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.

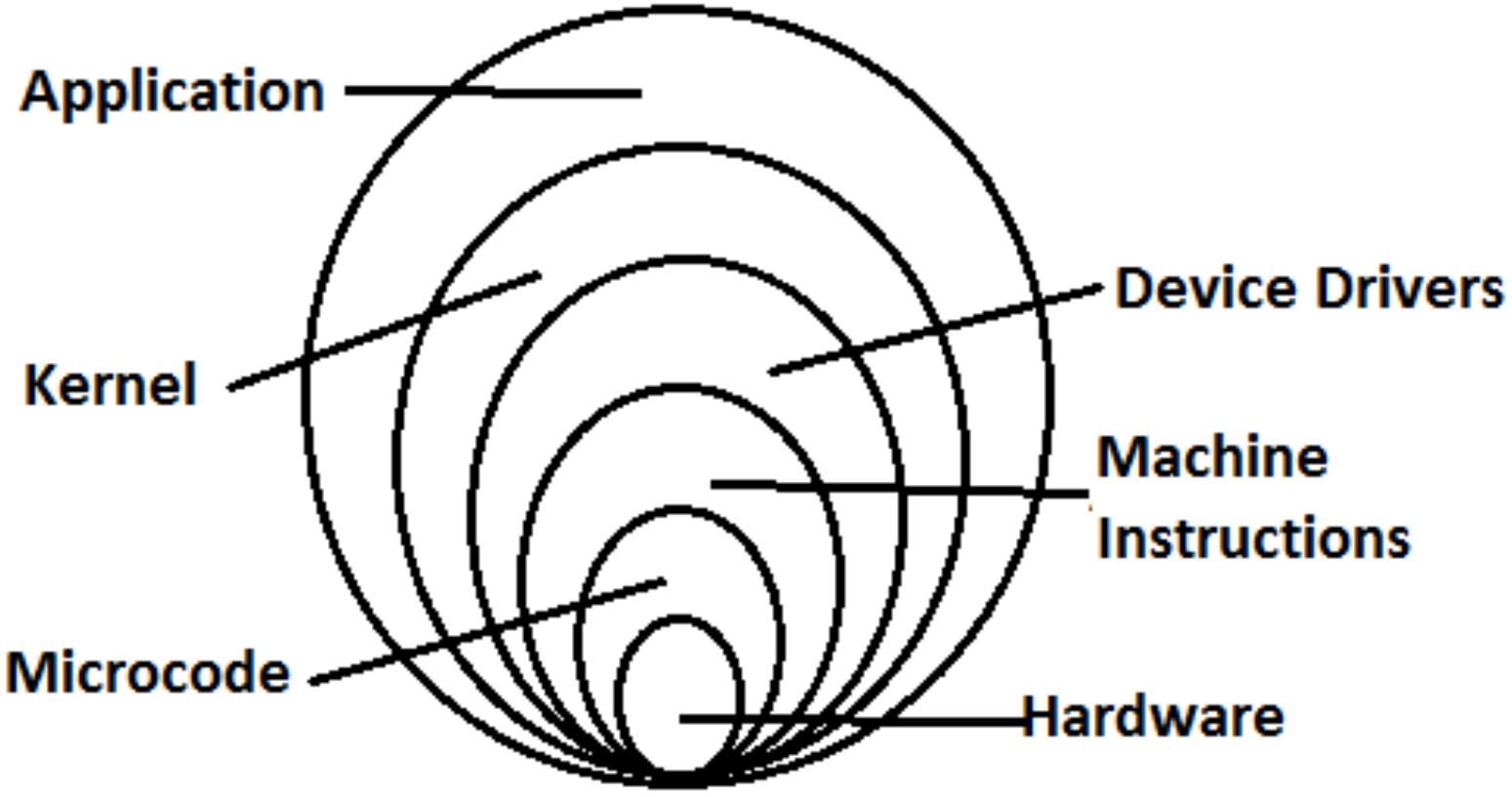
What are Operating Systems?

- **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.**

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.**

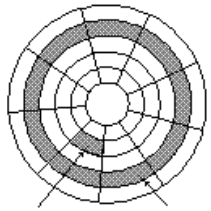
Basic Terminology: The Onion Model.



How an OS Works

Bootstrapping

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

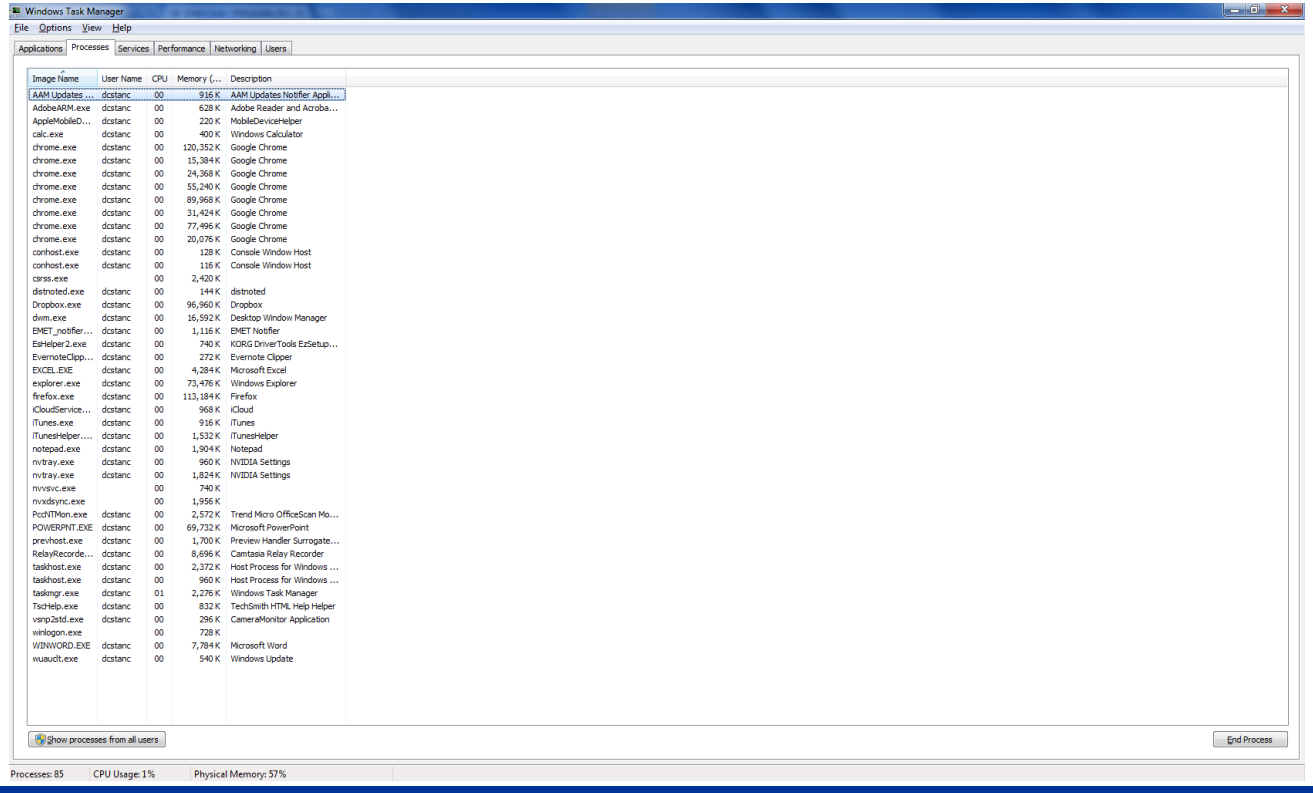
Process Management

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.

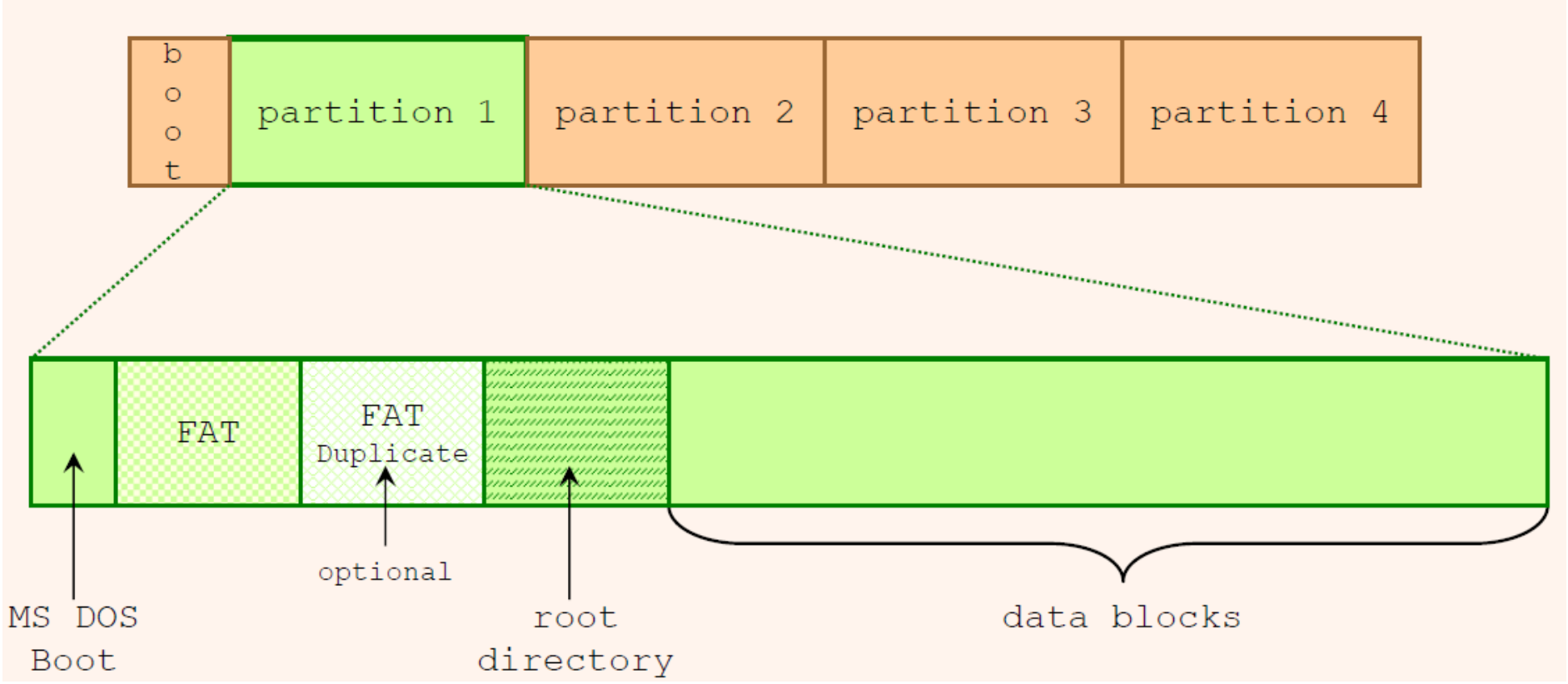
How an OS Works

Organizing Data On Your Computer

How an OS Works
File Systems

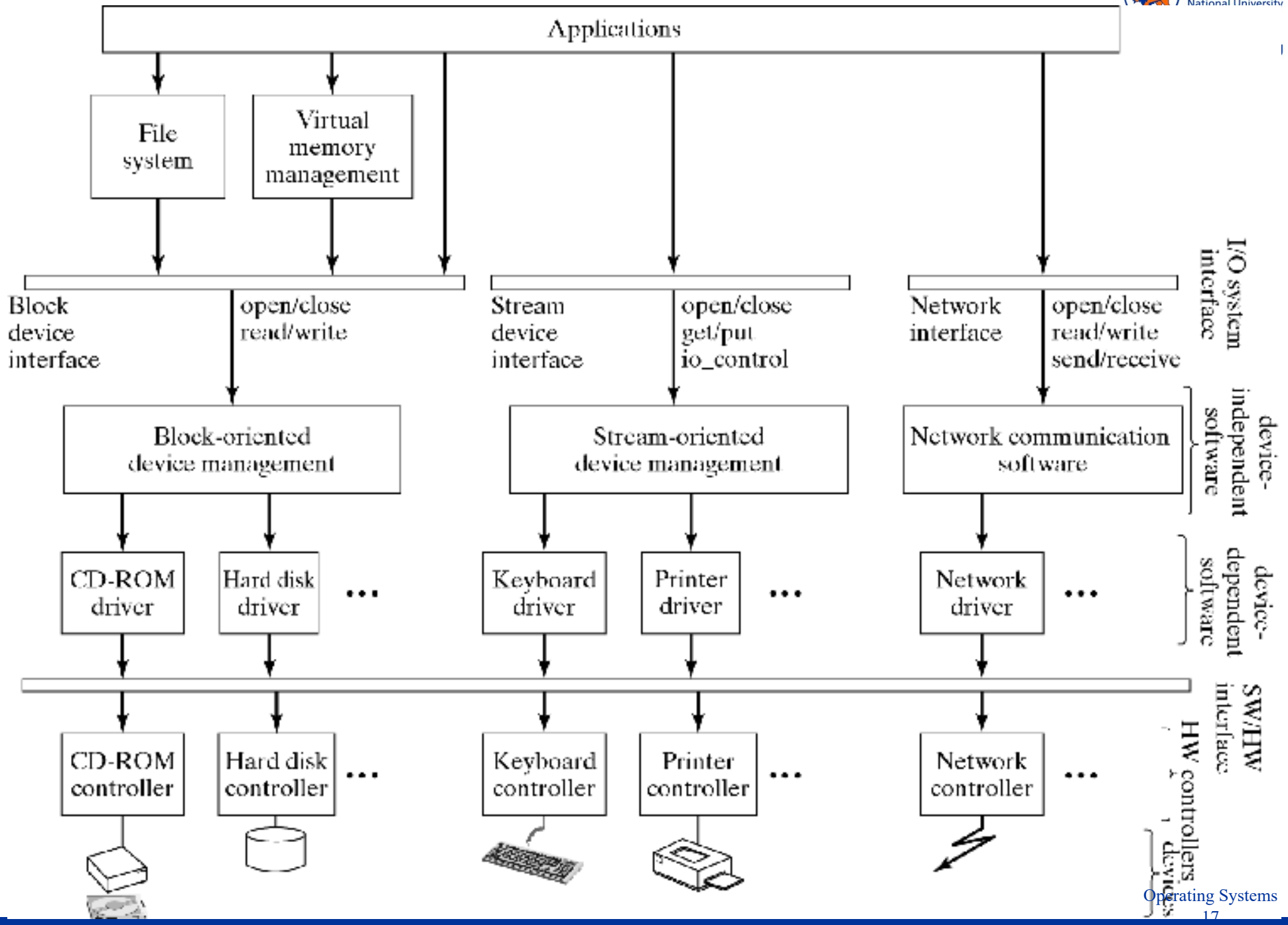
- **An OS must support persistent storage.**
  - This is storage whose contents do not disappear when the system is turned off.
  - E.g.:
    - ✓**Executable program files.**
    - ✓**Database files.**
    - ✓**…**
- **The primary way to do this is through a "file system" on persistent storage devices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

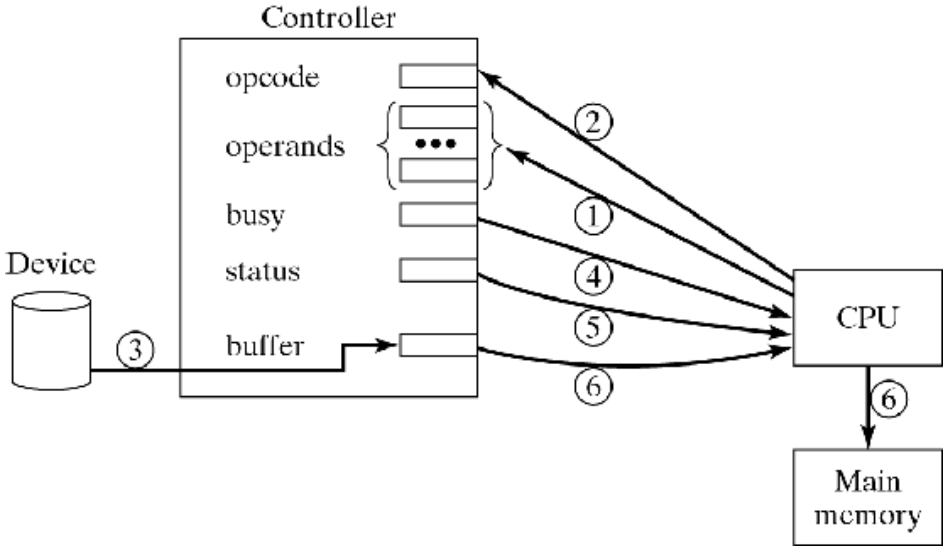
How an OS Works
File Systems

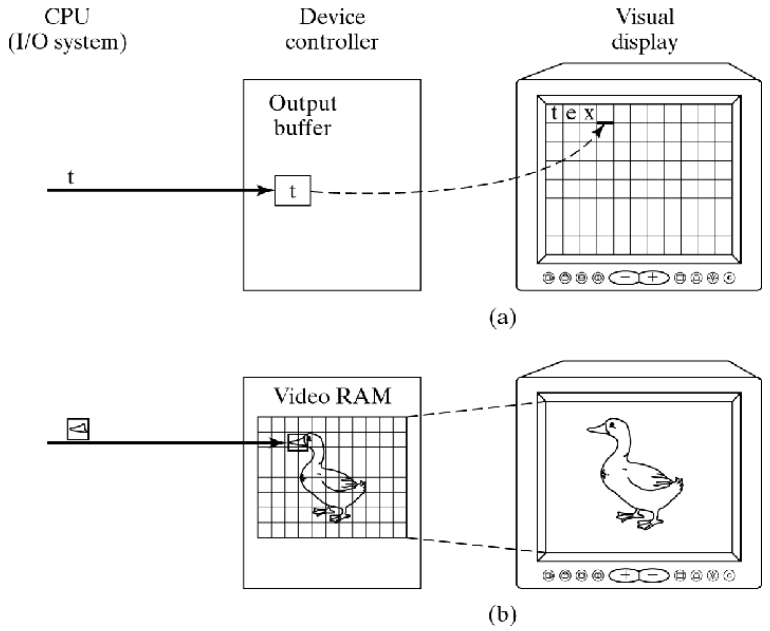


How an OS Works

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
devices


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 e x
(a)
Video RAM
(b)

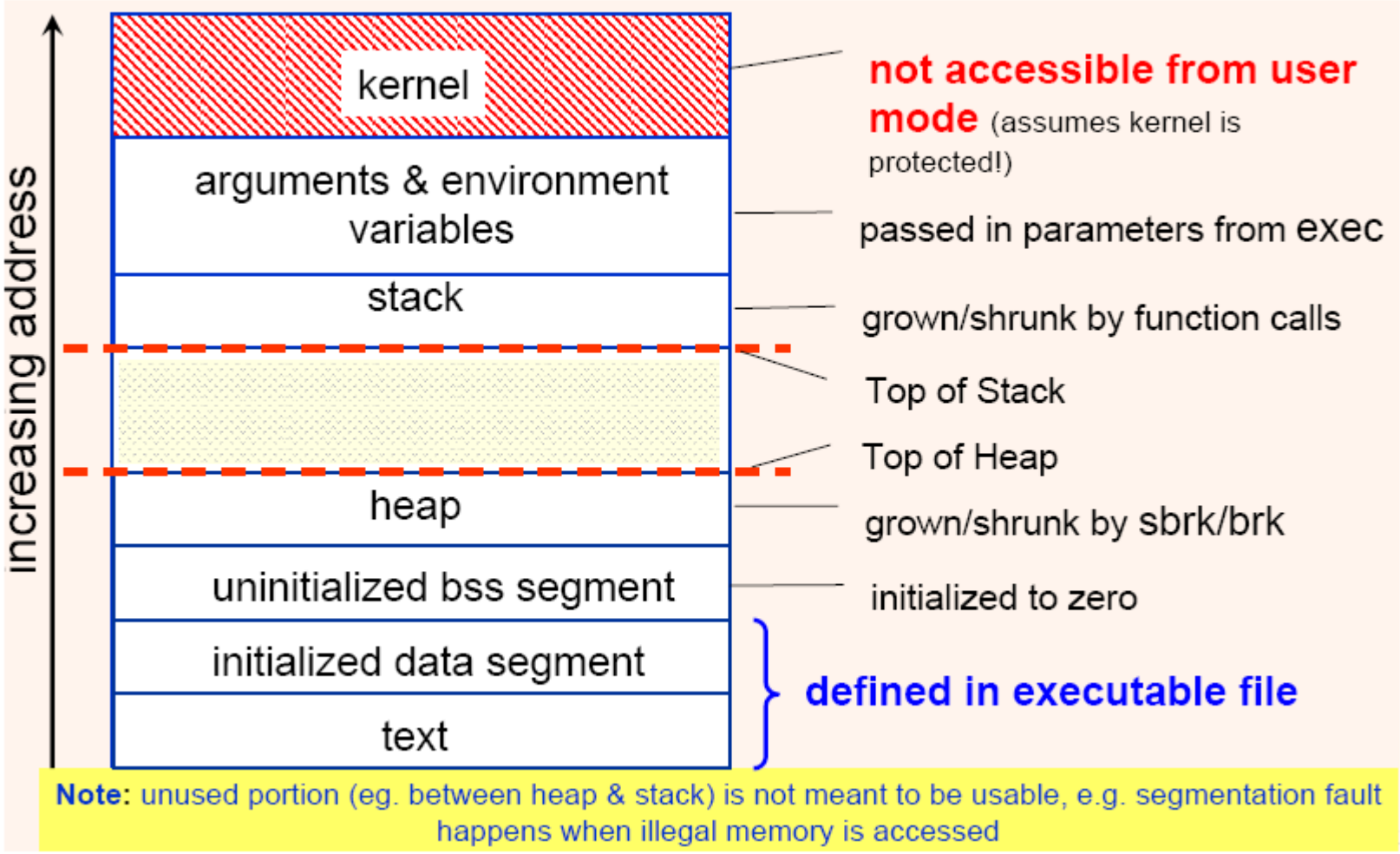
How an OS Works

Managing Memory

How an OS Works
Memory Management

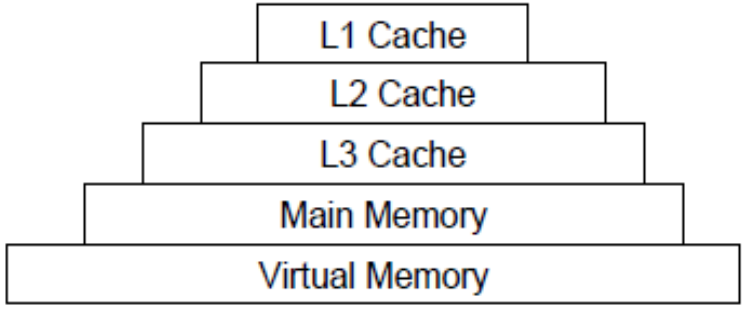
- **All programs require memory to work.**
  - Memory to store instructions
  - Memory to store data.
- **The operating system must try to provide memory requested by a program.**
  - Note:
    - ✓**We are not just talking about memory given to a program at start-up.**
    - ✓**Programs can also ask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

How an OS Works
Memory Management



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();
}
```

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

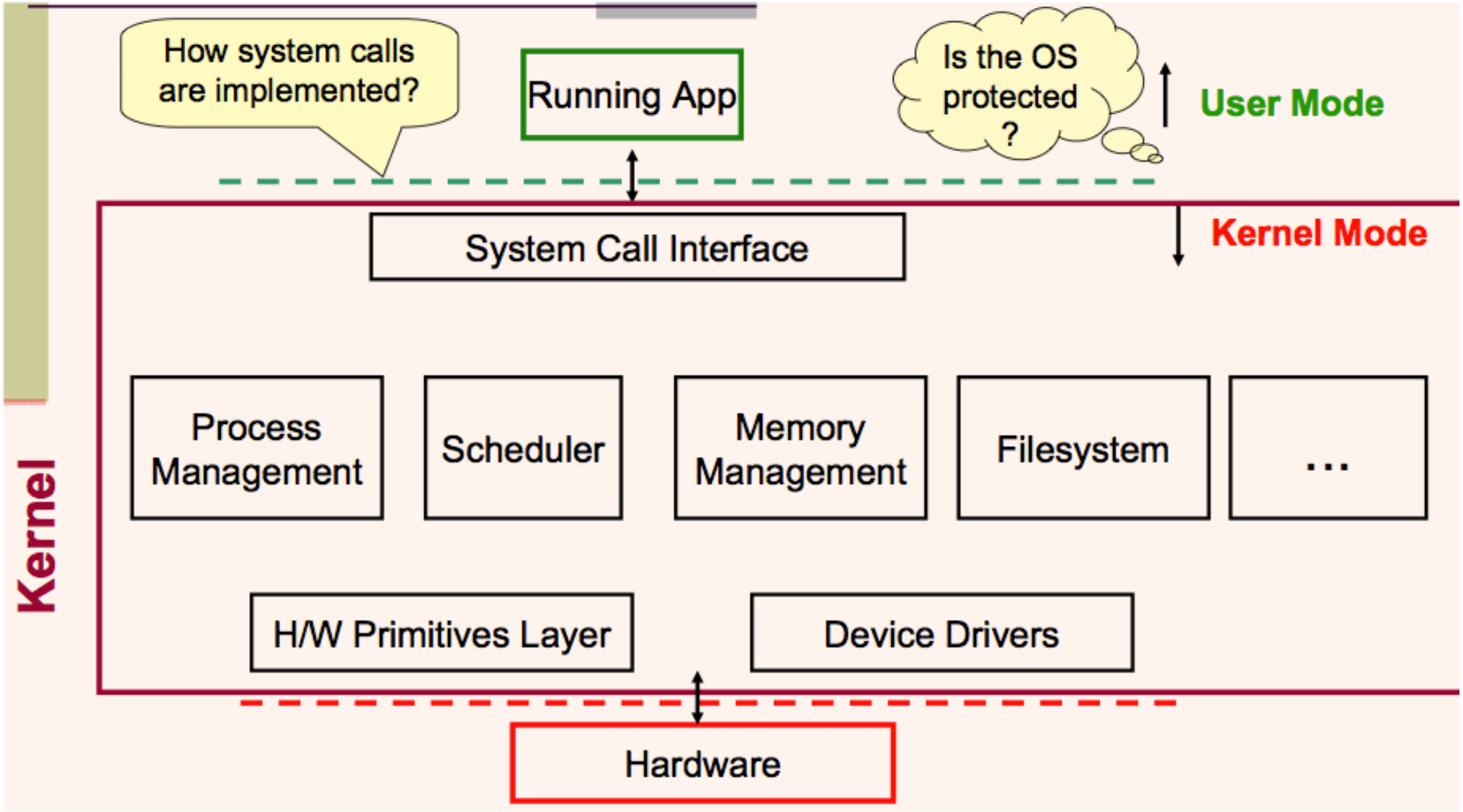
Machine dependent

- 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



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++).**

System Calls

