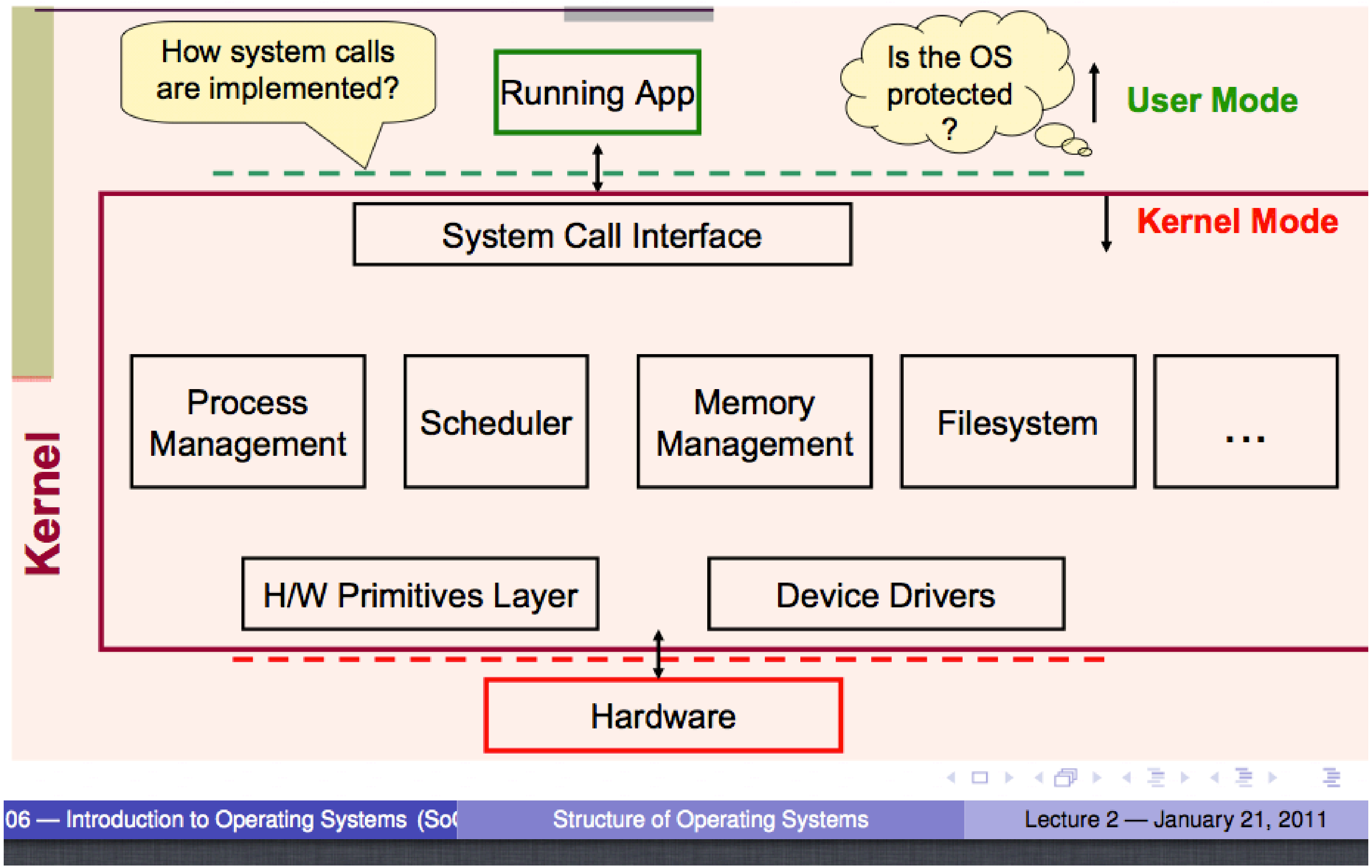
Machine dependent

- 39 KLOC
- 20% of kernel
- 3 KLOC in asm
- machine dependent headers, device drivers, VM

Monolithic Kernels

- **Kernels can be monolithic or microkernel.**
- **Monolithic kernels:**
  - All major parts of the OS – devices drivers, file systems, IPC, etc, running in "kernel space".
    - ✓ **"Kernel space" generally means an elevated execution mode where certain privileged operations are allowed.**
  - Bits and pieces of the kernel can be loaded and unloaded at runtime (e.g. using "modprobe" in Linux)
  - Popular examples of monolithic kernels: Linux, MS Windows.

Monolithic Kernels



NUS National University of Singapore School of Computing

Microkernels

- **In modular kernels:**
  - Only the "main" part of the kernel is in "kernel space":

Contains the important stuff like the scheduler, process management, memory management, etc.

  - The other parts of the kernel operate in "user space" as system services:
    - ✓**The file systems.**
    - ✓**USB device drivers.**
    - ✓**Other device drivers.**

Most famous microkernel OS: MacOS.

External View of an OS

- **The kernel itself is not very useful.**
  - Provides key functionality, but need a way to access all this functionality.
- **We need other components:**
  - System libraries (e.g. stdio, unistd, etc.)
  - System services (creat, read, write, ioctl, sbrk, etc.)
  - OS Configuration (task manager, setup, etc.)
  - System programs (Xcode, vim, etc.)
  - Shells (bash, X-Win, Windows GUI, etc.)
  - Admin tools (User management, disk optimization, etc.)
  - User applications (Word, Chrome, etc).