```
#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

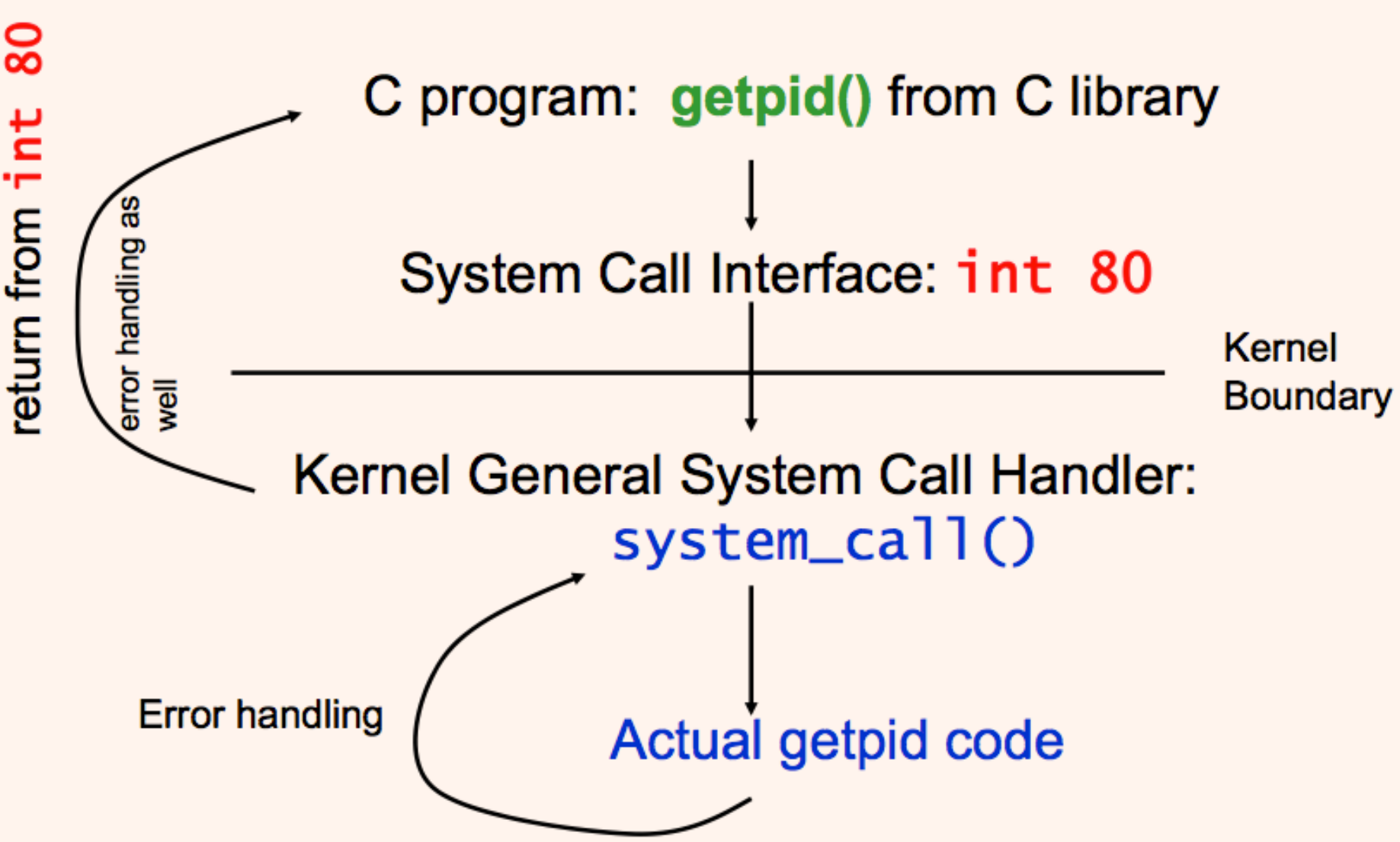
- want protection between kernel and executing program
- program (actually process) runs in user mode
- during system call – running kernel code in kernel mode
- after system call, back to user mode

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



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

IT5002

Computer Systems and Applications

Lecture 12

Process Management

colintan@nus.edu.sg

NUS
National University
of Singapore

School of Computing

Introduction

- **In this lecture we will look at:**
  - The difference between a program and a process.
  - Interrupts and How They Work
  - Process States.
  - How to run Multiple Processes in a Single CPU
  - Process Creation, termination and zombies.

Program vs. Process

- **A program consists of:**
  - Machine instructions (and possibly source code).
  - Data
  - A program exists as a file on the disk. E.g. command.exe, winword.exe
- **A process consists of:**
  - Machine instructions (and possibly source code).
  - Data.
  - Context
  - Exists as instructions and data in memory.
  - MAY be executing on the CPU.

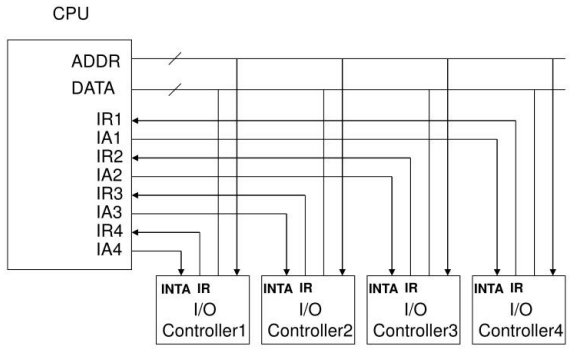
Program vs. Process

- **A single program can produce multiple processes.**
  - E.g. chrome.exe is a single program.
  - But every tab in Chrome is a new process!

Interrupts

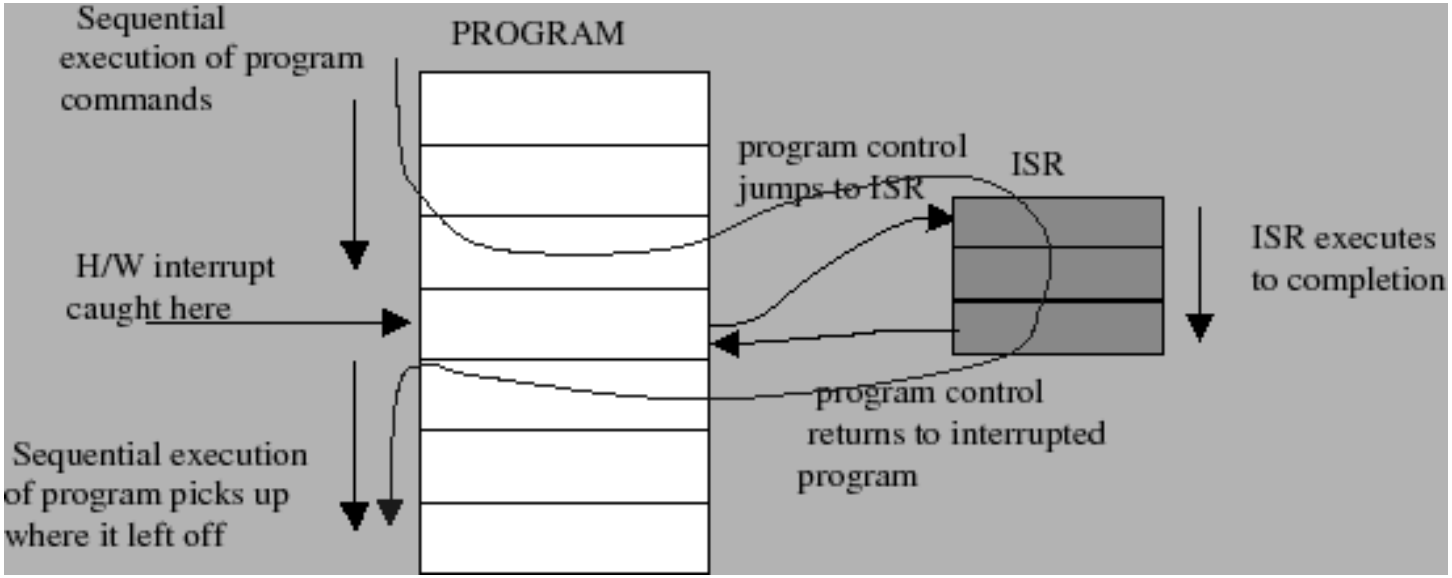
- **When a device needs attention from the CPU, it triggers what is called an "interrupt":**
  - Each device is connected to the CPU via an input called an "Interrupt Request" or IRQ line.
    - **✓When a device needs attention, it pull the IRQ line HIGH or LOW (dependin on whether line is "active high" or "active low")**

Multiple Interrupt Lines



Interrupts

- This line is checked at the end of the WB stage in the CPU Execution Cycle.
- Diagrams below show the CPU's program execution flow when an interrupt occurs:



Interrupts

- If line has been pulled, CPU interrupts the code it is currently running to run code to attend to the device:
  - ✓**Current PC is pushed onto the stack.**
  - ✓**CPU consults an "interrupt vector table" to look for the address of the "interrupt service routine" or ISR – a small bit of code that will read/write/tend to the device.**
  - ✓**This address is loaded into PC, and the CPU starts executing the handler.**
  - ✓**When the handler exits, the previous PC value is popped off the stack and back into PC, and execution resumes at the interrupted point.**

Interrupts

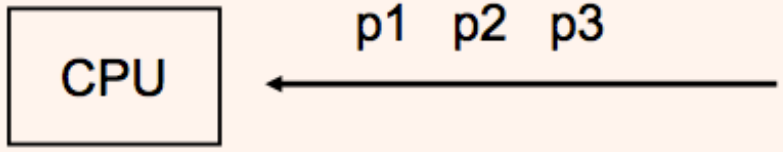
- **The CPU asserts the interrupt acknowledge (IA) line to tell the device that its request has been handled.**
  - Sometimes the CPU will de-assert the IRQ line instead of employing a separate IA line.
- **Interrupts are key to allowing us to run multiple processes on a CPU:**
  - A hardware device called a “timer” will interrupt the CPU every millisecond.
  - Interrupt handler will switch to a new process.

Execution Modes

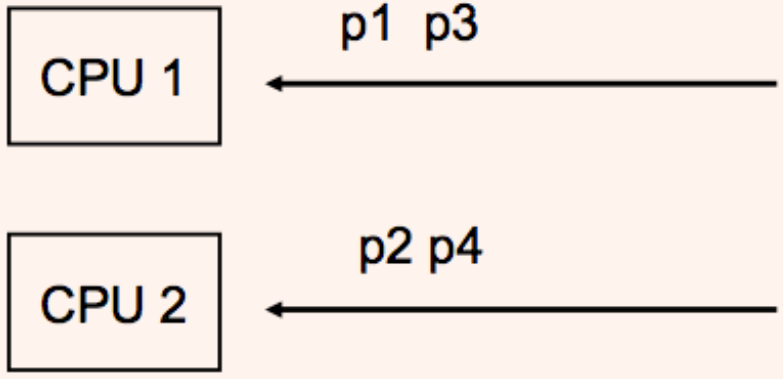
- **Programs usually run sequentially.**
  - Each instruction is executed one after the other.
- **Having multiple cores or CPUs allow parallel ("concurrent") execution.**
  - Streams of instructions with no dependencies are allowed to execute together.
- **A multitasking OS allows several programs to run "concurrently".**
  - Interleaving, or "time-slicing".

Execution Modes

- 1 CPU: timesliced execution of tasks



- multiprocessor: timeslicing on n CPUs



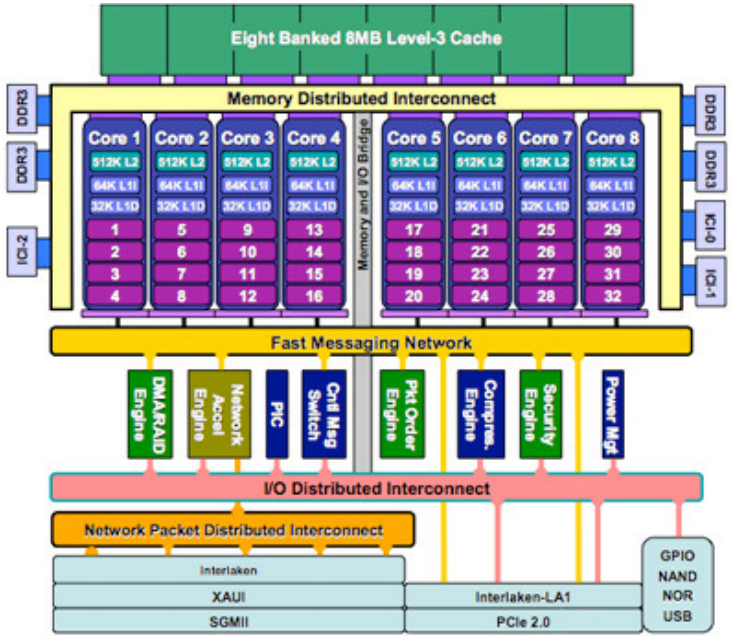
Note: we mostly assume no. processes ≥ no. of CPU otherwise can have idle task

Task Management

PROCESSES AND PROCESS MANAGEMENT

The Process Model

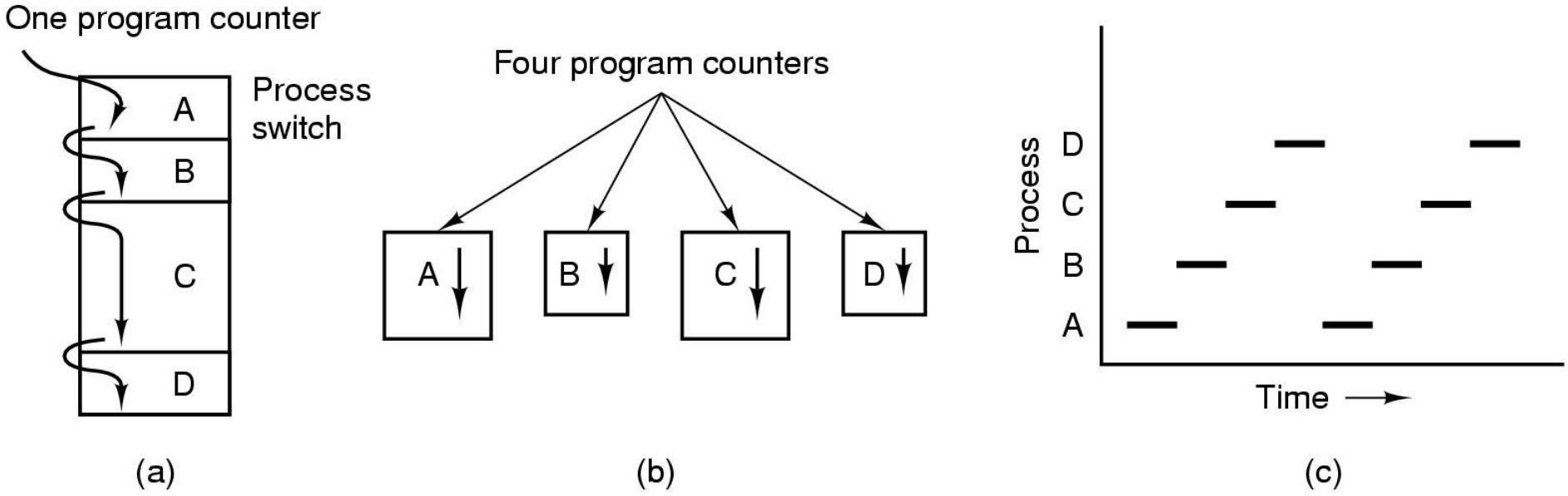
- **In this lecture we will assume a single processor with a single core.**
  - This is a legitimate assumption because in general the number of processes >> the number of cores.
  - So each core must still switch between processes.



XLP832 Processor Block Diagram

The Process Model

- **Since we have only a single-core single processor:**
  - At any one time, at most one process can execute.
  - Figures (a) to (c) below illustrate what happens:



The Process Model

- **Figure (b) shows what "appears" to be happening in a single processor system running multiple processes:**
  - There are 4 processes each with its own program counter (PC) and registers.
  - All 4 processes run independently of each other at the same time.

The Process Model

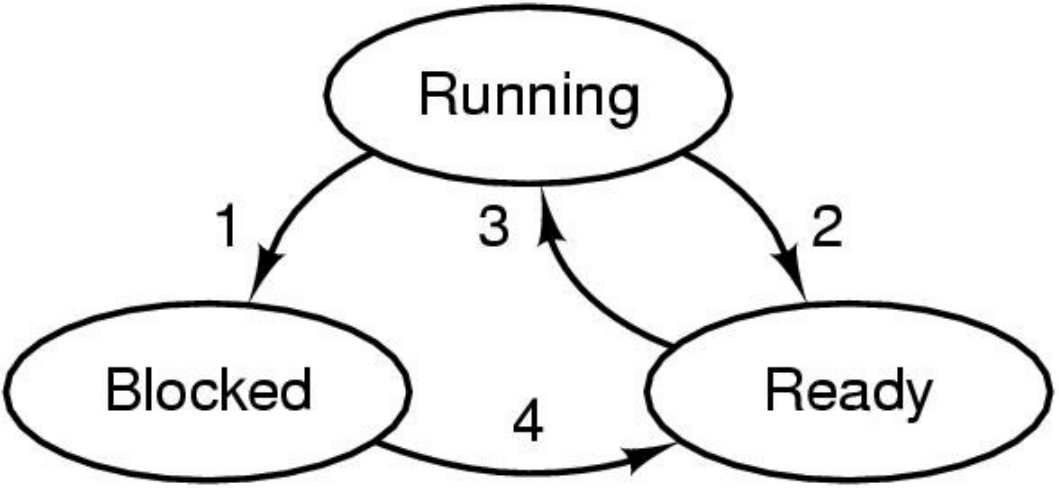
- **Figure (a) shows what actually happens.**
  - There is only a single PC and a single set of registers.
  - When one process ends, there is a "context switch" or "process switch":
    - ✓**PC, all registers and other process data for Process A is copied to memory.**
    - ✓**PC, register and process data for Process B is loaded and B starts executing, etc.**
  - Figure (c) illustrates how processes A to D share CPU time.

Process States

- **A process can be in one of 3 possible states:**
  - Running
    - ✓**The process is actually being executed on the CPU.**
  - Ready
    - ✓**The process is ready to run but not currently running.**
    - ✓**A "scheduling algorithm" is used to pick the next process for running.**
  - Blocked.
    - ✓**The process is waiting for "something" to happen so it is not ready to run yet.**
    - ✓**E.g. include waiting for inputs from another process.**

Process States

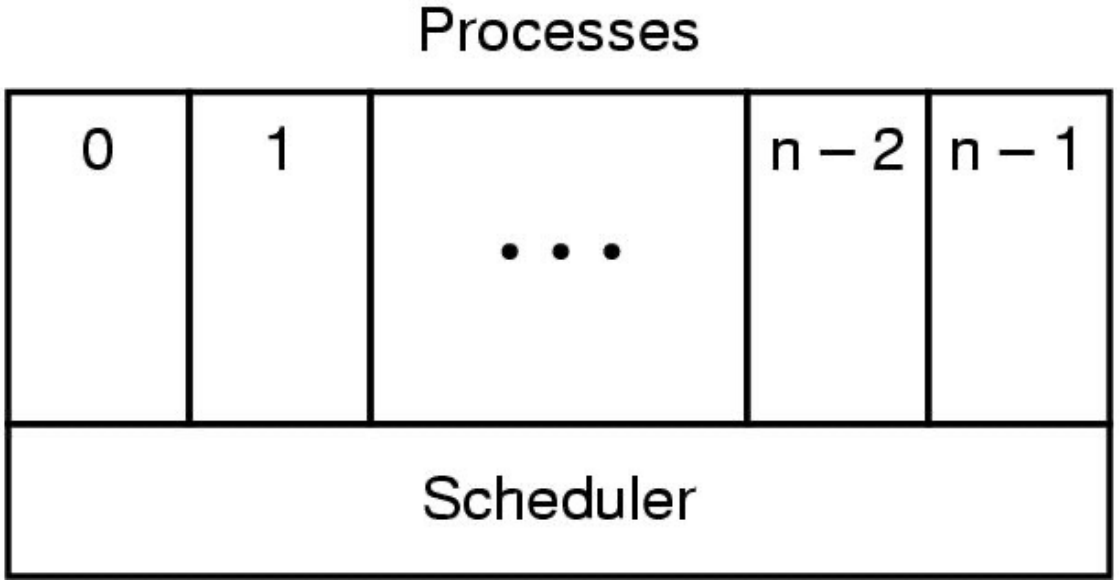
- **The diagram below shows the 3 possible states and the transitions between them.**



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process States

- **The figure below shows how the processes are organized.**
  - The lowest layer selects (schedules) which process to run next.
    - ✓**This is subject to "scheduling policies" which we will look at in a later lecture.**



Switching between Processes

- **When a process runs, the CPU needs to maintain a lot of information about it. This is called the "process context".**
  - CPU register values.
  - Stack pointers.
  - CPU Status Word Register.
    - ✓**This maintains information about whether the previous instruction resulted in an overflow or a "zero", whether interrupts are enabled, etc.**
    - ✓**This is needed for branch instructions – assembly equivalents of "if" statements.**

The AVR Status Register – SREG – is defined as:

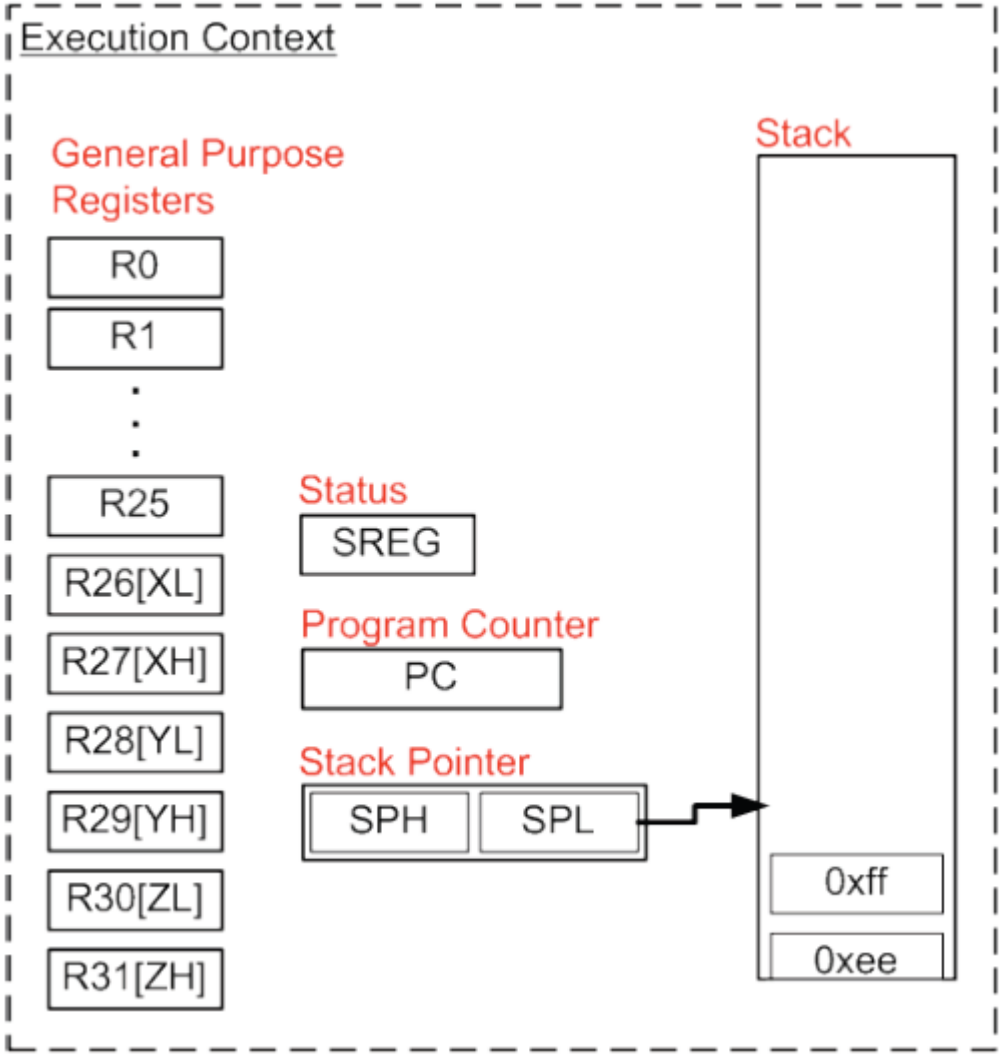
Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Switching between Processes

- **All of these values change as a process runs.**
- **When a process is blocked or put into a READY state, a new process will be picked to take control of the CPU.**
  - All the information for the current process must be saved!
  - The information for the new process must be loaded into the registers, stack pointer and status registers!
    - ✓**This is to allow the new process to run like as though it was never interrupted!**
- **This process is known as "context switching".**

Context Switching on the FreeRTOS Atmega Port

- **Each process is allocated a stack.**
  - Exactly what you learnt in IT5003
  - We use the term "task" instead of process here but means the same thing.
- **The diagram shows the complete Atmega context.**
  - Registers R0-R31, PC.
  - Status register SREG.
  - Stack pointer SPH/SPL.



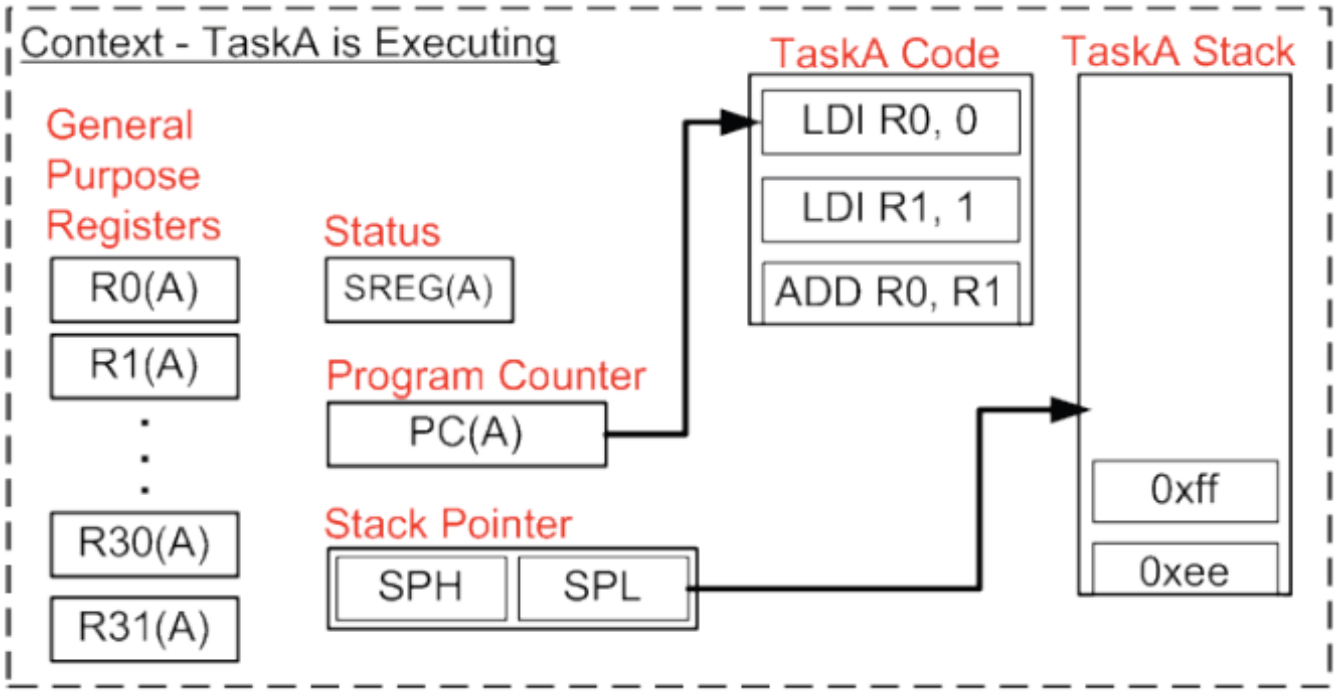
Context Switching on the FreeRTOS Atmega Port

- **FreeRTOS implements context saving in a macro called "portSAVE_CONTEXT".**

```
#define portSAVE_CONTEXT()\
asm volatile (\
      "push r0  \n\t"\              // Save R0
      "in r0, __SREG__    \n\t"\    // Read in status register SREG to R0
      "cli      \n\t"\              // Disable all interrupts for atomicity
      "push r0  \n\t"\              // Save SREG
      "push r1  \n\t"\              // Save R1
      "clr r1   \n\t"\              // AVR C expects R1 to be 0, so clear it.
      "push r2  \n\t"\              // Save R2 to R31
      ...
      "push r31 \n\t"\
      "in r26, __SP_L__   \n\t"\    // Read in stack pointer low byte
      "in r27, __SP_H__   \n\t"\    // and high byte
      "sts pxCurrentTCB+1, r27      \n\t"\    // And save it to pxCurrentTCB
      "sts pxCurrentTCB, r26        \n\t"\
      "sei      \n\t" : : :\                  // Re-enable interrupts
);
```

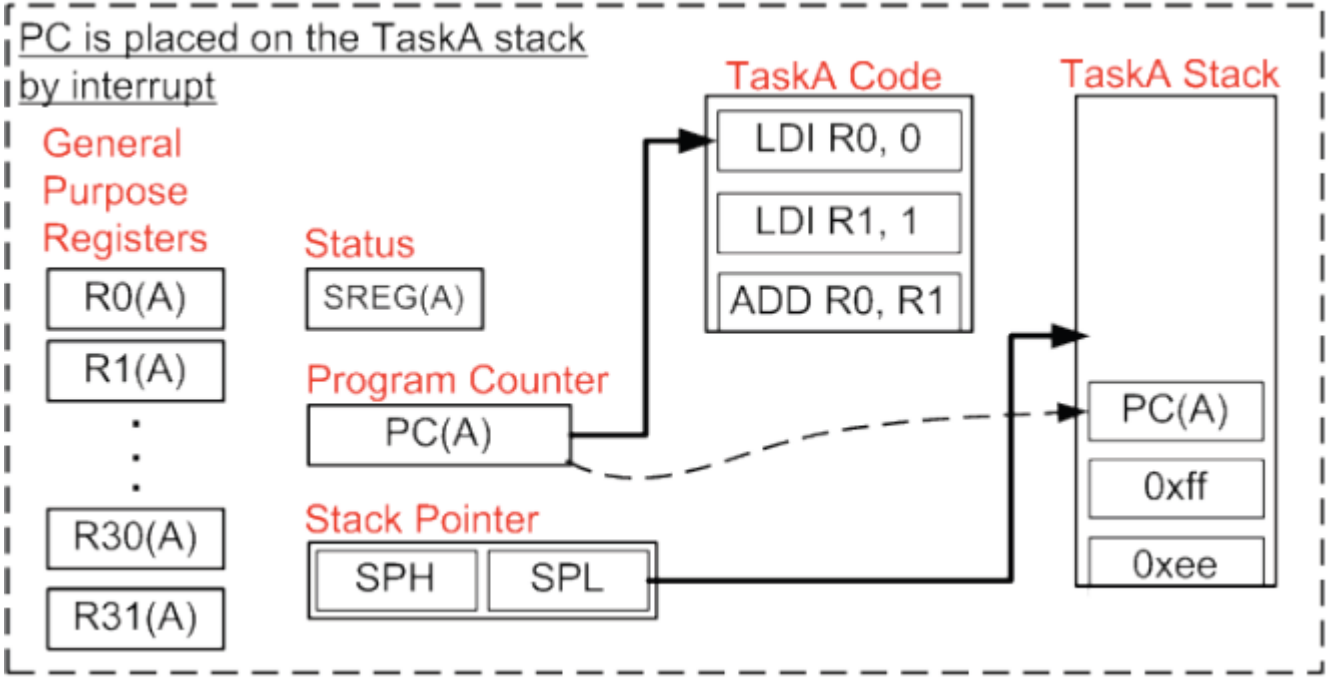
Context Switching on the FreeRTOS Atmega Port

- **We will now see step-by-step how this works.**
- **Assume that at first Task A is executing.**
  - PC would be pointing at Task A code, SPH/SPL pointing at Task A stack, Registers R0-R31 contain Task A data.



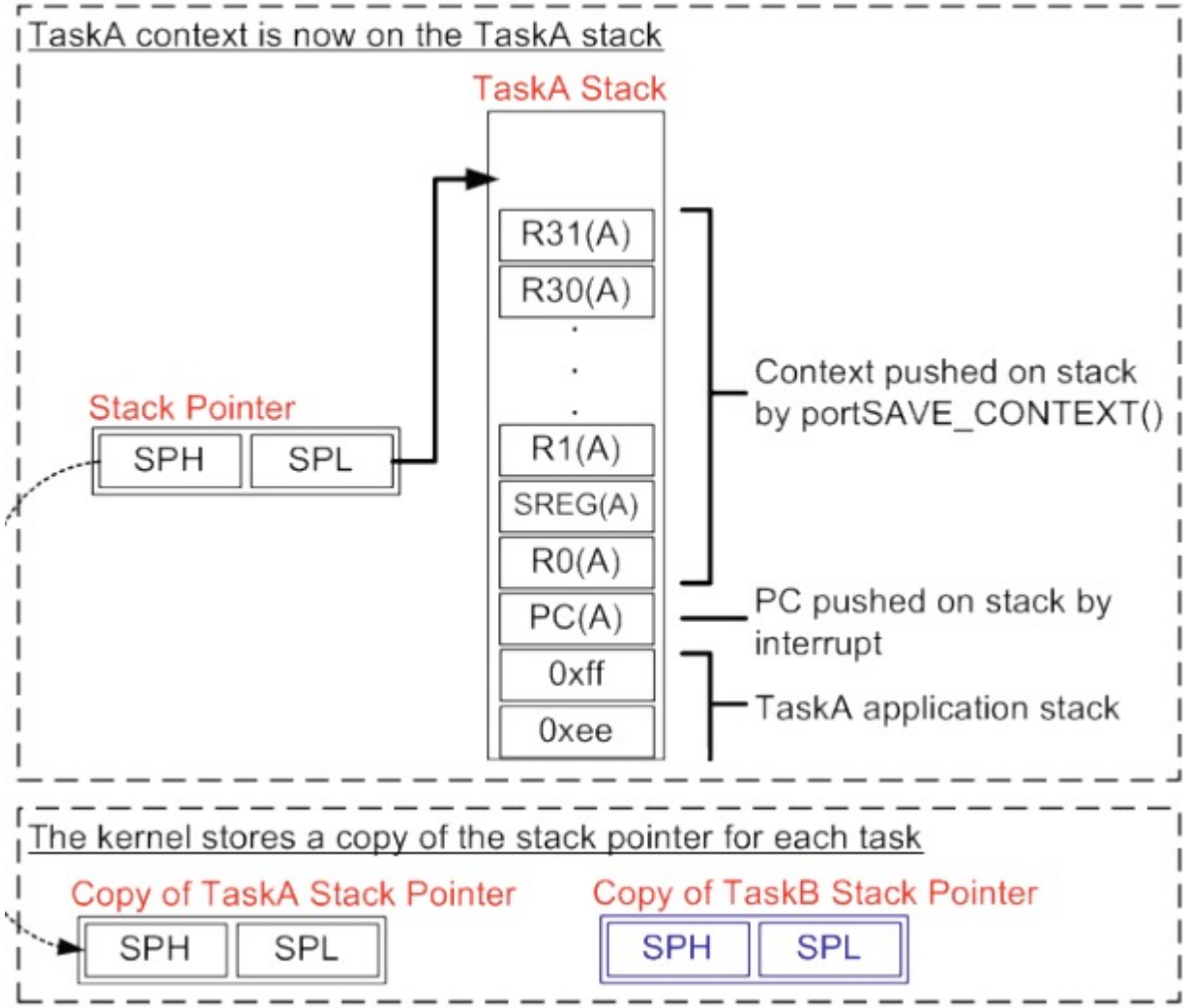
Context Switching on the FreeRTOS Atmega Port

- **FreeRTOS relies on regular interrupts from Timer 0 every millisecond to switch between tasks. When the interrupt triggers, PC is placed onto Task A's stack.**



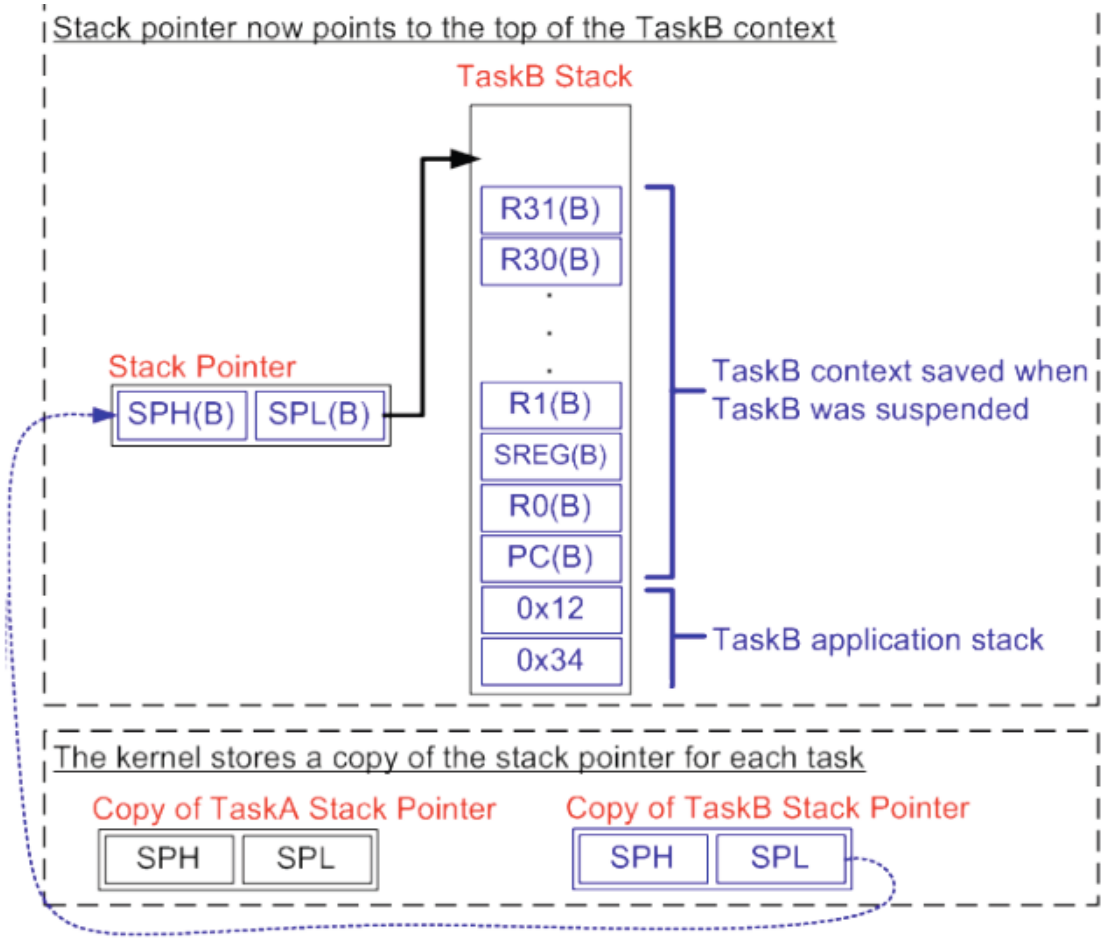
Context Switching on the FreeRTOS Atmega Port

- **The ISR calls portSAVECONTEXT, resulting in Task A's context being pushed onto the stack.**
- **pxCurrentTCB will also hold SPH/SPL _after_ the context save.**
  - This must be saved by the kernel.



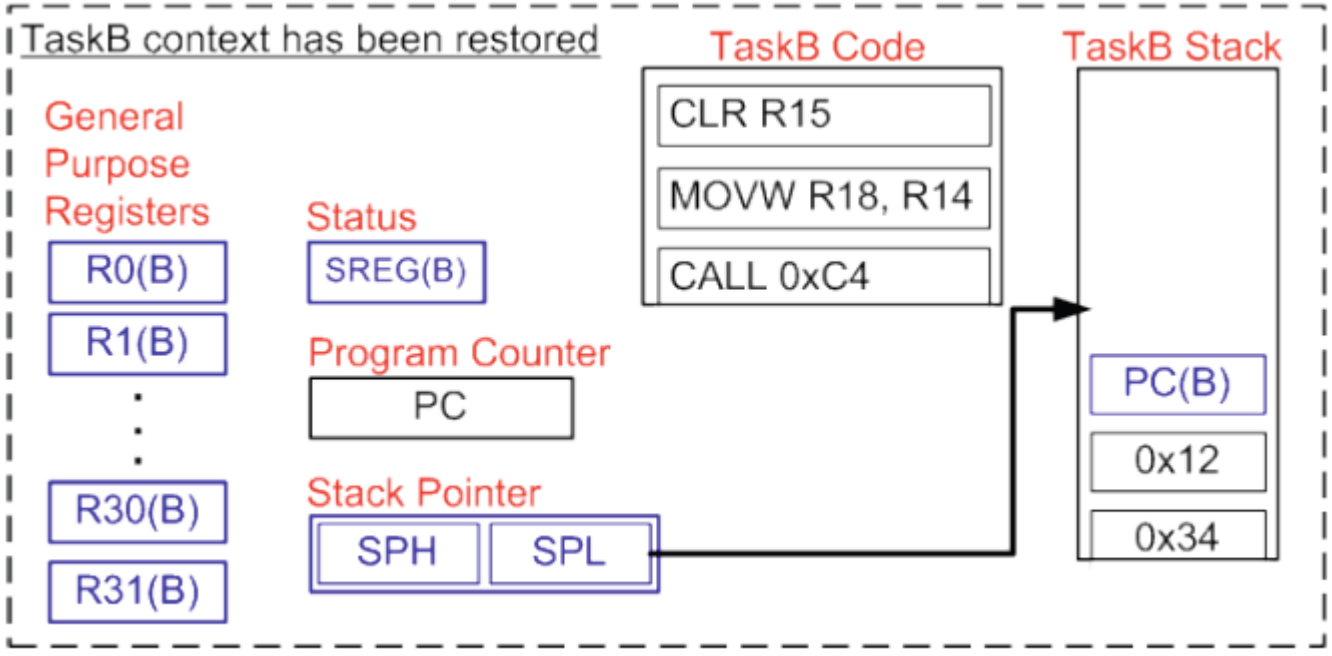
Context Switching on the FreeRTOS Atmega Port

- **The kernel then selects Task B to run, and copies its SPH/SPL values into pxCurrentTCB and calls portRESTORE_CONTEXT.**
  - The first two lines will copy pxCurrentTCB into SPH/SPL, causing SP to point to Task B's stack.



Context Switching on the FreeRTOS Atmega Port

- **The rest of portRESTORE_CONTEXT is executed, causing Task B's data to be loaded into R31-R0 and SREG.**
  - Now Task B can resume like as though nothing happened!



Only the program counter remains on the stack.

Context Switching on the FreeRTOS Atmega Port

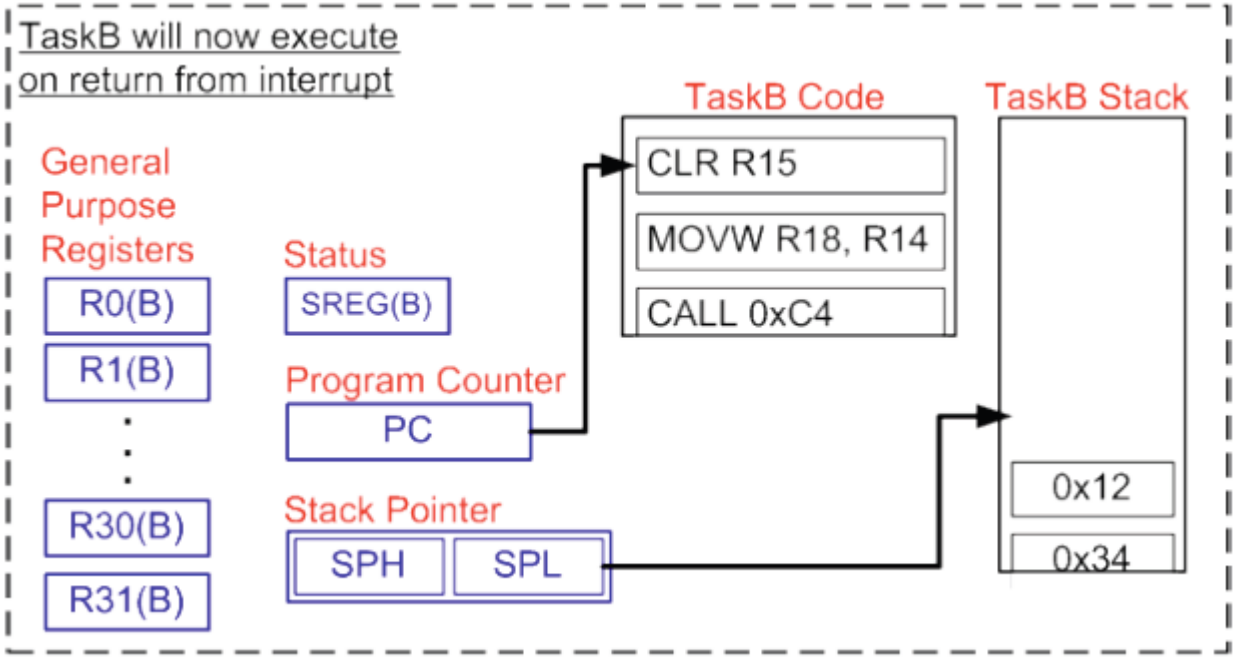
- **The reverse operation is portRESTORE_CONTEXT. The stack pointer for the process being restored must be in pxCurrentTCB.**

```
#define portRESTORE_CONTEXT()\
asm volatile (\
      "out __SP_L__, %A0 \n\t"\    // Copy SP_L and SP_H from the
      "out __SP_H__, %B0 \n\t"\    // pxCurrentTCB variable.
      "pop r31  \n\t"\             // Restore registers r31 to r1.
      ...
      "pop r0   \n\t"\             // Pop out SREG
      "out __SREG__, r0\n\t"\      // And restore it.
      "pop r0   \n\t": : "r" (pxCurrentTCB):\         // Restore R0

      );
```

Context Switching on the FreeRTOS Atmega Port

- **Only Task B's PC remains on the stack. Now the ISR exits, causing this value to be popped off onto the AVR's PC.**
  - PC points to the next instruction to be executed.
  - End result: Task B resumes execution, with all its data and SREG intact!



Context Switching on the FreeRTOS Atmega Port

- **Here we looked at context switching controlled by a timer.**
  - It can also be triggered by other things:
    - ✓**Currently running processed waiting for input.**
    - ✓**Currently running task blocking on a synchronization mechanism (see next lecture).**
    - ✓**Currently running task wants to sleep for a fixed period.**
    - ✓**Higher priority task becoming "READY".**
    - ✓**…**

Process Creation

- **A process can be created in Python by using a fork() call:**

```
# Python code to create child process
import os

def parent_child():
    n = os.fork()

    # n greater than 0  means parent process
    if n > 0:
        print("Parent process and id is : ", os.getpid())

    # n equals to 0 means child process
    else:
        print("Child process and id is : ", os.getpid())

# Driver code
parent_child()
```

- **The creating process is called a “parent” process, while the created process is called a “child” process.**

Process Creation

- **When you run a program in your OS shell, the shell uses the OS to create a new process, then run the program in the new process.**
- **In Python this is done by using subprocess.call:**

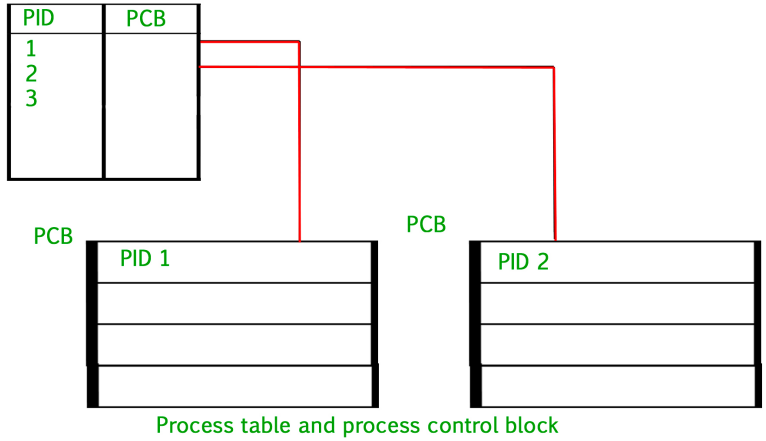
```
import subprocess

subprocess.call(["ls", "-lha"])
```

  - The launched is thus a child process of the shell.
- **Ultimately, all UNIX processes are children of "init", the main starting process in UNIX.**

Process Control Blocks

- **When a process is created, the Operating System also creates a data structure to maintain information about that process:**
  - Called a "Process Control Block" (PCB) and contains:
    - ✓**Process ID (PID)**
    - ✓**Stack Pointer**
    - ✓**Open files**
    - ✓**Pending signals**
    - ✓**CPU usage**
    - ✓**…**



  - PCB is stored in a table called a "Process Table".
    - ✓**One Process Table for entire system.**
    - ✓**One PCB per process.**

Terminating A Process

- **When a process terminates:**
  - Most resources like open files, etc., can be released and returned to the system.
  - However the PCB is retained in memory:
    - ✓**Allows child processes to return results to the parent.**
  - Parent retrieves the results using a "wait" function call, afterwhich the PCB is released.
- **What if the parent never calls "wait"?**
  - PCB remains in memory indefinitely.
  - Child becomes a "zombie". Eventually process table will run out of space and no new process can be created.