System Calls

- **System calls are calls made to the "Application Program Interface" or API of the OS.**
  - UNIX and similar OS mostly follow the POSIX standard.
    - ✓**Based on C.**
    - ✓**Programs become more portable.**
  - Windows follows the WinAPI standard.
    - ✓**Windows 7 and earlier provide Win32/Win64, based on C.**
    - ✓**Windows 8 provide Win32/Win64 (based on C) and WinRT (based on C++).**

NUS National University of Singapore
School of Computing

System Calls

```c
#include <unistd.h>
#include <stdio.h>
main()
{
  int pid;
  pid = getpid(); /* gets process ID */
  printf("process id = %d\n", pid);
  exit(0);
}
```

System call

library function:
also happens to make
system calls

Notes: we will in processes why exit() is not a system call

System Calls

System calls used:

- `getpid`: does actual trap to execute real `getpid` system call
- `write`: called by `printf`
- `close`: called by `exit`
- `_exit`: called by **exit**

User Mode + Kernel Mode

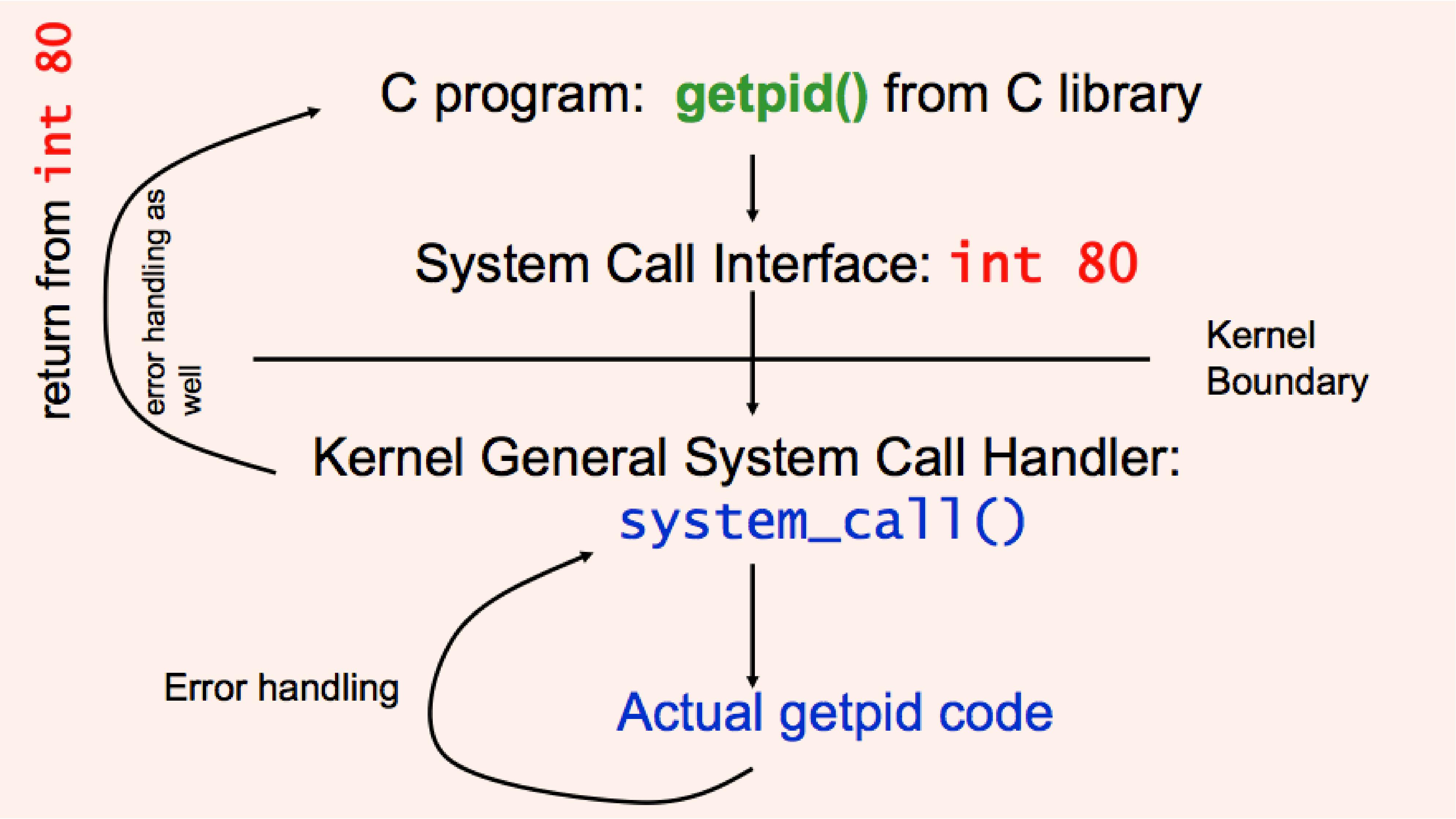
- want protection between kernel and executing program
- program (actually process) runs in user mode
- during system call – running kernel code in kernel mode
- after system call, back to user mode

How to switch modes? (processor specific)

Use privilege mode switching instructions:

- syscall instruction
- software interrupt – instruction which raises specific interrupt from software

System Calls in LINUX



NUS National University of Singapore
School of Computing

LINUX System Call

- **user mode:** (outside kernel)
  - C function wrapper (e.g. `getpid()`) for every system call in C library (not really the real system call, sometimes loosely call it a system call but **not** technically correct)
  - assembler code to setup system call no, arguments
  - trap to kernel (the real system call after all arguments setup)
- **kernel mode:** (inside kernel)
  - dispatch to correct routine
  - check arguments for errors (eg. invalid argument, invalid address, security violation)
  - do requested service
  - return from kernel trap to user mode
- **user mode:** (outside kernel)
  - returns to C wrapper – check for error return values

POSIX

- Portable Operating System Interface for uniX
- Standard IEEE 1003
- Minimal set of system calls for application portability between variants of Unix
- Linux is mostly POSIX-compliant
- Cygwin is an abstraction layer for POSIX compatibility under Win32