IT5002

Computer Systems and Applications

Lecture 13

Process Scheduling

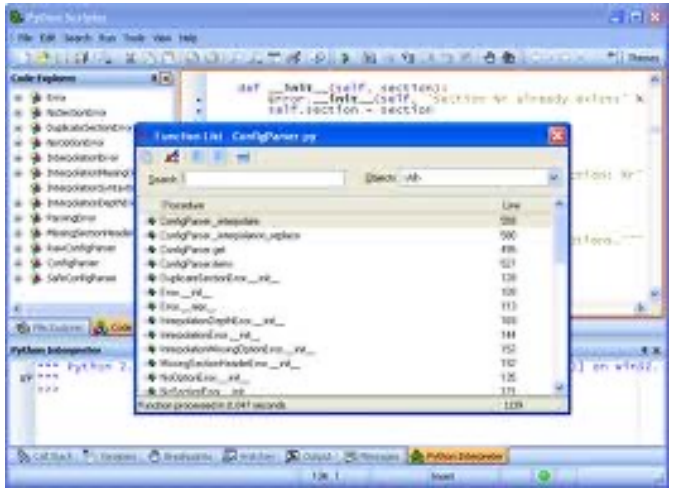
colinta@nus.edu.sg

National University of Singapore

School of Computing

What does your Computer Spend its Time Doing?

- **A mix of jobs:**
  - Computations + reading/writing memory.
  - Input/Output
    - ✓**Reading from the keyboard**
    - ✓**Writing to the screen**
    - ✓**Reading from the mouse**
    - ✓**Sending/receiving data over the network.**
    - ✓**Reading/writing the disk**
    - ✓**…**
  - How do we manage all these varied jobs?

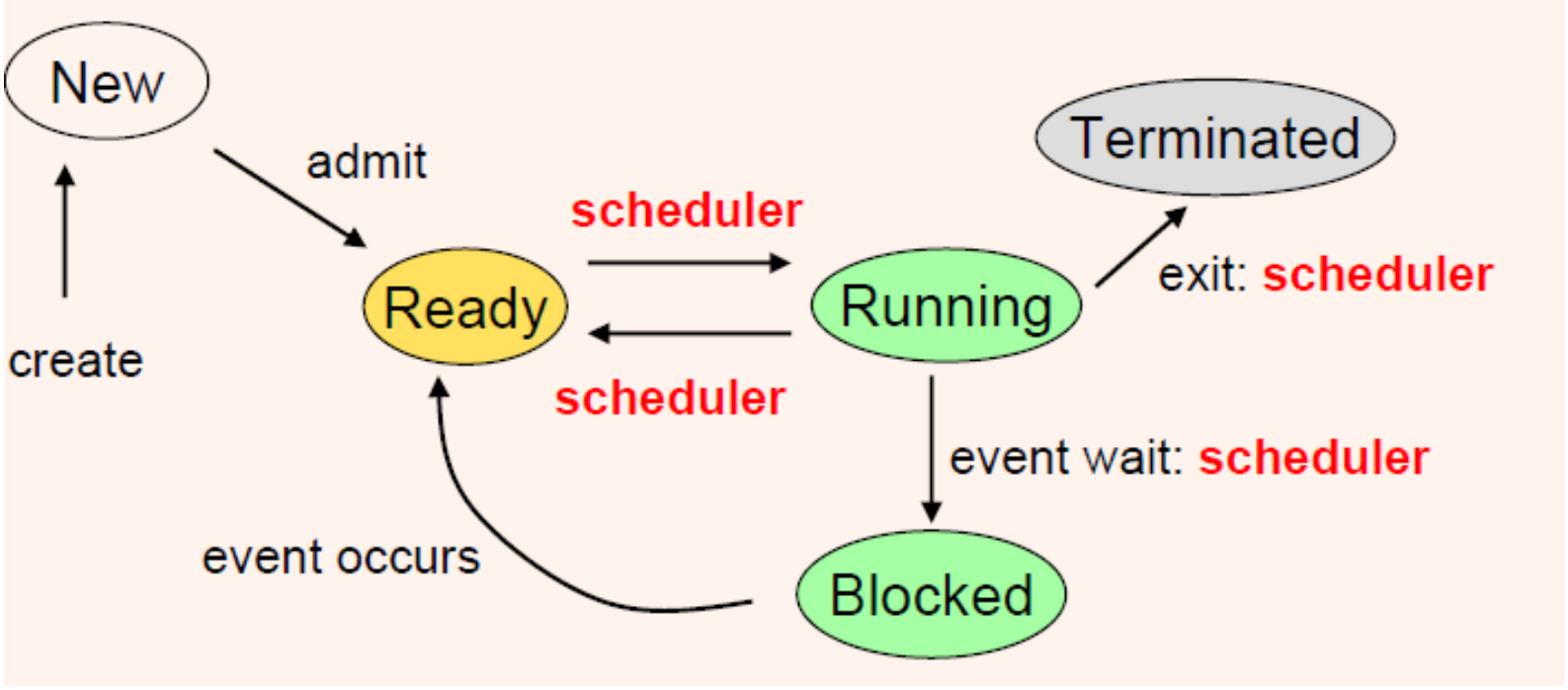


The Scheduling Environment

- **Processes can be:**
  - CPU bound
    - ✓**Most of the time spent on processing on CPU**
    - ✓**Graphics-intensive applications are considered to be "CPU" bound.**
    - ✓**Multitasking opportunities come from having to wait for processing results.**
  - I/O bound
    - ✓**Most of the time is spent on communicating with I/O devices**
    - ✓**Multitasking opportunities come from having to wait for data from I/O devices.**

Process States

- **Processes switch between a fixed set of states depending on events that take place.**
  - Scheduler is invoked at various points as shown below.



Generic Scheduler Algorithm

```
schedule() {
   while (queue not empty) {
      task = pick task from ready queue; // policy dependent
       delete task from queue;
       switch to task; // how is it this done? architecture dependent
   }
}
```

Question: How do we determine policies to pick the next task?

Types of Multitaskers

- **Policies are determined by the kind of multitasking environment.**
  - Batch Processing
    - ✓**Not actually multitasking since only one process runs at a time to completion.**
  - Co-operative Multitasking
    - ✓**Currently running processes cannot be suspended by the scheduler.**
    - ✓**Processes must volunteer to give up CPU time.**
  - Pre-emptive Multitasking
    - ✓**Currently running processes can be forcefully suspended by the scheduler.**

Types of Multitaskers

- Real-Time Multitasking
  - ✓**Processes have fixed deadlines that must be met.**
- What if we don't meet the deadlines?
  - ✓**Hard Real Time Systems: Disaster strikes! System fails, possibly catastrophically!**
  - ✓**Soft Real Time Systems: Mostly just an inconvenience. Performance of system is degraded.**

Scheduling Policies for Multitaskers

- **Scheduling Policies enforce a priority ordering over processes.**
  - As mentioned earlier, determined by multitasking type.
- **Example Policies**
  - Simplest Policy (Great for all types of multitaskers)
    - ✓**Fixed Priority**
  - Policies for Batch Processing
    - ✓**First-come First Served (FCFS)**
    - ✓**Shortest Job First (SJF)**
  - Policies for Co-operative Multitaskers
    - ✓**Round Robin with Voluntary Scheduling (VS)**

Scheduling Policies for Multitaskers

- **Example Policies**
  - Policies for Pre-emptive Multitaskers
    - ✓**Round Robin with Timer (RR)**
    - ✓**Shortest Remaining Time (SRT)**
  - Policies for Real-Time Multitaskers (Not covered)
    - ✓**Rate Monotonic Scheduling (RMS)**
    - ✓**Earliest Deadline First Scheduling (EDF)**

Fixed Priority Policy

- **This is a simple policy that can be used across any type of multitasker.**
  - Each task is assigned a priority by the programmer.
    - ✓**Usually priority number 0 has the highest priority.**
  - Tasks are queued according to priority number.
  - Batch, Co-operative:
    - ✓**Task with highest priority is picked to be run next.**
  - Pre-emptive, Real-Time:
    - ✓**When a higher priority task becomes ready, current task is suspended and higher priority task is run.**

Batch Scheduling Policies

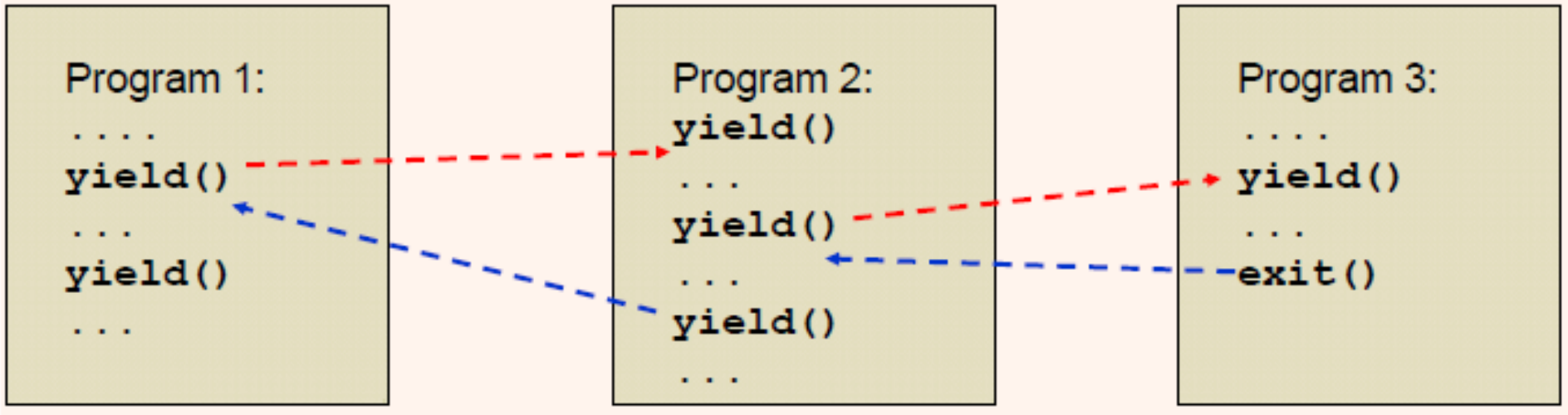
- **First Come First Served**
  - Arriving jobs are stored in a queue.
  - Jobs are removed in turn and run.
  - Particularly suited for batch systems.
  - Extension for interactive systems:
    - ✓**Jobs removed for running are put back into the back of the queue.**
    - ✓**This is also known as "round-robin scheduling"**
  - Starvation free as long as earlier jobs are bounded.

Batch Scheduling Policies

- **Shortest Job First**
  - Processes are ordered by total CPU time used.
  - Jobs that run for less time will run first.
  - Reduces average waiting time if number of processes is fixed.
  - Potential for starvation.

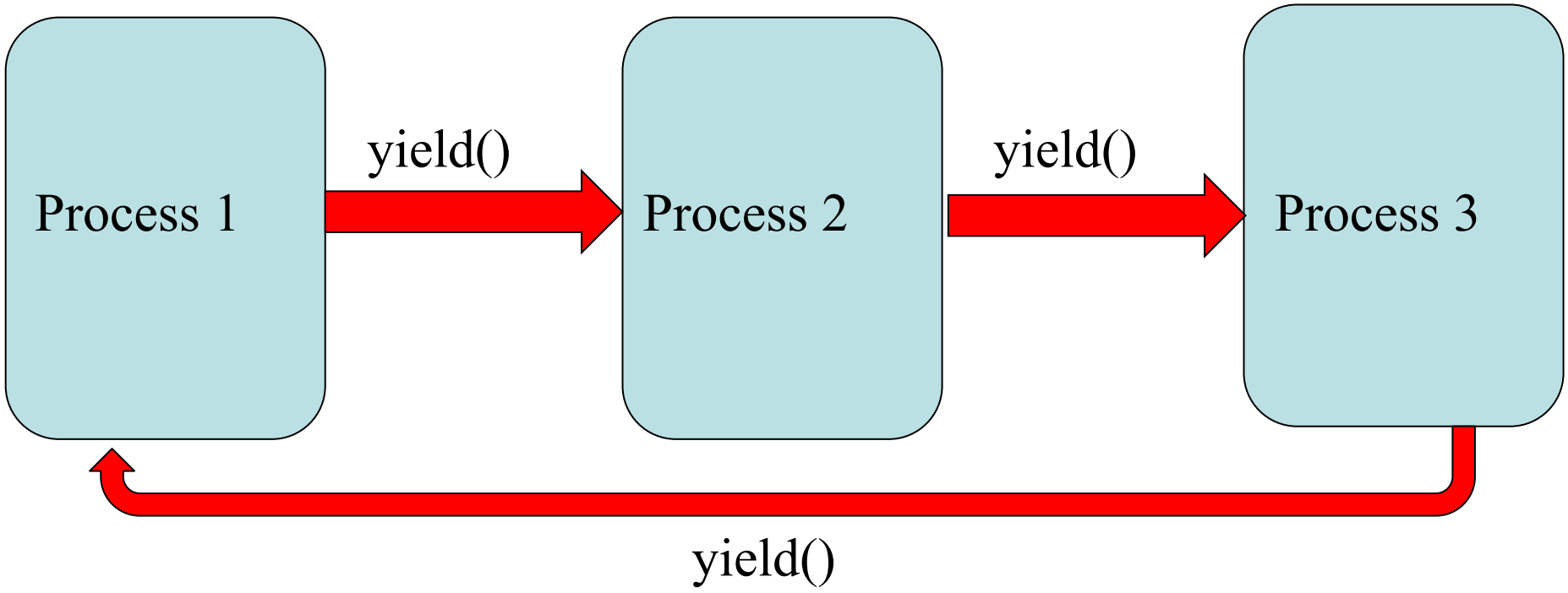
Co-operative Scheduling Policies

- **Voluntary Scheduling.**
  - Processes call a special "yield" function.
    - ✓**This invokes the scheduler.**
    - ✓**Causes the process to be suspended and another process started up.**



Co-operative Scheduling Policies

- **In many systems VS is used with a round-robin arrangement.**



Pre-emptive Scheduling Policies

- **Shortest Remaining Time**
  - Pre-emptive form of SJF.
  - Processes are ordered according to remaining CPU time left.
- **Round-robin with Timer**
  - Each process is given a fixed time slot c_i.
  - After time c_i, scheduler is invoked and next task is selected on a round-robin basis.

Managing Multiple Policies

- **Multiple policies can be implemented on the same machine using multiple queues:**
  - Each queue can have its own policy.
  - This scheme is used in Linux, as we will see shortly.

high priority:	P1, P3	(RR policy)
medium priority:	P2	(RR policy)
low priority:	P4, P5	(batch queue, FCFS policy)

Scheduling in Linux

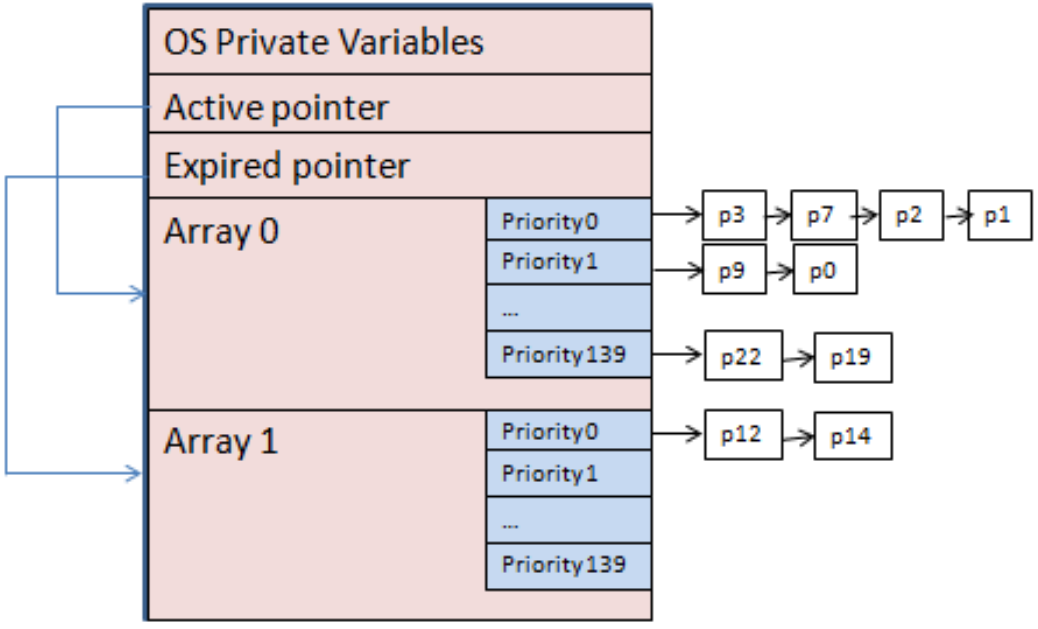
- **Processes in Linux are dynamic:**
  - New processes can be created with fork()
  - Existing processes can exit.
- **Priorities are also dynamic:**
  - Users and superusers can change priorities using "nice" values.
  - nice –n 19 tar cvzf archive.tgz *
    - ✓**Allows tar to run with a priority lowered by 19 to reduce CPU load.**
    - ✓**Normal users can only $0 \leq n \leq 19$**
    - ✓**Superusers can specify $-20 \leq n \leq 19$. Negative nice increases priority.**

Scheduling in Linux

- **Linux maintains three types of processes:**
  - Real-time FIFO:
    - ✓**RT-FIFO processes cannot be pre-empted except by a higher priority RT-FIFO process.**
  - Real-time Round-Robin:
    - ✓**Like RT-FIFO but processes are pre-empted after a time slice.**
  - Linux only has "soft real-time" scheduling.
    - ✓**Cannot guarantee deadlines, unlike RMS and EDF we saw earlier.**
    - ✓**Priority levels 0 to 99**
  - Non-real time processes
    - ✓**Priority levels 100 to 139**

Scheduling in Linux

- **Linux maintains 280 queues in two sets of 140:**
  - An active set.
  - An expired set.



Scheduling in Linux

- **The scheduler is called at a rate of 1000 Hz.**
  - E.g. time tick is 1 ms, called a "jiffy".
  - RT-FIFO processes are always run if any are available.
  - Otherwise:
    - ✓**Scheduler picks highest priority process in active set to run.**
    - ✓**When its "time quantum" is expired, it is moved to the expired set. Next highest priority process is picked.**
    - ✓**When active set is empty, active and expired pointers are swapped. Active set becomes expired set and vice versa.**
    - ✓**Scheme ensures no starvation of lowest priority processes.**

Scheduling in Linux

- **What happens if a process becomes blocked? (e.g. on I/O)**
  - CPU time used so far is recorded. Process is moved to a queue of blocked processes.
  - When process becomes runnable again, it continues running until its time quantum is expired.
  - It is then moved to the expired set.
- **When a process becomes blocked its priority is often upgraded (see later).**

Scheduling in Linux

- **Time quantums for RR processes:**
  - Varies by priority. For example:
    - ✓**Priority level 100 – 800 ms**
    - ✓**Priority level 139 – 5 ms**
    - ✓**System load.**
- **How process priorities are calculated:**
  - Priority = base + f(nice)+g(cpu usage estimate)
    - ✓**f(.) = priority adjustment from nice value.**
    - ✓**g(.) = Decay function. Processes that have already consumed a lot of CPU time are downgraded.**

Scheduling in Linux

- Other heuristics are used:
  - ✓**Age of process.**
  - ✓**More priority for processes waiting for I/O – I/O boost.**
  - ✓**Bias towards foreground tasks.**
- **I/O Boost:**
  - Rationale:
    - ✓**Tasks doing read() has been waiting for a long time. May need quick response when ready.**
    - ✓**Blocked/waiting processes have not run much.**
    - ✓**Applies also to interactive processes – blocked on keyboard/mouse input.**

Scheduling in Linux

- Implementation: We can -
  - ✓**Boost time quantum.**
  - ✓**Boost priotiy.**
  - ✓**Do both.**
- How long does this boost last?
  - ✓**Temporary boost for sporadic I/O**
  - ✓**Permanent boost for the chronically I/O bound?**
  - ✓**E.g. Linux gives -5 boost for interactive processes.**

IT5002

Computer Systems and Applications

Lecture 14

Inter-Process Communication

colintan@nus.edu.sg

NUS
National University
of Singapore

School of Computing

Learning Objectives

- **By the end of this lecture you will be able to:**
  - Understand what race conditions are, and why they are bad.
  - Understand the various ways to prevent race conditions.
  - Understand how to pass messages between processes.
- **Some example code are taken from ArdOS, the Arduino Operating System.**
  - Simple kernel code.
  - Obtainable from http://www.bitbucket.org/ctank/ardos-ide

Introduction

- **In the lecture we looked at how multiple processes can run on a single CPU.**
- **In real-world applications, there are "dependencies" between processs.**
  - Process B cannot proceed because it is waiting for Process A's result.
  - Process B and Process A update the same shared variable, which can result in errors.
  - …

Introduction

- **If both Process A and Process B are allowed to run freely, errors will occur.**
  - Process B proceeds before Process A completes, resulting in B using stale results.
  - Process A and B update a variable at the same time, causing one process to over-write the results of the other process.
  - Etc.
- **Some form of coordination is therefore required!**
- **Material from this course comes from different places.**
  - Some books use “process” and some use “task”.
  - Just remember that they mean the same thing.

Inter-Process Communications

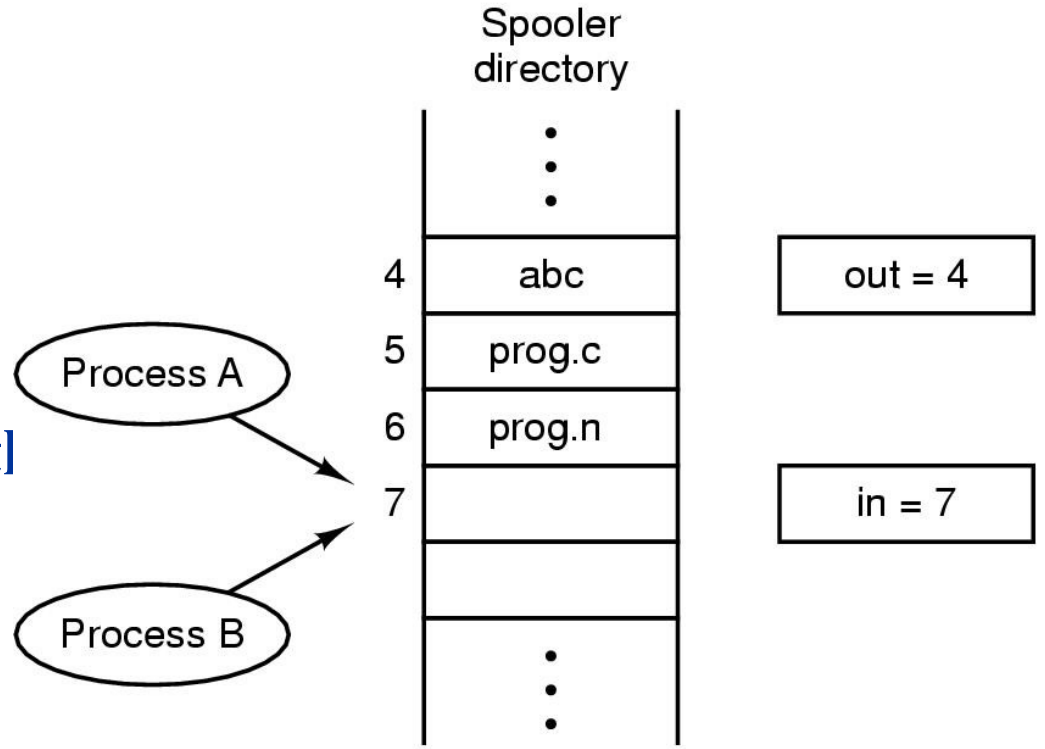
RACE CONDITIONS AND CRITICAL SECTIONS

Race Conditions

- **Race conditions occur when two or more processes attempt to access shared storage.**
  - This causes the final outcome to depend on who runs first.
  - "Shared storage" can mean:
    - ✓**Global variables.**
    - ✓**Memory locations.**
    - ✓**Hardware registers.**
      - *This refers to configuration registers rather than CPU registers.*
    - ✓**Files.**
  - To understand race conditions, we will consider the example of a queue in a print spooler.

Race Conditions

- **A process that wants to print enters the name of the file into a print directory.**
- **A print daemon (printd) periodically checks the directory, prints out the next file and removes its entry.**
- **Two variables keep track of tl queue:**
  - IN: Next available slot.
  - OUT: Next file for printing.



Race Conditions

- **The following can happen:**
  - Process A reads IN as 7 and stores it in a local variable *next*.
  - The OS pre-empts A and starts B.
  - B reads IN as 7 and stores it in a local variable *next*.
  - B inserts its file into slot 7 and updates IN to 8.
  - B is pre-empted and A restarts.
  - A still thinks slot 7 is available and inserts its file there, overwriting B's file, then updates IN to 8.
- **The daemon is unaware of the mistake, and B never gets a printout.** ☹

Critical Sections

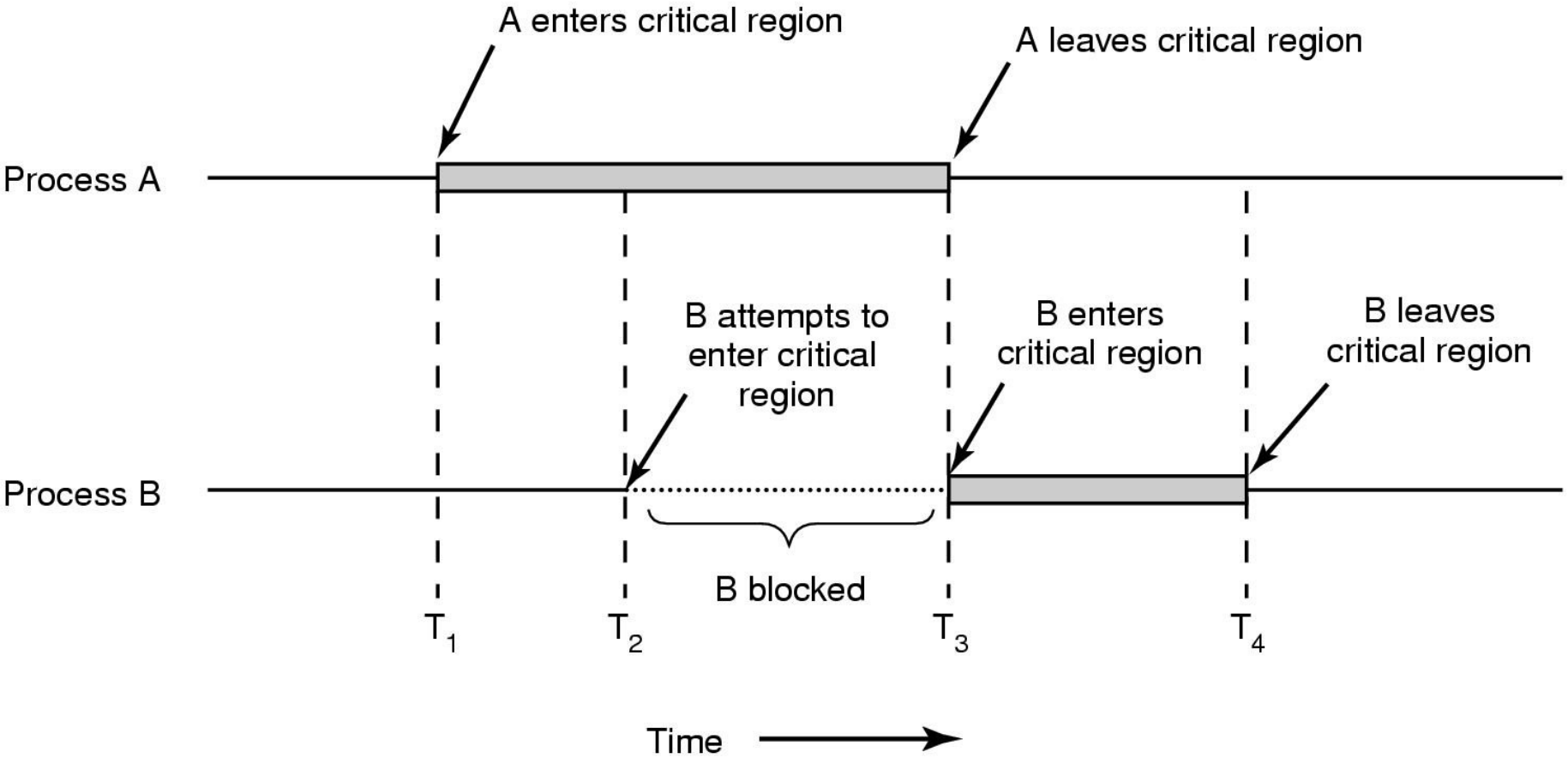
- **To prevent race conditions, we must prevent two processes from reading/writing shared resources at the same time.**
- **This is known as a "mutual exclusion", shortened to "mutex".**
- **Conceptually, a RUNNING process is always in one of two possible states:**
  - It is performing local computation. This does not involve global storage, hence to race condition is possible.
  - It is reading/updating global variables. This can lead to race conditions.
- **When a RUNNING process is in the second state, it is within its "critical section".**

Critical Sections

- **To prevent race conditions, 4 rules must be followed:**
  - No two processes can simultaneously be in their critical section.
  - No assumptions may be made about speeds or # of CPUs.
    - ✓**Note: We can relax this assumption for *most* embedded systems since they have single CPUs.**
    - ✓**May apply to systems using multicore microcontrollers.**
  - No process outside of its critical section can block other processes.
  - No process should wait forever to enter its critical section.

Critical Sections

- **In an ideal state, this is how mutual exclusion works:**



Inter-Process Communications

IMPLEMENTING MUTUAL EXCLUSION

Implementing Mutual Exclusion

- **There are several ways of implementing mutexes, each with their own + and – points:**
  - Disabling interrupts.
  - Lock variables.
  - Strict alternation.
  - Perterson's Solution.
  - Test and set lock.
  - Sleep/Wakeup

Implementing Mutual Exclusion Disabling Interrupts

- **Disabling Interrupts.**
  - This works because:
    - ✓**Time-slicing depends on a timer interrupt. If this is disable, the scheduler is never activated to switch to another process.**
    - ✓**Similarly, processes that are blocked pending an event (e.g. arrival of data from the network), depend on an interrupt to tell the scheduler that the event has taken place.**
  - Therefore disabling interrupts will prevent other processes from starting up and entering their critical sections.

Implementing Mutual Exclusion Disabling Interrupts

- **Disabling Interrupts.**
  - There are several problems with this approach:
    - ✓**Carelessly disabling interrupts can cause the entire system to grind to a halt.**
    - ✓**This only works on single-processor, single core systems. Violates Rule 2.**
  - This approach is used in ArdOS since the supported boards are all single-processor and single-core.

Implementing Mutual Exclusion Disabling Interrupts

- **ArdOS Implementation:**

```
// Atomicity Control
inline void OSMakeAtomic(unsigned char *_csreg)
{
    *_csreg = SREG;
    cli();
}

inline void OSExitAtomic(unsigned char _csreg)
{
    SREG=_csreg;
}
```

Call to OSMakeAtomic and OSExitAtomic

```
unsigned char sreg;
OSMakeAtomic(&sreg);
// Set sleep time
_sleepTime[_running]=millis-1;

if(_sleepTime[_running]<0)
    _sleepTime[_running]=0;

_sleepFlag |= (1<<_running);

// Set blocked flag
_tasks[_running].status|=_OS_BLOCKED;

// Note: No need to remove from READY queue because a _running process would have already been de-queued from there.
// So just call scheduler to swap.

OSExitAtomic(sreg);
```

Implementing Mutual Exclusion Lock Variables

- **Using Lock Variables.**
  - A single global variable “lock” is initially 1.
  - Process A reads this variable and sets it to 0, and enters its critical section.
  - Process B reads “lock” and sees it’s a 0. It doesn’t enter its critical section and waits until “lock” is 1.
  - Process A finishes and sets “lock” to 1, allowing B to enter.

Implementing Mutual Exclusion Lock Variables

- **This approach obviously doesn't work!!**
  - Process A reads in "lock" and sees a "1". It gets pre-empted and Process B runs.
  - Process B reads in "lock", sees a "1", sets it to 0 and enters its critical section.
  - Before B leaves, A is re-started, and enters the critical section.
- **Now >1 process is in the critical section!**
- **PROBLEM: There's a race condition on "lock" itself!**

Test and Set Lock

- **Many microprocessors have an instruction that looks like this:**
  - TSL reg, lock ; lock is a variable in memory
- **How this works:**
  - CPU locks the address and data buses, and reads "lock" from memory.
    - ✓**The locked address and data buses will block accesses from all other CPUs.**
  - The current value is written into register "reg".
  - A "1" (or sometimes "0") value is written to "lock".
  - CPU unlocks the address and data buses.
- **The TSL is "atomic".**
  - This means that NOTHING can interrupt execution of this instruction.
  - This is guaranteed in hardware.

Test and Set Lock

- **The TSL instruction is used as follows:**

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered


leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET | return to caller
```

Test and Set Lock

- **An alternative is the XCHG instruction, used on Intel machines.**
  - Swaps contents of "lock" and "reg" instead of just writing "1" to lock.

```
enter region:
        MOVE REGISTER, #1       ; Set REGISTER to 1
        XCHG REGISTER, LOCK     ; Exchange with Lock
        CMP REGISTER, #0        ; Was Lock 0?
        JNE enter_region        ; No, go back and try again.
        RET                     ; Yes, enter critical region.

leave_region:
        MOVE LOCK, #0           ; Clear the lock
        RET                     ; And exit
```

Deadlock

- **Busy-wait approaches like TSL/XCHG have a problem called "deadlock".**
- **Consider two processes H and L, and a scheduler rule that says that H is always run when it is READY. Suppose L is currently in the critical region.**
  - H becomes ready, and L is pre-empted.
  - H tries to obtain a lock, but cannot because L is in the critical region.
  - H loops forever, and CPU control never gets handed to L.
  - As a result L never releases the lock.
- **Note: The book calls this a "priority inversion", which is incorrect.**

Sleep/Wake

- **One solution to this problem is through the use of "Sleep/Wake" functions.**
  - When a process finds that a lock has been set (i.e. another process in the critical section), it calls "sleep" and is put into the blocked state.
  - When the other process exits the critical section and clears the lock, it can call "wake" which moves the blocked process into the READY queue for eventual execution.
- **While this sounds like an ideal solution, it can create a problem called the "producer-consumer problem".**

Inter-Process Communications

THE PRODUCER/CONSUMER PROBLEM

The Producer/Consumer problem

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

The Producer/Consumer problem

- **Producer and consumer share a fixed-size buffer.**
  - A global variable "count" keeps track of the number of items.
    - ✓**If count==N (FULL), producer sleeps, if count==0 (EMPTY) consumer sleeps.**
  - After reading from the buffer, if count==N-1:
    - ✓**Consumer reasons that the buffer was earlier full and wakes the producer.**
  - After writing to the buffer, if count==1
    - ✓**Producer reasons that the buffer was earlier empty and wakes the consumer.**

The Producer/Consumer problem

- **Deadlock occurs when:**
  - Consumer checks "count" and finds it is 0.
  - Consumer gets pre-empted and producer starts up.
  - Producer adds an item, increments count to "1", then sends a WAKE to the consumer.
    - ✓**Since consumer is not technically sleeping yet, the WAKE is lost.**
  - Consumer starts up, and since count is 0, goes to SLEEP.
  - Producer starts up, fills buffer until it is full and SLEEPs.
- **Since consumer is also SLEEPing, no one wakes the producer. Deadlock.**

Inter-Process Communications

SEMAPHORES

Semaphores

- **A semaphore is a special lock variable that counts the number of wake-ups saved for future use.**
  - A value of "0" indicates that no wake-ups have been saved.
- **Two ATOMIC operations on semaphores:**
  - DOWN, TAKE, PEND or P:
    - ✓**If the semaphore has a value of >0, it is decremented and the DOWN operation returns.**
    - ✓**If the semaphore is 0, the DOWN operation blocks.**
  - UP, POST, GIVE or V:
    - ✓**If there are any processes blocking on a DOWN, one is selected and woken up.**
    - ✓**Otherwise UP increments the semaphore and returns.**

Using Semaphores in the Producer/Consumer Problem

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                          /* decrement empty count */
          down(&mutex);                          /* enter critical region */
          insert_item(item);                     /* put new item in buffer */
          up(&mutex);                            /* leave critical region */
          up(&full);                             /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                           /* decrement full count */
          down(&mutex);                          /* enter critical region */
          item = remove_item( );                 /* take item from buffer */
          up(&mutex);                            /* leave critical region */
          up(&empty);                            /* increment count of empty slots */
          consume_item(item);                    /* do something with the item */
     }
}
```

- EMPTY - # of empty slots.
- FULL - # of full slots.
- MUTEX – Prevents simultaneous access to the buffer.

Mutual Exclusion with Semaphores

- **When a semaphore's counting ability is not needed, we can use a simplified version called a "mutex".**
  - 1 = Unlocked.
  - 0 = Locked.
- **Two processes can then attempt do DOWN the semaphore.**
  - Only one will succeed. The other will block.
  - When the successful process exits the critical section, it does an UP, waking the other process up.

Mutual Exclusion with Semaphores

Process A

```
sema=1
…
non_critical_section()
DOWN(sema)
critical_section()
UP(sema)
…
```

Process B

```
non_critical_section()
DOWN(sema)
critical_section()
UP(sema)
…
```

Mutual Exclusion with TSL/XCHG

- **We can also implement mutexes with TSL or XCHG.**
  - 0 = Unlocked, 1 = Locked.

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again later
ok:     RET | return to caller; critical region entered


mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET | return to caller
```

Semaphore Implementation in ArdOS

- **Taking a Semaphore:**

```
void OSTakeSema(TOSSema *sema)
{
    unsigned char sreg;

    OSMakeAtomic(&sreg);
    if(sema->semaval>0)
        sema->semaval--;
    else
    {
        // Block current process
        _tasks[_running].status |= _OS_BLOCKED;

        // Enqueue this task
        prioEnq(_running, _tasks, &sema->taskQ);
        OSExitAtomic(sreg);

        // Call scheduler.
        OSSwap();
    }
    OSExitAtomic(sreg);
}
```

Semaphore Implementation in ArdOS

- **Giving a Semaphore**

```
void OSGiveSema(TOSSema *sema)
{
    unsigned char sreg;
    OSMakeAtomic(&sreg);

    unsigned char tsk=procDeq(&sema->taskQ);

    if(tsk != 255)
    {
        // Removed blocked flag
        _tasks[tsk].status &= ~(_OS_BLOCKED);
        procEnq(tsk, _tasks, &_ready);

        // Call scheduler
        OSExitAtomic(sreg);
        OSPrioSwap();
    }
    else
        if(sema->isBinary)
            sema->semaval=1;
        else
            sema->semaval++;
        OSExitAtomic(sreg);
}
```

Inter-Process Communications

DEADLOCKS WITH SEMAPHORES

Problems with Semaphores Deadlock

- **Our producer/consumer solution has a problem:**
  - If we swapped the semaphores for empty/full with the mutex semaphore, we have a potential deadlock:

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&mutex);                          /* decrement empty count */
          down(&empty);                          /* enter critical region */
          insert_item(item);                     /* put new item in buffer */
          up(&mutex);                            /* leave critical region */
          up(&full);                             /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&mutex);                          /* decrement full count */
          down(&full);                           /* enter critical region */
          item = remove_item( );                 /* take item from buffer */
          up(&mutex);                            /* leave critical region */
          up(&empty);                            /* increment count of empty slots */
          consume_item(item);                    /* do something with the item */
     }
}
```

Problems with Semaphores Deadlock

- **This can happen:**
  - Producer successfully DOWNs the mutex.
  - Producer DOWNs "empty". However the queue is full so this blocks.
  - Consumer DOWNs mutex and blocks.
    - ✓**Consumer now never reaches the UP for "empty" and therefore cannot unblock the producer.**
    - ✓**The producer in turn never reaches the UP for mutex and cannot unblock the consumer.**
    - ✓**Deadlock!**

Deadlock
Reusable/Consumable Resources

- **Reusable Resources**
  - Examples: memory, devices, files, tables
  - Number of units is **constant**
  - Unit is either free or allocated; **no sharing**
  - Process **requests, acquires, releases units**
- **Consumable Resources**
  - Examples: messages, signals
  - Number of units **varies** at runtime
  - Process **releases** (create) units (w/o acquire)
  - Other process **requests** and **acquires** (consumes)

Deadlock
More Deadlock Examples

```
p1: ...                    p2: ...
    open(f1,w);                open(f2,w);
    open(f2,w);                open(f1,w);
    ...                        ...
```

- **Deadlock when executed concurrently**

```
p1: if (C) send(p2,m);     p2: ...
      while(1) {...              while(1) {...
        recv(p2,m);                recv(p1,m);
        send(p2,m);                send(p1,m);
        ... }                      ... }
```

- **Deadlock when C not true**

Dealing with Deadlocks

1. **Detection and Recovery**
   - Allow deadlock to happen and eliminate it
2. **Avoidance (dynamic)**
   - Runtime checks disallow allocations that might lead to deadlocks
3. **Prevention (static)**
   - Restrict type of request and acquisition to make deadlock impossible

Deadlock Prevention

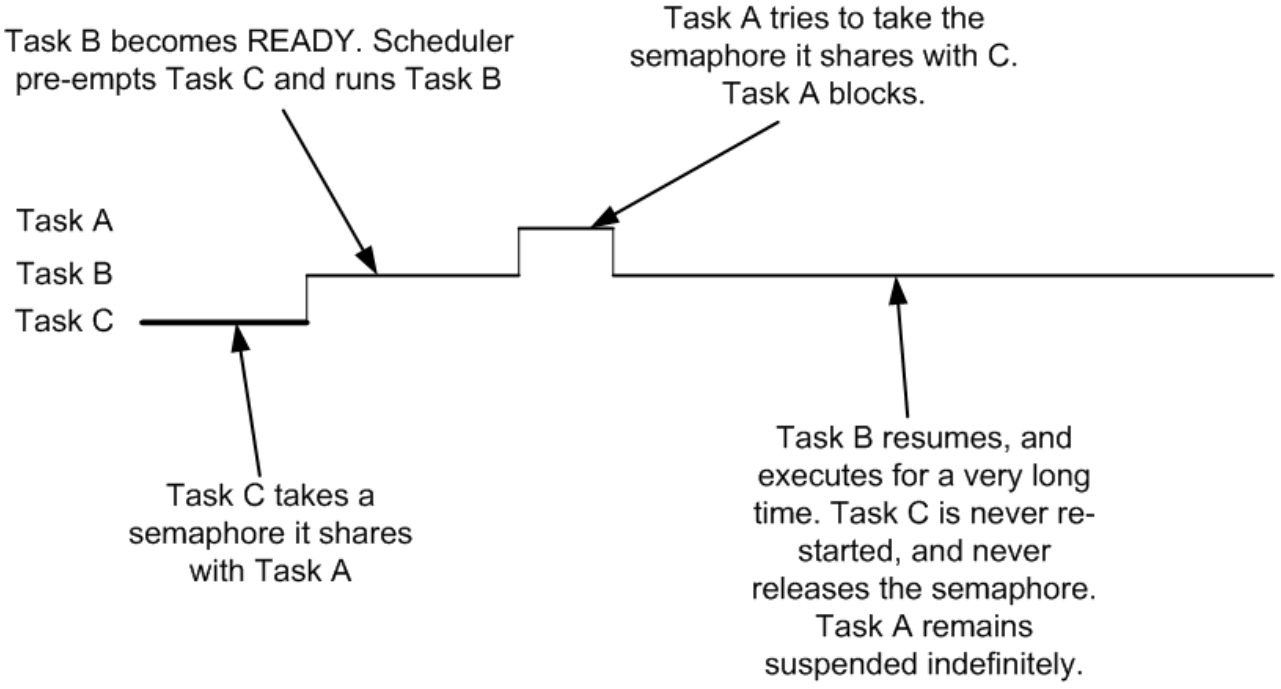
- **Deadlock requires the following 3 conditions:**
  1. **Mutual exclusion**:
     - ✓**Resources not sharable**
  2. **Hold and wait:**
     - ✓**Process must be holding one resource while requesting another**
  3. **Circular wait:**
     - ✓**At least 2 processes must be blocked on each other**

Deadlock Prevention

1. **Eliminate mutual exclusion**
   - Not possible in most cases
   - Spooling makes I/O devices sharable
2. **Eliminate hold-and-wait**
   - **Request** all resources **at once**
   - **Release** all resources **before a new request**
   - **Release** all resources **if current request blocks**
3. **Eliminate circular wait**
   - Order all resources
   - Process must request in **ascending order**

Problems with Semaphores
Priority Inversion

- **In the diagram on the following page, priority(Process C) < priority(Process B) < priority(Process A).**



- **Process B effectively blocks out Process A, although Process A has higher priority!**

Inter-Process Communications

MONITORS AND CONDITIONAL VARIABLES

Monitors

- **A monitor is similar to a class or abstract-data type in C++ or JAVA:**
  - Collection of procedures, variables and data structures grouped together in a package.
    - ✓**Access to variables and data possible only through methods defined in the monitor.**
  - However, only one process can be active in a monitor at any point in time.
    - ✓**I.e. if any other process tries to call a method within the monitor, it will block until the other process has exited the monitor.**

Monitors

- **Implementation:**
  - When a process calls a monitor method, the method first checks to see if any other process is already using it.
  - If so, the calling process blocks until the other process has exited the monitor.
    - ✓**This can be achieved using mutexes or binary semaphores.**
    - ✓**The mutex/semaphore operations are inserted by the compiler itself rather than by the user, reducing the likelihood of errors.**

Monitors and Condition Variables

- **Monitors achieve mutual exclusion, but we also need other mechanisms for coordination.**
  - E.g. in our producer/consumer problem, mutual exclusion alone is not enough to prevent the producer from proceeding when the buffer is full.
- **We introduce "condition variables".**
  - One process WAITs on a condition variable and blocks, until..
  - Another process SIGNALs on the same condition variable, unblocking the WAITing process.

Monitors and Condition Variables

- **Implementing the Producer/Consumer problem with semaphores and condition variables:**
  - When the buffer is full (count==N), producer will WAIT on a full condition.
  - When buffer is empty (count==0), consumer will WAIT on empty.

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;
        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Monitors and Condition Variables

- **When a process encounters a WAIT, it is blocked and another process is allowed to enter the monitor.**
- **Problem:**
  - When there's a SIGNAL, the sleeping process is woken up.
  - We will potentially now have two processes in the monitor at the same time:
    - ✓**The process doing the SIGNAL (the signaler).**
    - ✓**The process that just woke up because of the SIGNAL (the signaled).**

Monitors and Condition Variables

- **We have 3 ways to resolve this:**
  - We require that the signaler exits immediately after calling SIGNAL.
  - We suspend the signaler immediately and resume the signaled process.
  - We suspend the signaled process until the signaler exits, and resume the signaled process only after that.

Monitors and Condition Variables

- **A condition variable is different from a semaphore.**
  - Semaphore:
    - **✓If Process A UPs a semaphore with no pending DOWN, the UP is saved.**
    - **✓The next DOWN operation will not block because it will match immediately with a preceding UP.**
  - Condition variable:
    - **✓If Process A SIGNALs a condition variable with no pending WAIT, the SIGNAL is simply lost.**
    - **✓This is similar to the SLEEP/WAKE problem earlier on.**

Monitors and Condition Variables

- **This code looks suspiciously like our original producer/consumer problem on Page 23!**
  - Same issues too:
    - ✓**Page 23:**

      Consumer sees count==0, and intends to SLEEP but gets pre-empted.

      Producer sends a WAKE but the WAKE is lost.

      In this case, if the consumer gets pre-empted before a WAIT, the corresponding SIGNAL from the producer is also lost!

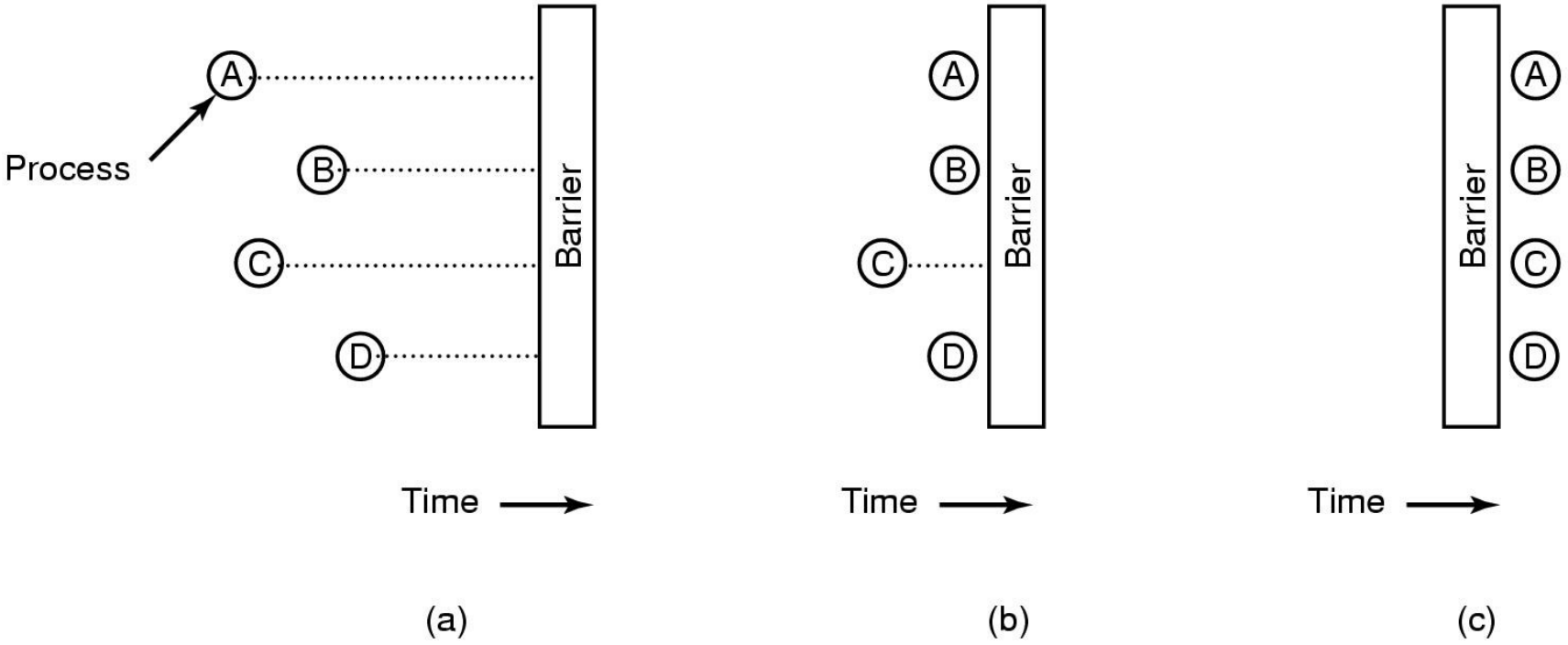
    - ✓**However we see that in this case, the mutual exclusion from the monitor prevents the SIGNAL from being lost! (WHY?)**

Inter-Process Communications

BARRIERS

Barriers

- **A "barrier" is a special form of synchronization mechanism that works with groups of processes rather than single processes.**



Barriers

- **The idea of a barrier is that all processes must reach the barrier (signifying the end of one phase of computation) before any of them are allowed to proceed.**
  - Process D reaches the end of the current phase and calls a BARRIER primitive in the OS. It gets blocked.
  - Similarly processes A and B reach the end of the current phase, calls the same BARRIER primitive and is blocked.
  - Finally process C reaches the end of its computation, calls the BARRIER primitive, causing all processes to be unblocked at the same time.

Barriers

- **Example barrier application.**
  - Computing fluid motion across a surface represented by a 1,000,000 x 1,000,000 matrix.
    - ✓**The complete matrix for iteration *N* must be available before computing for iteration *N+1*.**
    - ✓**1,000 individual processes each computing a part of the entire matrix.**
    - ✓**We must wait for all 1,000 processes to complete finding the entire matrix before we can start on the next iteration.**
  - A barrier would be very useful in achieving this.

IT5002

Computer Systems and Applications

Lecture 15

Memory Management

colinta@comp.nus.edu.sg

National University of Singapore

School of Computing

Learning Objectives

- **By the end of this lecture you will be able to:**
  - Understand the concept of address spaces and memory management.

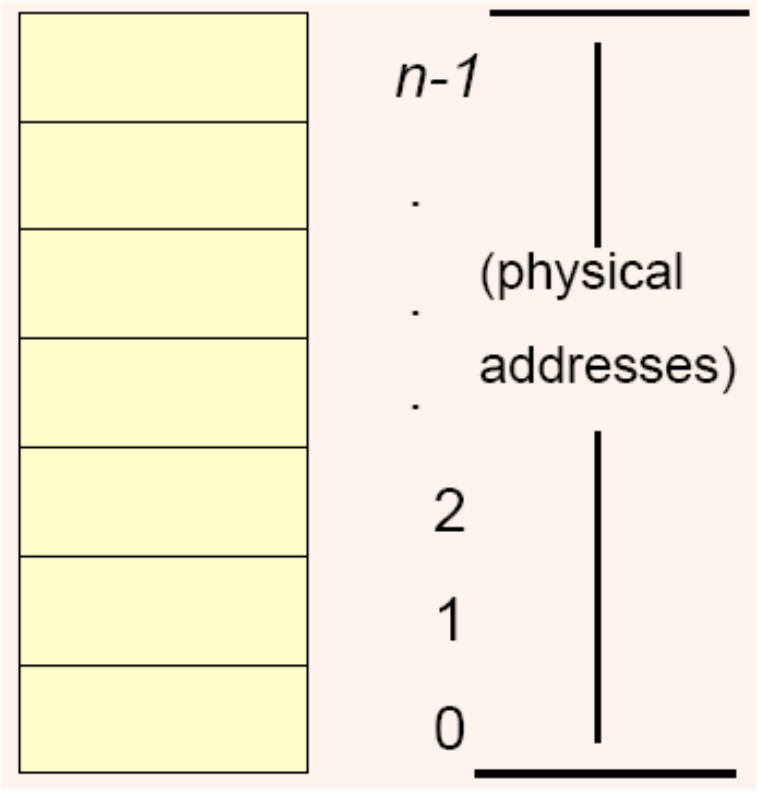
Introduction

- **Memory is crucial for computers:**
  - Used to store:
    - ✓**Kernel code and data**
    - ✓**User code and data**
- **What responsibilities does the OS have here?**
  - Allocate memory to new processes.
  - Manage process memory.
  - Manage kernel memory for its own use.
  - Provide OS services to:
    - ✓**Get more memory, e.g. via malloc.**
    - ✓**Free memory, e.g. via free**
    - ✓**Protect memory.**

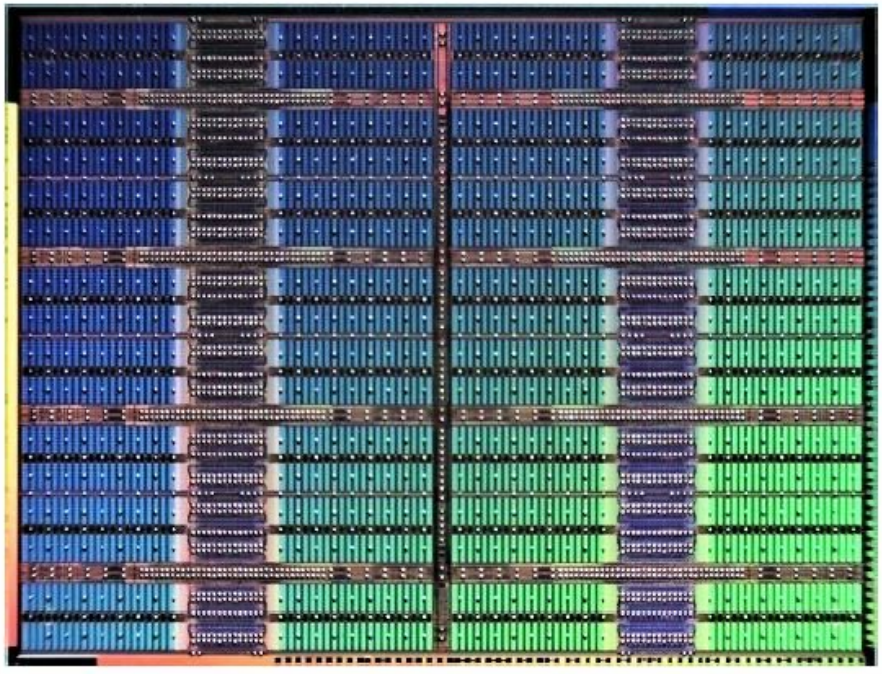
PHYSICAL MEMORY ORGANIZATION

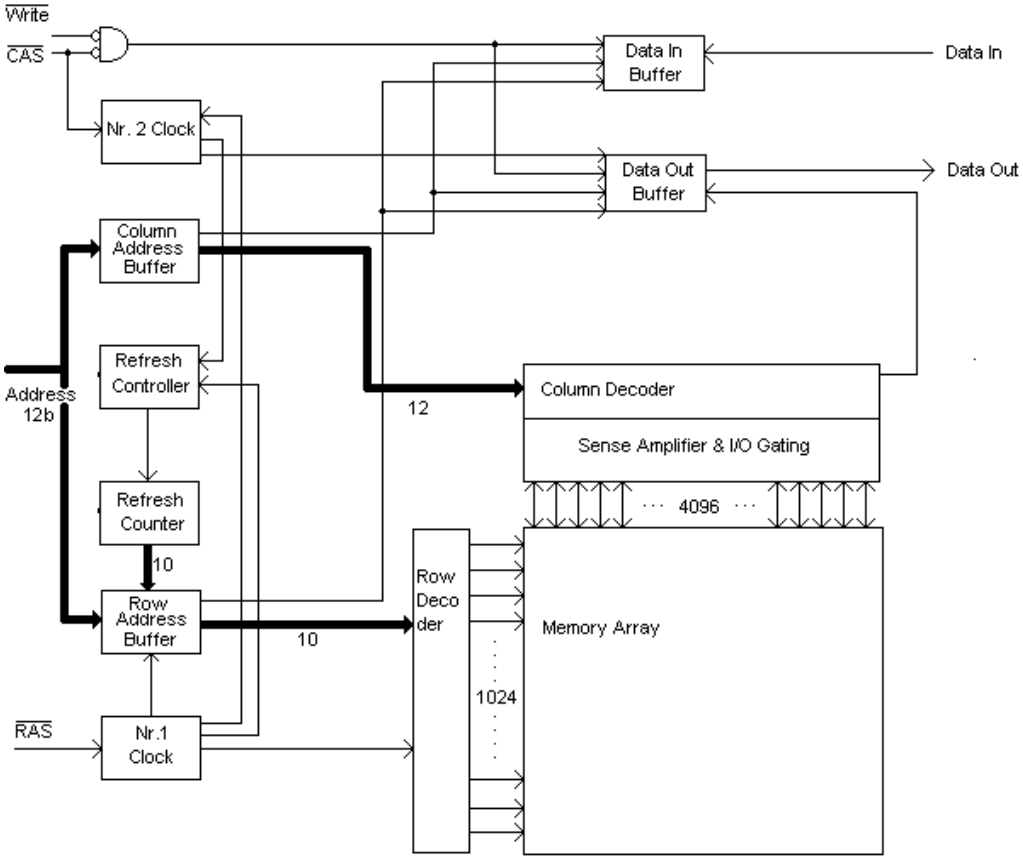
Physical Memory Organization

- **Physical memory is:**
  - The actual matrix of capacitors (DRAM) or flip-flops (SRAM) that stores data and instructions.
  - Arranged as an array of bytes.
- **Memory addresses serve as byte indices.**



Physical Memory Organization





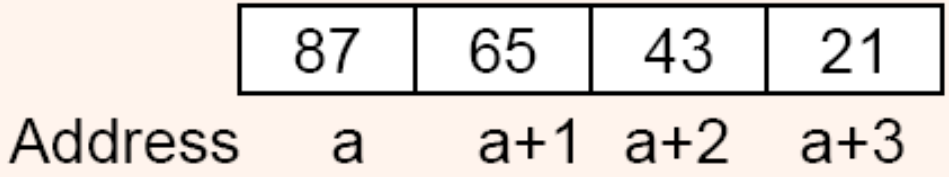
Words

- **Physical memory is organized in bytes, but CPUs often transfer data in units of >1 byte.**
  - This unit is known as a "word".
  - 1 byte in 8-bit machines (ATMega328P, Intel 8080), 2 bytes in 16 bit machines (Intel 80286), 4 bytes in 32-bit machines (Intel Xeon), 8 bytes in 64-bit machines (Intel Celeron)
- **How do we map multi-byte words to single-byte memory?**

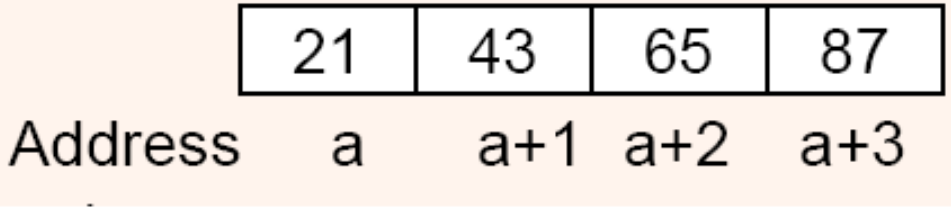
Endianness

How to store multibyte word (object > 1 byte), 2 general schemes, Eg: 0x87654321 (4 byte int)

- **Big Endian**: higher order bytes at lower addresses



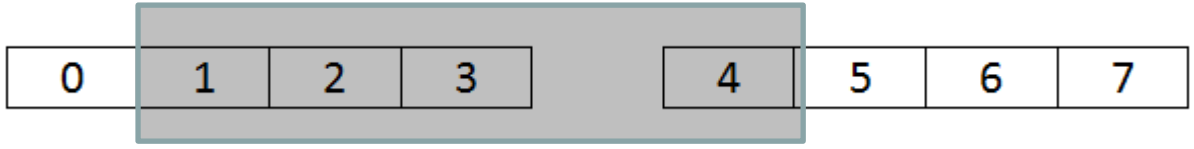
- **Little Endian**: lower order bytes at lower addresses



x86: little endian ; Sparc: big endian ; powerpc: configurable

Alignment Issues

- **Data can be fetched across word boundaries.**



  - E.g. fetching from address 1 in a 32-bit machine:
    - ✓**Bytes from addresses 0 to 3 are fetched.**
    - ✓**Bytes from addresses 4 to 7 are fetched.**
    - ✓**Bytes from addresses 0, 5 to 7 are discarded.**
    - ✓**Bytes from addresses 1 to 3, 4 are merged.**
  - SLOW!!

Alignment Issues

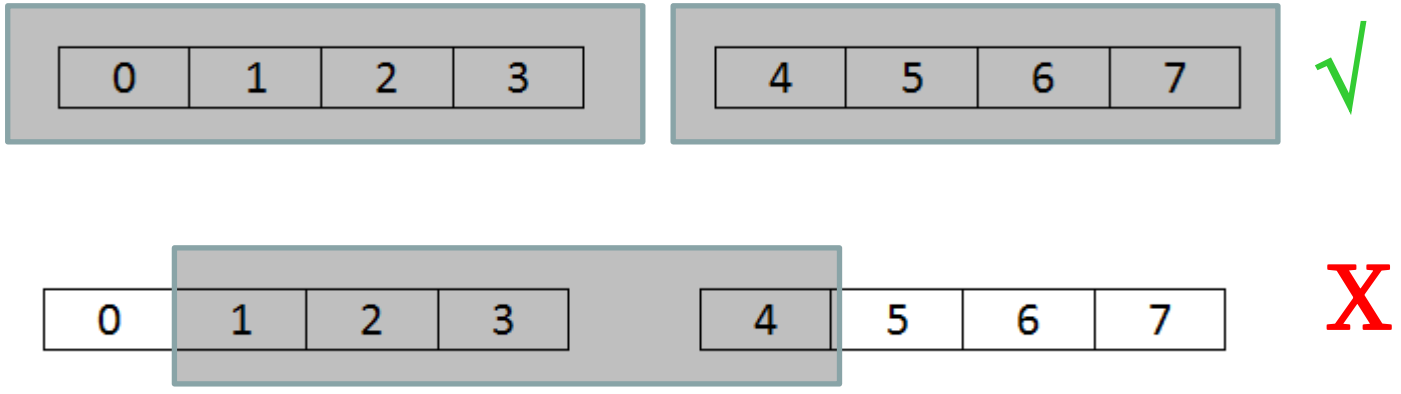
- **Since mis-aligned reads are very inefficient, data structures should always be created in units of words.**
  - Add unused bytes to ensure this.

- eg.
```
struct {
      char a;
      int b;  // 32 bits, must be aligned on 32-bit boundary
    } x;
```
- treat as if padding added
```
struct {
      char a;
      char pad[3]; // force address of b to be aligned
      int b;  // b now aligned
    } x1;
```

sizeof(x1) = 8; if x1 is aligned on its size (8 bytes) then b is also aligned on 32-bit boundary

Alignment Issues

- **Due to CPU fetch circuitry design, instructions usually must be fetched on word boundaries.**



- **Instructions fetched across word boundaries trigger "Bus Error" faults.**

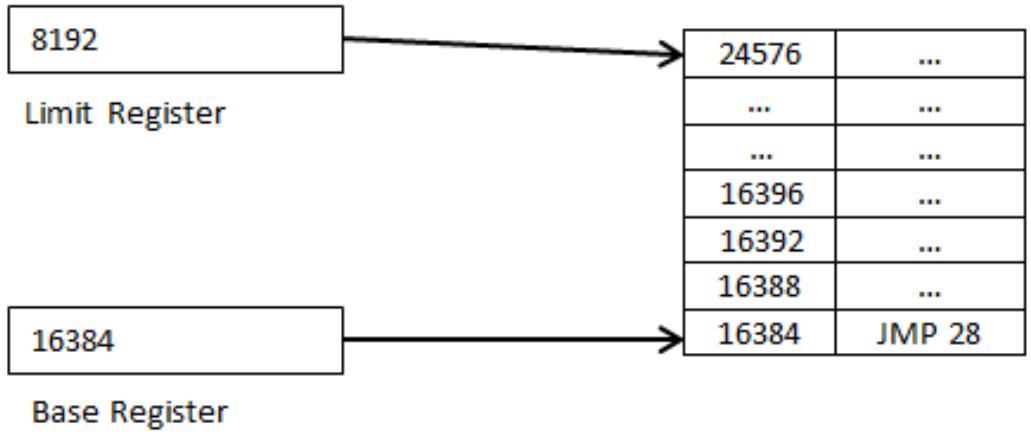
MEMORY MANAGEMENT

Memory Management

- **Why Memory Management?**
  - We want to use memory efficiently:
    - ✓**What memory has already been allocated and to whom?**
    - ✓**What memory is now free and usable by others?**
  - We want to protect processes from each other.
    - ✓**One process should not be able to trash another process or the OS.**
    - ✓**E.g. we don't want Process 1 to be messing about with the variables and memory used by Process 2.**

Logical vs. Physical Addresses

- **Logical addresses:**
  - These are the addresses as "seen" by executing processes code.
- **Physical addresses:**
  - These are addresses that are actually sent to memory to retrieve data or isntructions.



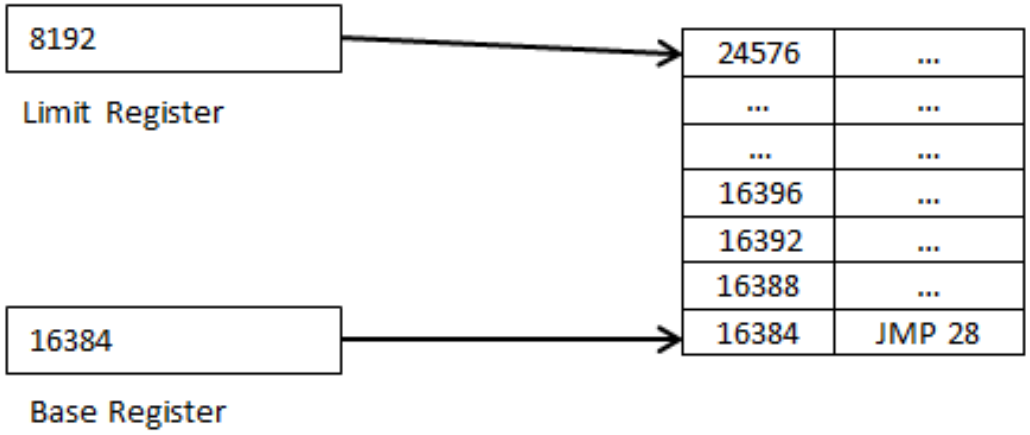
Multiple Program Systems

- **Having multiple processes complicates memory management:**
  - Conflicting addresses.
    - ✓**What if >1 program expects to load at the same place in memory?**
  - Access violations.
    - ✓**What if 1 program overwrites the code/data of another?**
    - ✓**Worse, what if 1 program overwrites parts of the operating system?**
  - The ideal situation would be to give each program a section of memory to work with.
    - ✓**Basically each program will have its own address space!**

Multiple Program Systems: Base and Limit Registers

- **To do this we require extra hardware support:**
  - Base register:
    - ✓**This contains the starting address for the program.**
    - ✓**All program addresses are computed relative to this register.**
  - Limit register:
    - ✓**This contains the length of the memory segment.**
- **These registers solve both problems:**
  - We can resolve address conflicts by setting different values in the base register.
  - If a program tries to access memory below the base register value or above the (base+limit) register value, a "segmentation fault" occurs!

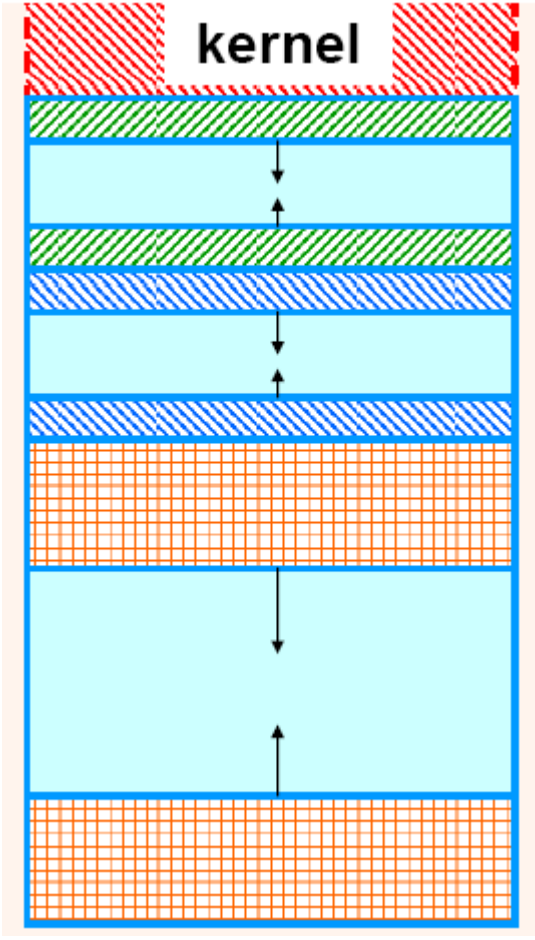
Multiple Program Systems: Base and Limit Registers



- **All memory references in the program are relative to the Base Register.**
  - E.g. "jmp 28" above will cause a jump to location 16412.
- **Any memory access to location 24576 and above (or 16383 and below) will cause segmentation faults.**
  - Other programs will occupy spaces above and below the segment given to the program shown here.

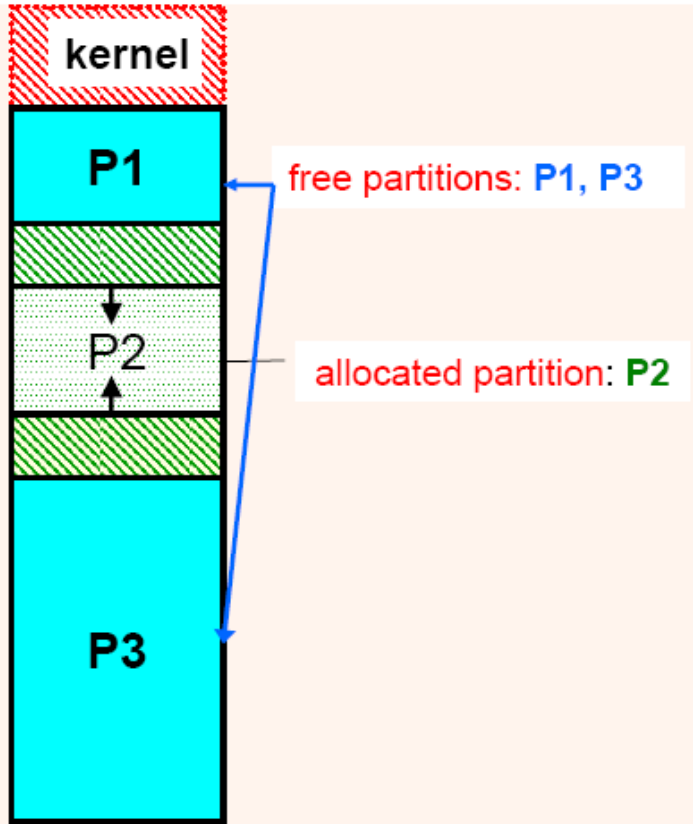
Multiple Program Systems: Partitioning

- **Base and limit registers allow us to partition memory for each running process.**
  - Each process has its own fixed partition.
  - Assumed that we know how much memory each process needs.



Partitioning Issues: Fragmentation

- **Two types of fragmentation can arise:**
  - Internal fragmentation:
    - ✓**Partition is much larger than is needed.**
    - ✓**Cannot be used by other processes.**
    - ✓**Extra space is wasted.**
  - External fragmentation:
    - ✓**Free memory is broken into small chunks by allocated memory.**
    - ✓**Sufficient free memory in TOTAL, but individual chunks insufficient to fulfill requests.**



Managing Memory within Processes

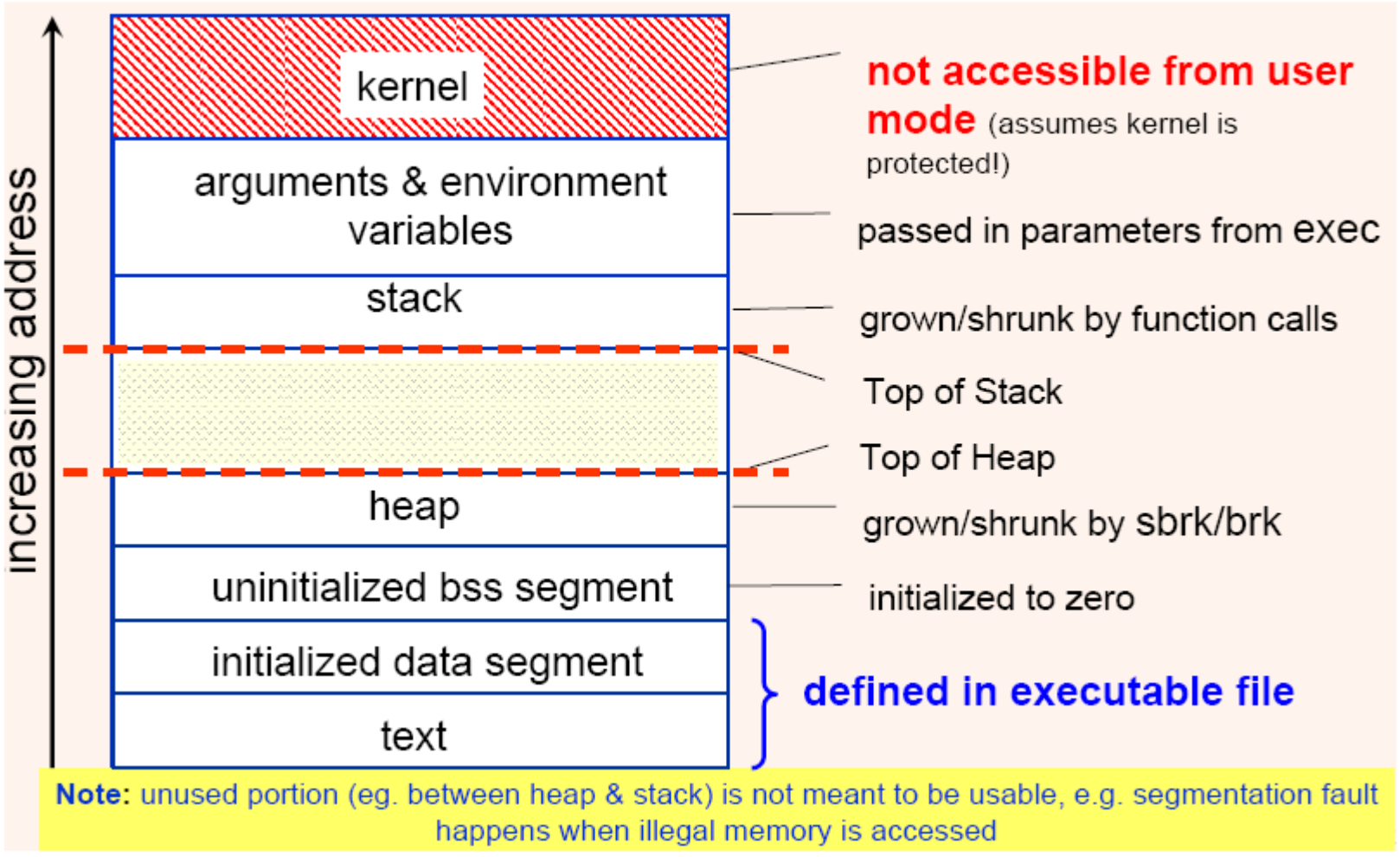
- **We've seen how we can manage multiple processes within an operating system. But:**
  - What about memory within individual programs?
    - ✓**OS allocates memory for instructions.**
    - ✓**Global variables are created as part of the program's environment, and don't need to be specially managed.**
    - ✓**A "stack" is used to create local variables and store local addresses.**
    - ✓**A "heap" is used to create dynamic variables.**

Managing Memory within Processes

- **E.g. in UNIX, process space is divided into:**
  - Text segments: Read-only, contains code. May have >1 text segments.
  - Initialized Data: Global data initialized from executable file. E.g. when you do:

    char *mesg[]="Hello world!";
  - BSS Segment: Contains uninitialized globals.
  - Stack: Contains statically allocated local variables and arguments to functions, as well as return addresses.
  - Heap: Contains dynamically allocated memory.

Managing Memory within Processes



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

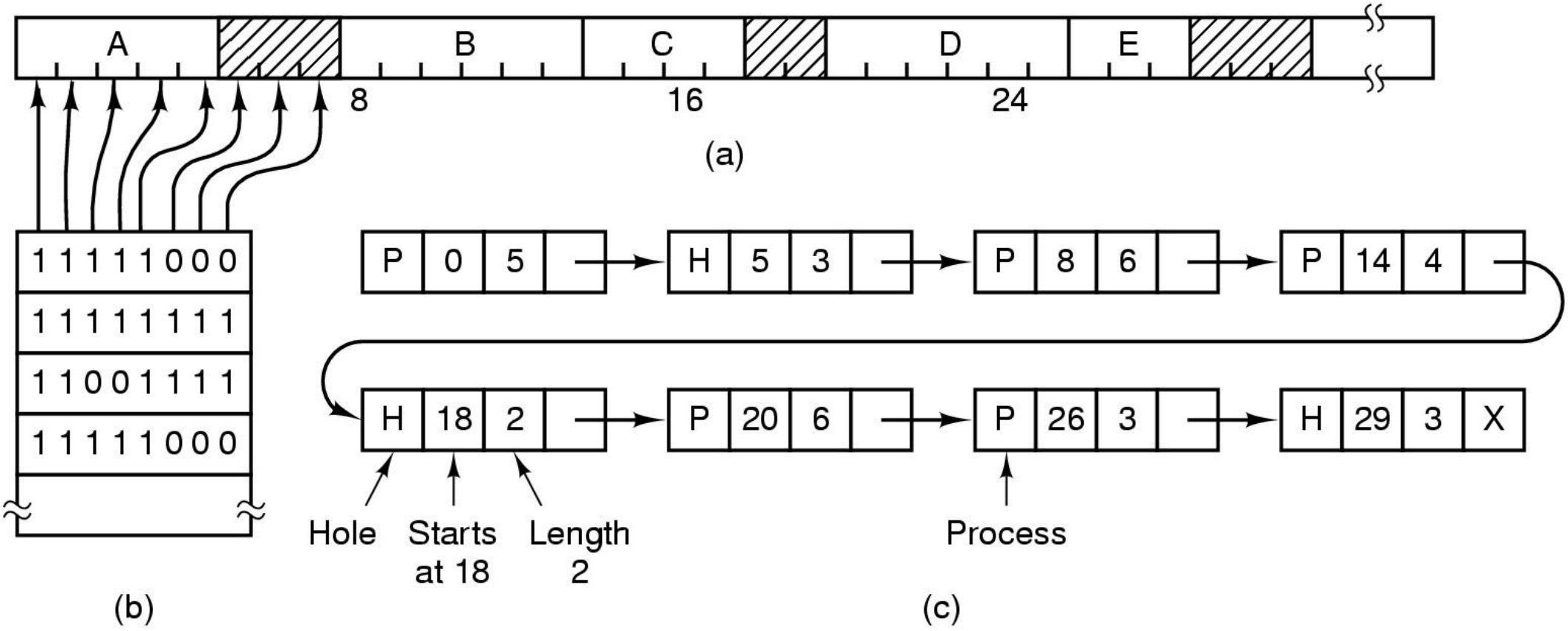
Managing Free Memory

- **When memory can be allocated and de-allocated (e.g. using malloc/free or new/delete), it has to be managed by the operating system.**
  - E.g. when a program requests for 168 bytes of memory, where should it come from?
  - When a program frees 215 bytes of memory, what should happen?
    - ✓**E.g. if it happens to be next to a 64 byte block of free memory, should they be coalesced into a single 279 byte block?**
  - Should small fragments of free memory scattered all over be compacted?

Managing Free Memory

- **To manage free memory, we must know where these free chunks of memory are.**
- **Two approaches:**
  - Bit maps
  - Free/Allocated List
- **In either approach, memory is divided up into fixed sized chunks called "allocation units".**
  - Common sizes range from several bytes (e.g. 16 bytes) to several kilobytes.
  - Each "tick mark" in figure (a) on the next page represents the boundary of an allocation unit.

Managing Free Memory



Managing Free Memory
Bit Maps

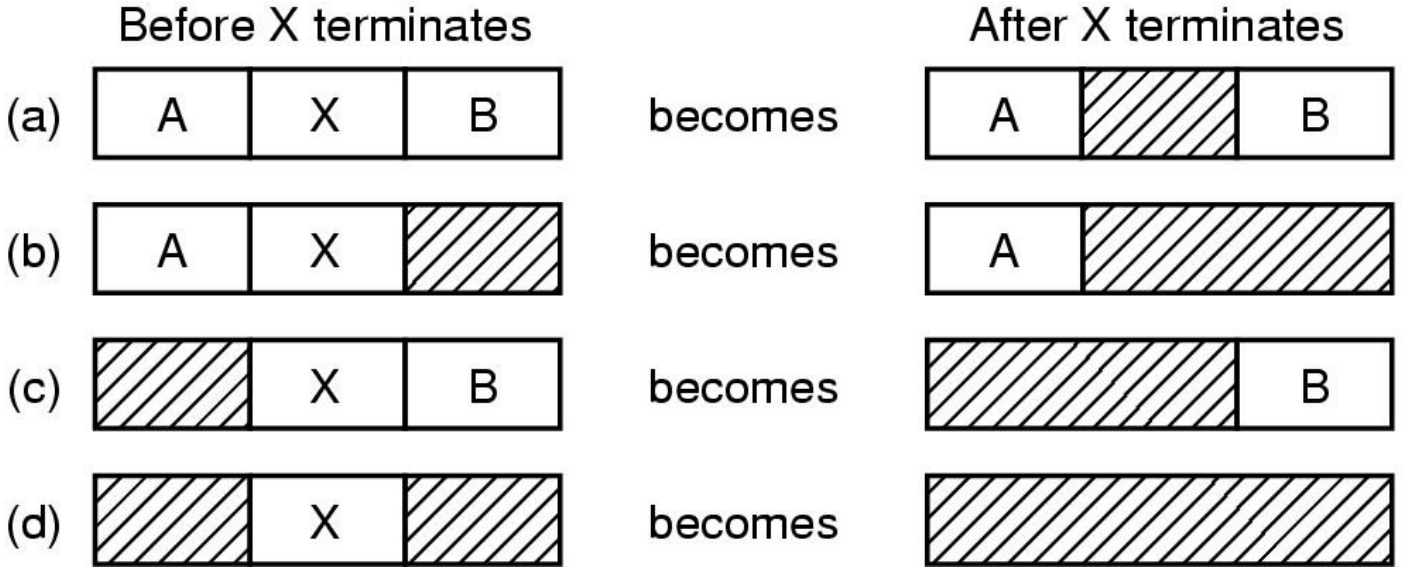
- **Figure (b) on the previous page shows how bit maps are used to keep track of free memory.**
  - Each bit corresponds to an allocation unit.
    - ✓**A "0" indicates a free unit, a "1" indicates an allocated unit.**
  - If a program requests for 128 bytes:
    - ✓**Find how many allocation units are needed. E.g. if each unit is 16 bytes, this corresponds to 8 units.**
    - ✓**Scan through the list to find 8 consecutive 0's.**
    - ✓**Allocate the memory found, and change the 0's to 1's.**
  - If a program frees 64 bytes:
    - ✓**Mark the bits corresponding to the 4 allocation units as "0".**

Managing Free Memory
Free/Allocated List

- **Figure (c) shows an alternative method:**
  - A single linked list is used to track allocated ("P") units and free ("H") units.
    - ✓**Each node on the linked list also maintains where the block of free units start, and how many consecutive free units are present in that block.**
  - Allocating free memory becomes simple:
    - ✓**Scan the list until we reach a "H" node that points to a block of a sufficient number of free units.**

Managing Free Memory
Free/Allocated List

- **The list is implemented as a double-linked list.**
  - Diagram below shows possible "neighbor combinations" that can occur when a process X terminates.
  - The "back pointer" in a double-linked list makes it easy to coalesce freed blocks together.



Managing Free Memory Allocation Policies

- **The bit map and linked list tell us where the free space is, but not what to allocate!**
- **There are several possibilities on how to do this.**
  - Note that in each case, whatever free allocation units left at the end of an allocated block is returned to the free list.
    - ✓**E.g. if a block of 8 units is found and only 6 are needed, the remaining 2 units are marked as "free".**
  - First Fit
    - ✓**Scan through the list/bit map and find the first block of free units that can fit the requested size.**
    - ✓**Fast, easy to implement.**

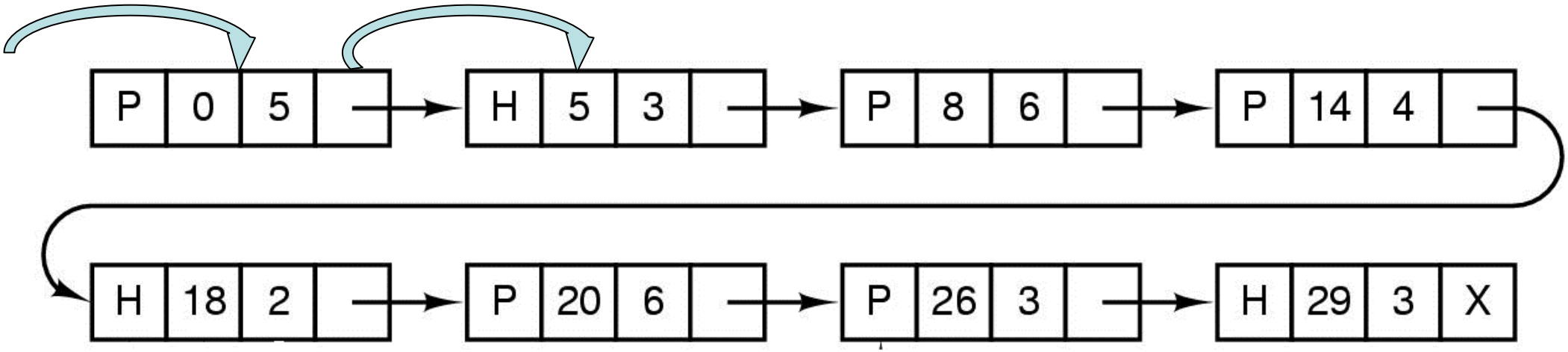
Managing Free Memory Allocation Policies

- Best Fit
  - ✓**Scan through the list/bit map to find the smallest block of free units that can fit the requested size.**
  - ✓**Theoretically should minimize "waste".**
  - ✓**However can lead to scattered bits of tiny useless holes.**
- Worst Fit
  - ✓**Find the largest block of free memory.**
  - ✓**Theoretically should reduce the number of tiny useless holes.**

- **We can sort the free memory from smallest to largest for best fit, or largest to smallest for worst fit.**
  - This minimizes search time.
  - However coalescing free neighbors will be much harder.

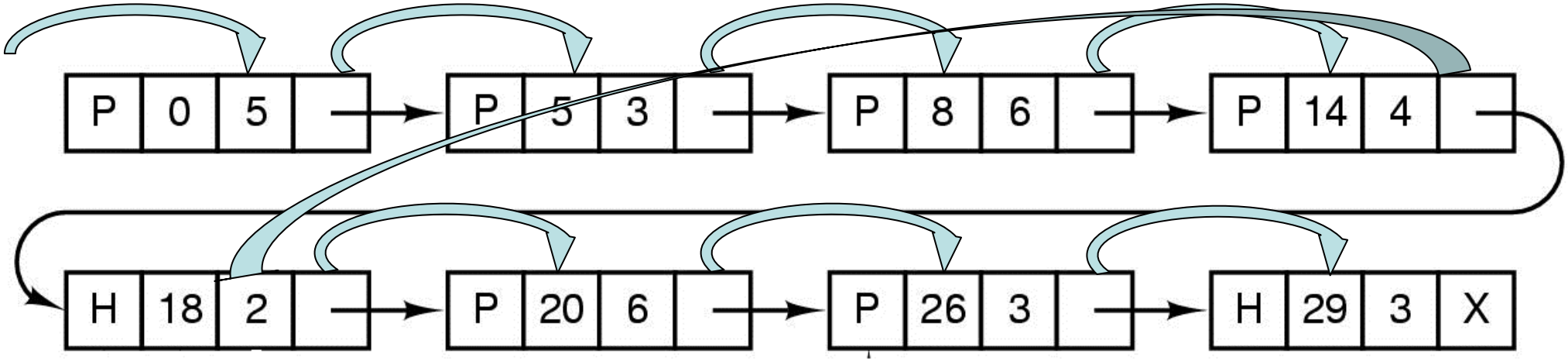
Managing Free Memory Allocation Policies

- **E.g. using first-fit and linked lists:**
  - First allocation: 3 units.
  - Second allocation: Another 3 units.



First Allocation

Managing Free Memory Allocation Policies

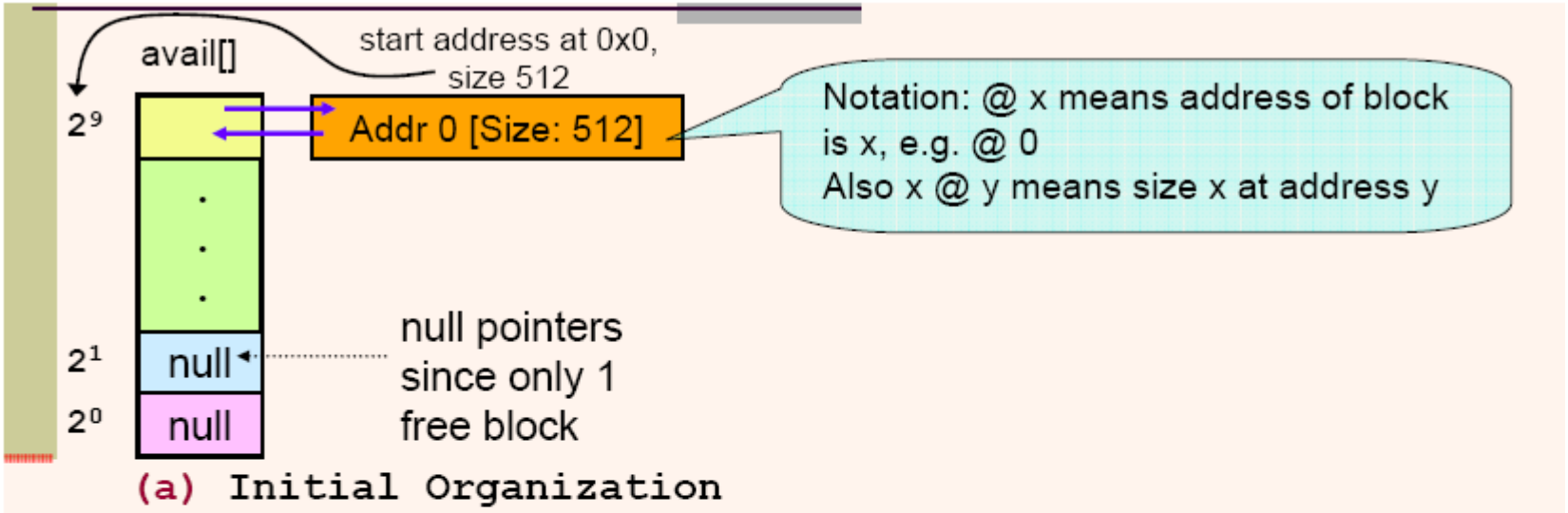


Second Allocation

Buddy Allocation (aka Quick Fit)

- **This is an efficient (better than O(n)) way to manage free blocks.**
  - Binary splitting.
    - ✓**Half of the block is allocated.**
    - ✓**The two halves are called "buddy blocks".**
    - ✓**Can coalesce again when two buddy blocks are free.**
  - This scheme is also known as Quick Fit.

Buddy Allocation



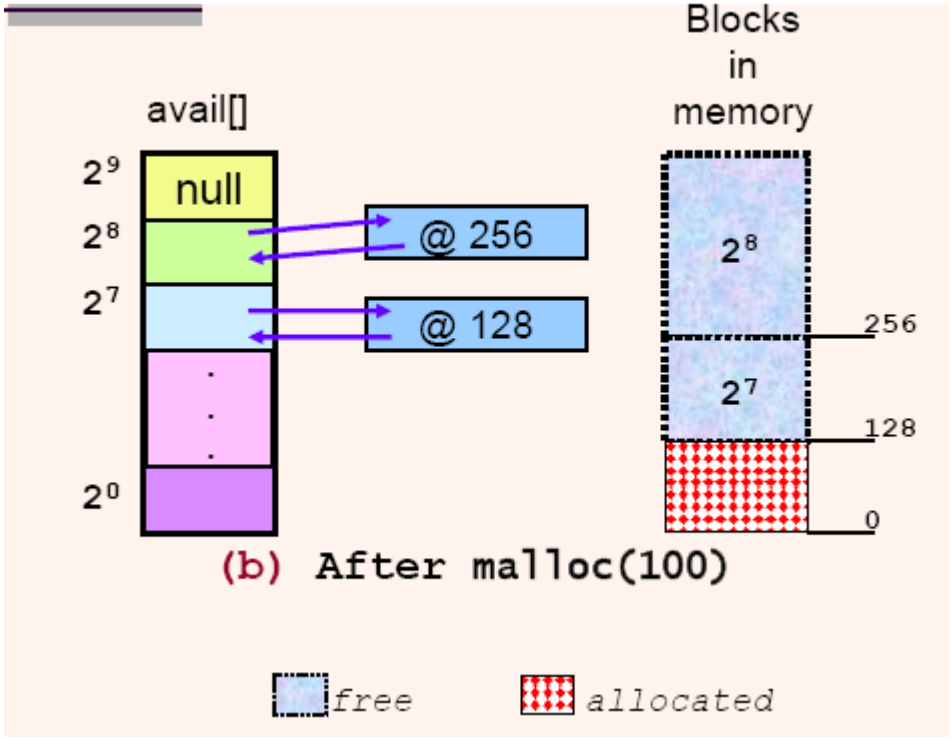
a. initial (one single free block, maximum free memory = 512, start address at 0)

- avail[i]: link list of free blocks of size 2^i (doubly linked in this eg.)
- avail[i] = NULL means no free blocks of that size
- avail[0] for illustration since doesn't make sense to have management for 1 byte free blocks

Note: can generalize starting address to be non 0

Buddy Allocation

- **Example:**
  - We initially have one free block of 512 bytes.
  - We want to do malloc(100)
  - Steps:
    - ✓**Split 512 byte block into 2 blocks of 256 bytes.**
    - ✓**Split one 256 byte block into two 128 byte blocks.**

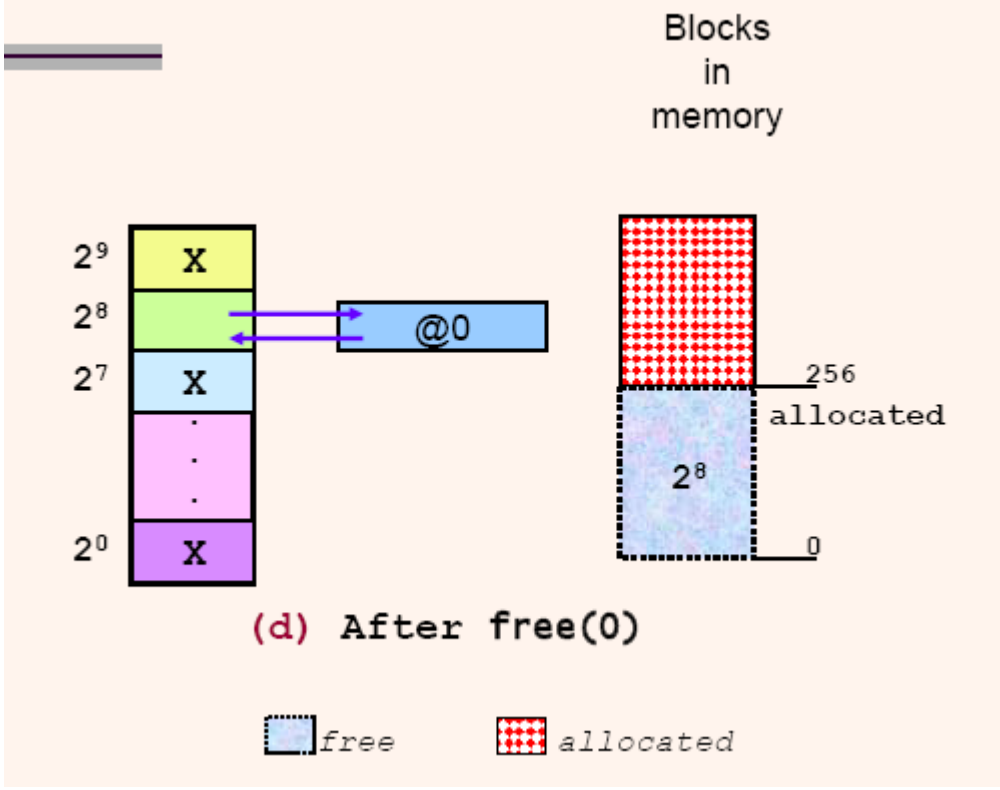


Buddy Allocation

- **These allocations have created blocks at addresses:**
  - 0 to 127: the block given to the malloc, since we return the first free block of at least 100 bytes.
  - 128-255: The free buddy block of the block at address 0.
  - 256-511: The free buddy block of the block that was broken up to 0-127 and 128-255.
- **Suppose now we want to do malloc(256).**
  - Block at address 256 is returned.

Buddy Allocation

- **We now call free(0):**
  - This frees up addresses 0 to 127.
  - Coalesces with its buddy block to form a single 256 byte block at address 0.



IT5002

Computer Systems and Applications

Lecture 17

File Systems

colintan@nus.edu.sg

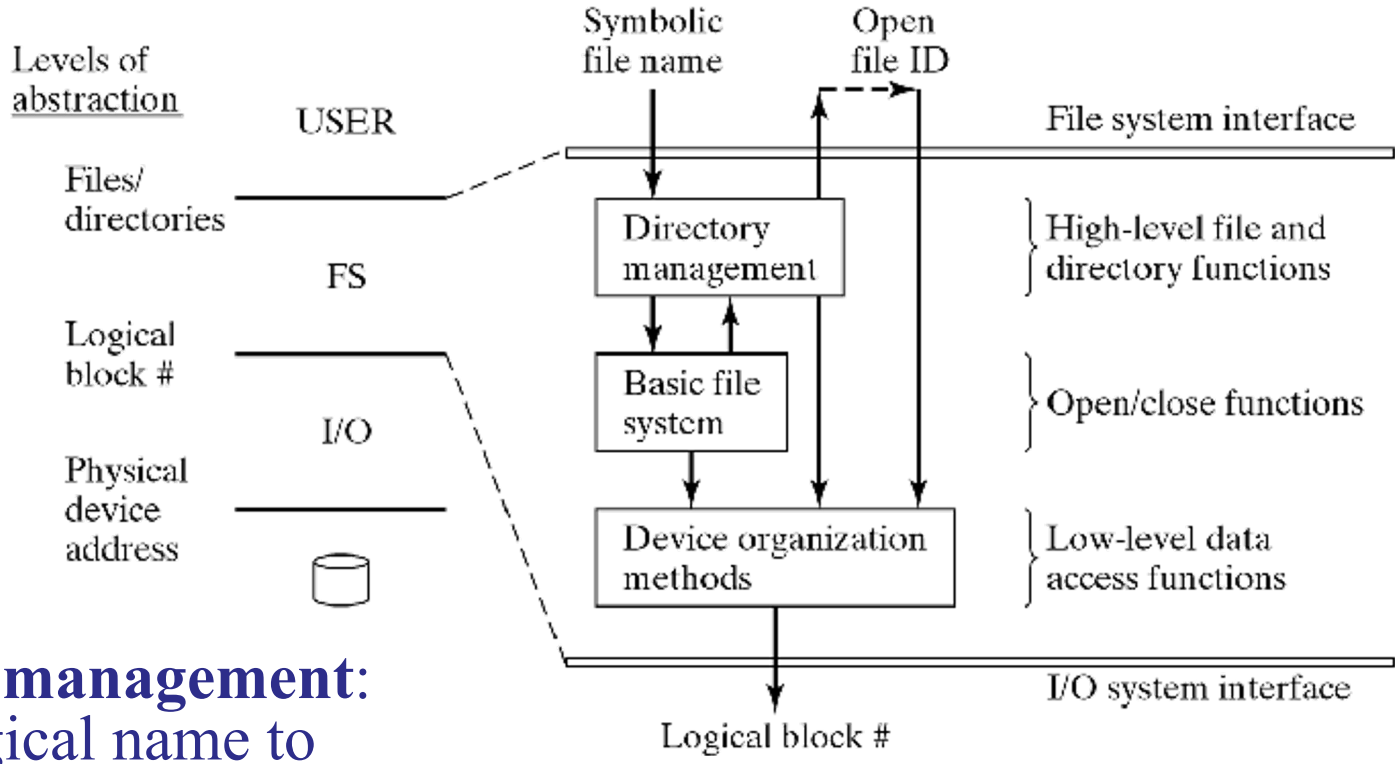
NUS
National University
of Singapore

School of Computing

What is a File System?

- **Present logical (abstract) view of files and directories**
  - Accessing a disk is very complicated:
    - ✓**2D or 3D structure, track/surface/sector, seek, rotation, …**
  - Hide complexity of hardware devices
- **Facilitate efficient use of storage devices**
  - Optimize access, e.g., to disk
- **Support sharing**
  - Files persist even when owner/creator is not currently active (unlike main memory)
  - Key issue: Provide protection (control access)

Hierarchical View of File Systems



Directory management:
- map logical name to unique Id, file descriptor

Basic file system:
- open/close files

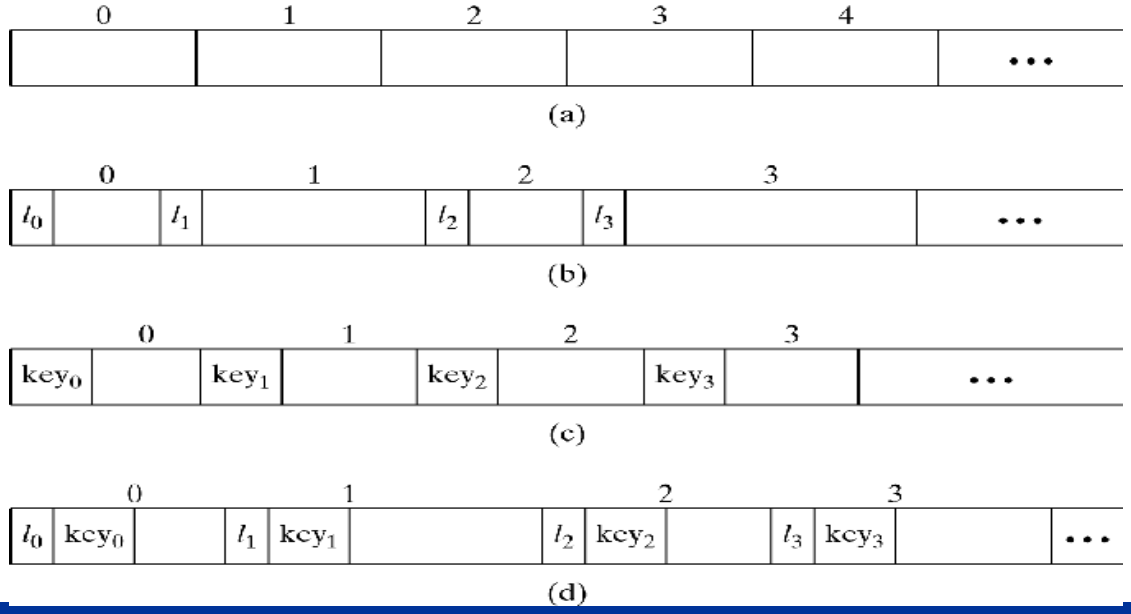
Physical device organization:
- **map file data to disk blocks**

User's View of a File

- **File name and type**
  - Valid **name**
    - ✓**Number of characters**
    - ✓**Lower *vs* upper case, illegal characters**
  - **Extension**
    - ✓**Tied to type of file**
    - ✓**Used by applications**
  - File **type** recorded in header
    - ✓**Cannot be changed (even when extension changes)**
    - ✓**Basic types: text, object, load file, directory**
    - ✓**Application-specific types, e.g., .doc, .ps, .html**

User's View of a File

- **Logical file organization**
  - Most common: **byte stream**
  - Fixed-size or variable-size **records**
  - Addressed
    - ✓**Implicitly (sequential access to next record)**
    - ✓**Explicitly by position (record#) or key**



a) Fixed Length Record

b) Variable Length Record

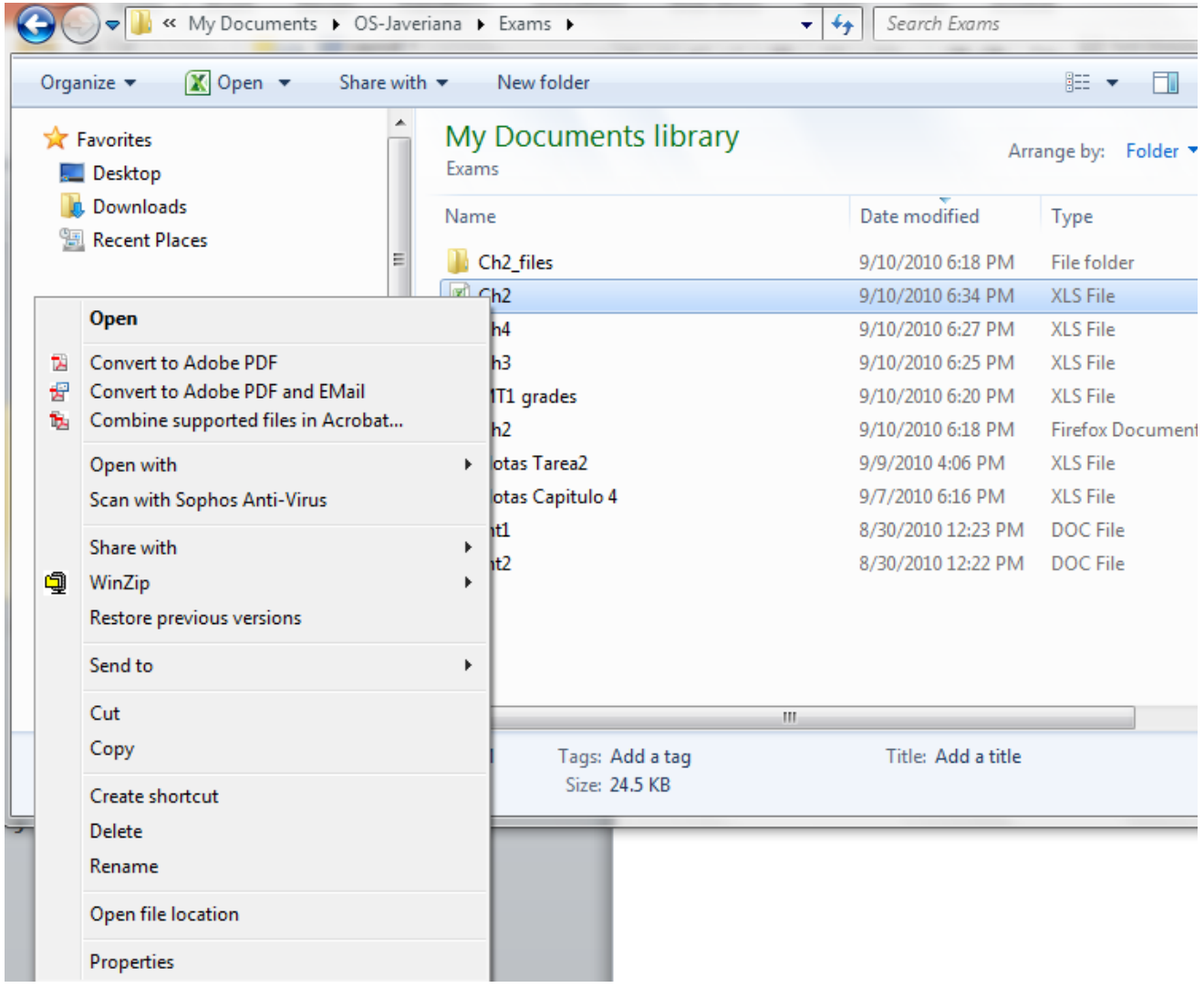
c) Fixed Length with Key

d) Variable Length with Key

Operations on Files

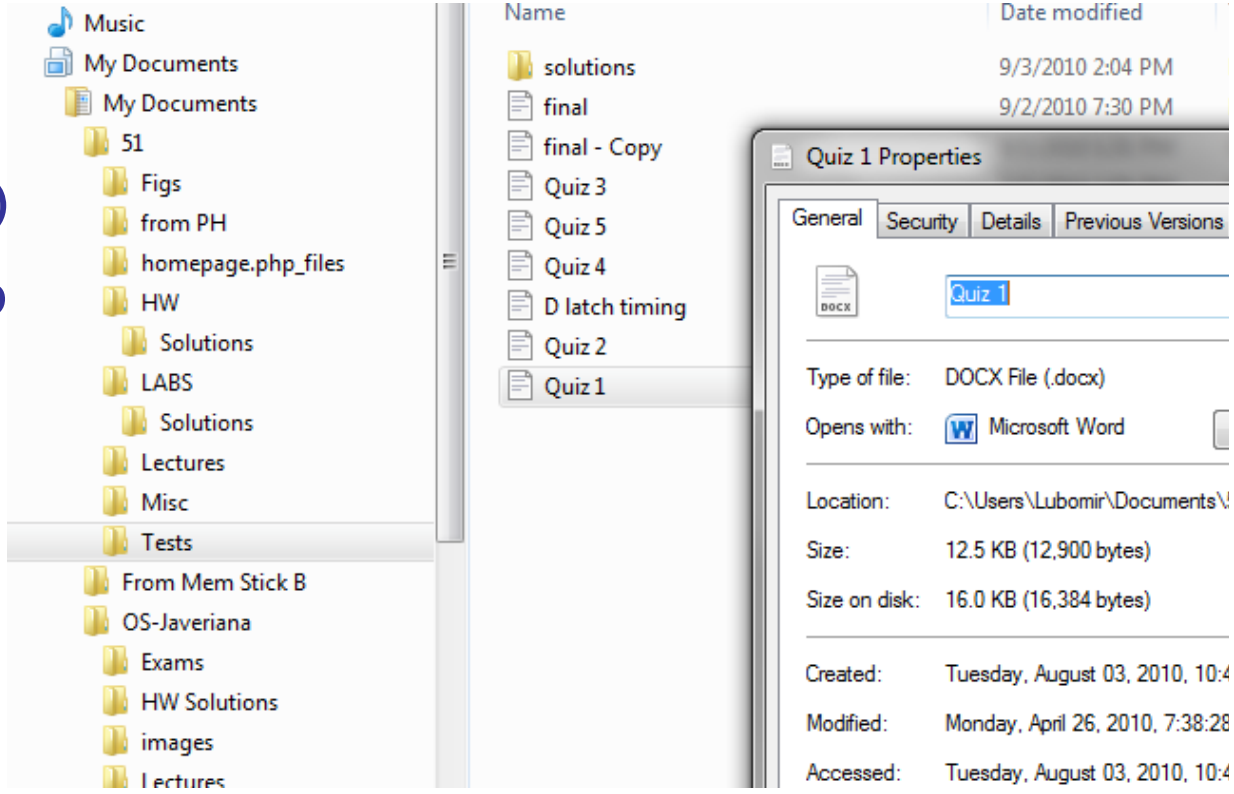
- **Create/Delete**
- **Open/Close**
- **Read/Write (sequential or direct)**
- Seek/Rewind (sequential)
- **Copy (physical or just link)**
- **Rename**
- **Change protection**
- **Get properties**
- **Each involves parts of Directory Management, BFS, or Device Organization**
- **GUI is built on top of these functions**

Operations on Files



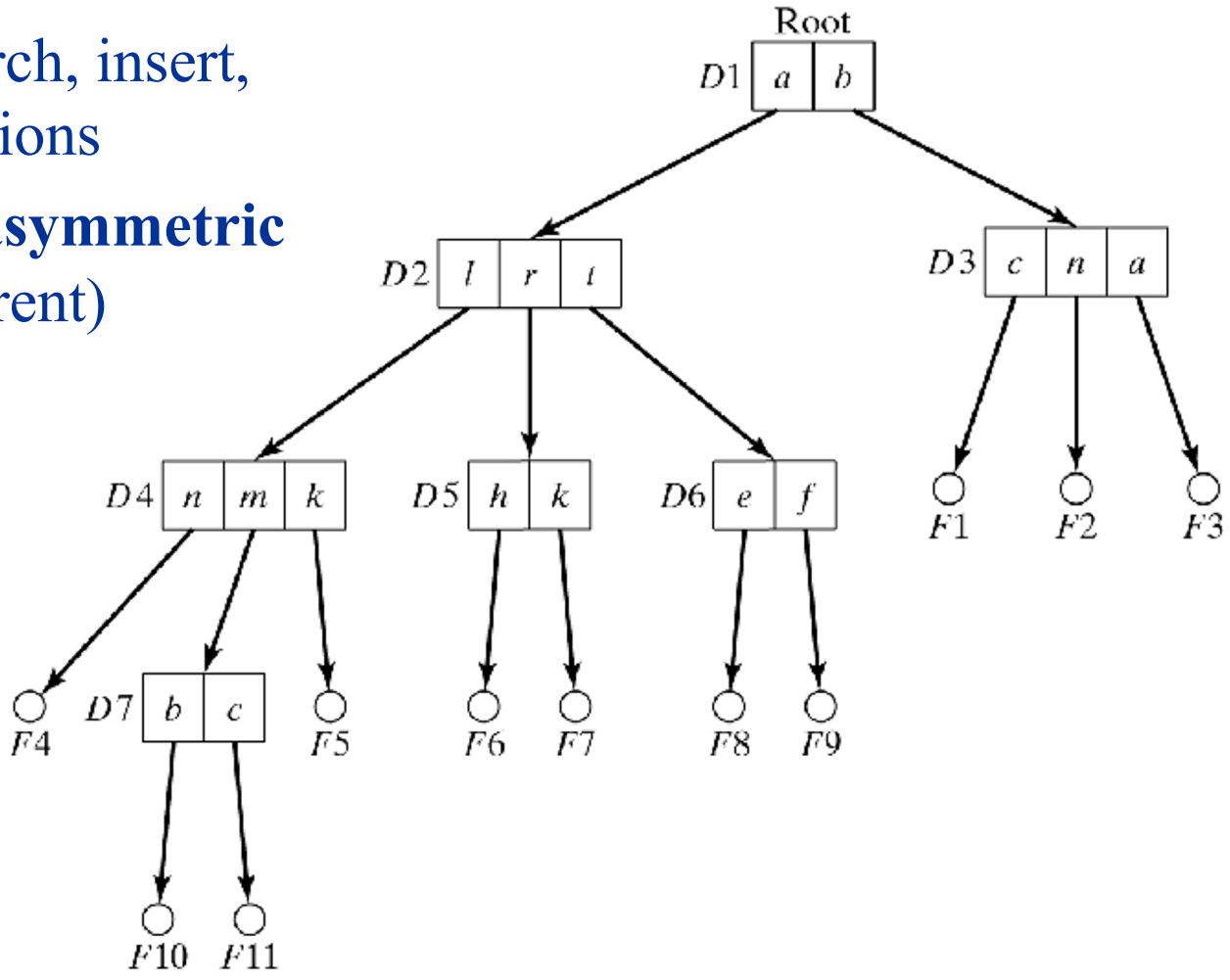
Directory Management

- Main issues:
  - **Shape** of data structure (e.g. tree, shortcuts, …)
  - **What** info to keep about files
  - **Where** to keep it? (in directory?)
  - **How** to organize entries for efficiency?



File Directories

- **Tree-structured**
  - **Simple** search, insert, delete operations
  - Sharing is **asymmetric** (only one parent)



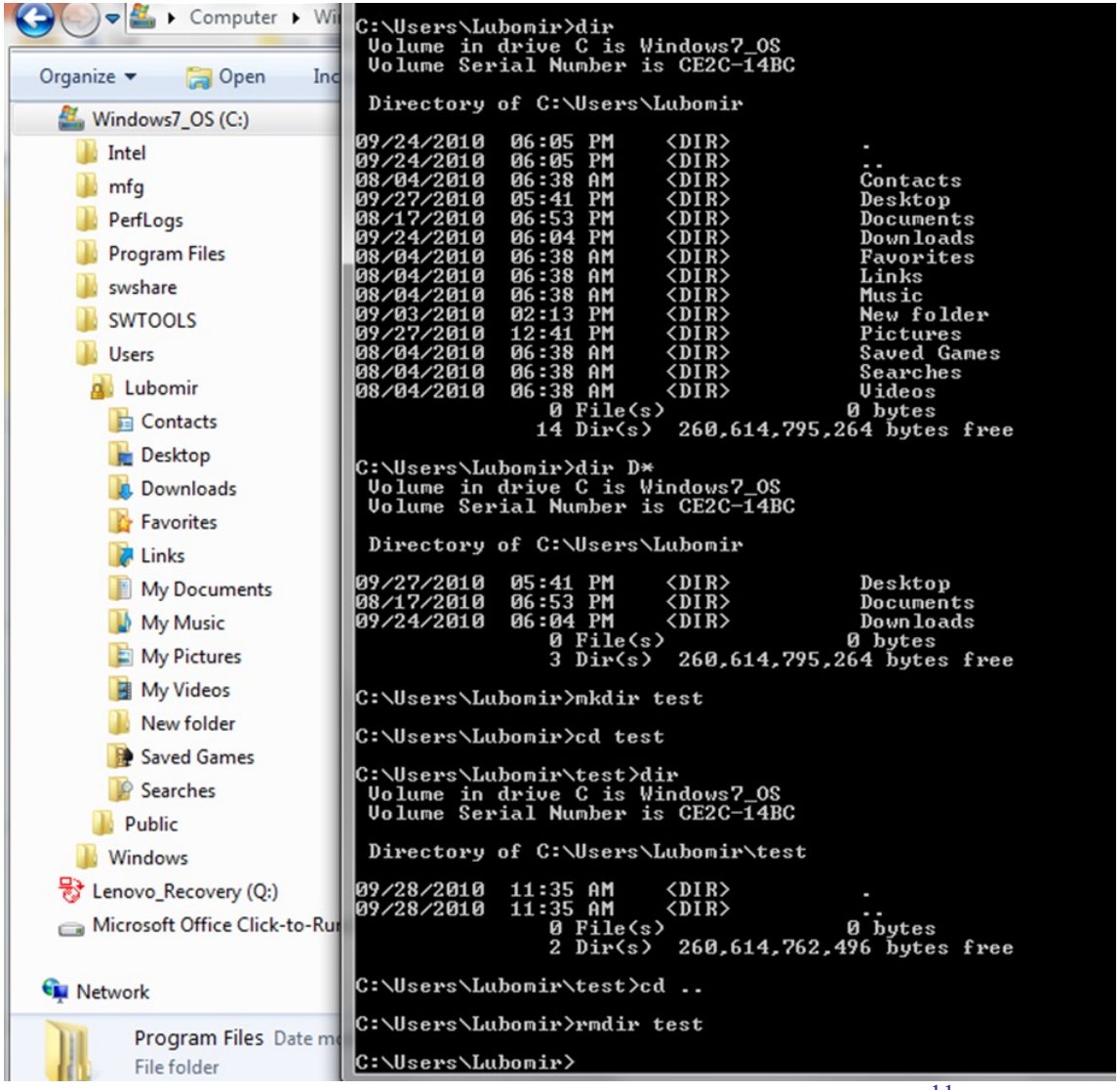
File Directories

- **How to uniquely name a file in the hierarchy?**
- **Path names**
  - Concatenated local names with delimiter:
    (. or / or \ )
  - **Absolute** path name: start with root
    (/)
  - **Relative** path name: Start with current directory
    (.)
  - Notation to move upward in hierarchy
    (. .)

NUS National University of Singapore
School of Computing

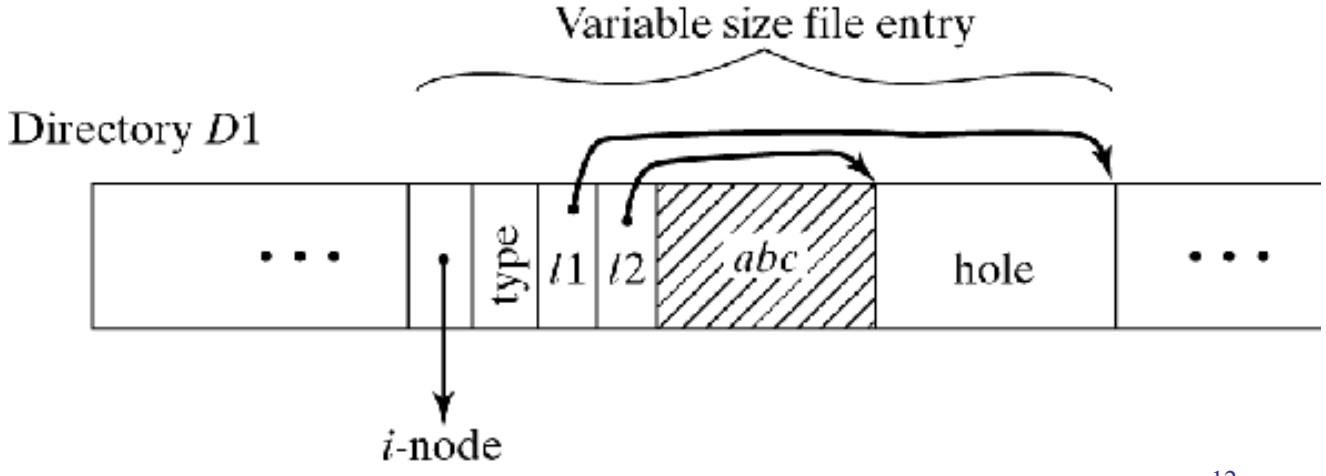
Operations on File Directories

- **GUI vs commands**
  - Create/delete
  - List: sorting, wild cards, recursion, information shown
  - Change (current, working) directory
  - Move
  - Rename
  - Change protection
  - Create/delete link (symbolic)
  - Find/search routines



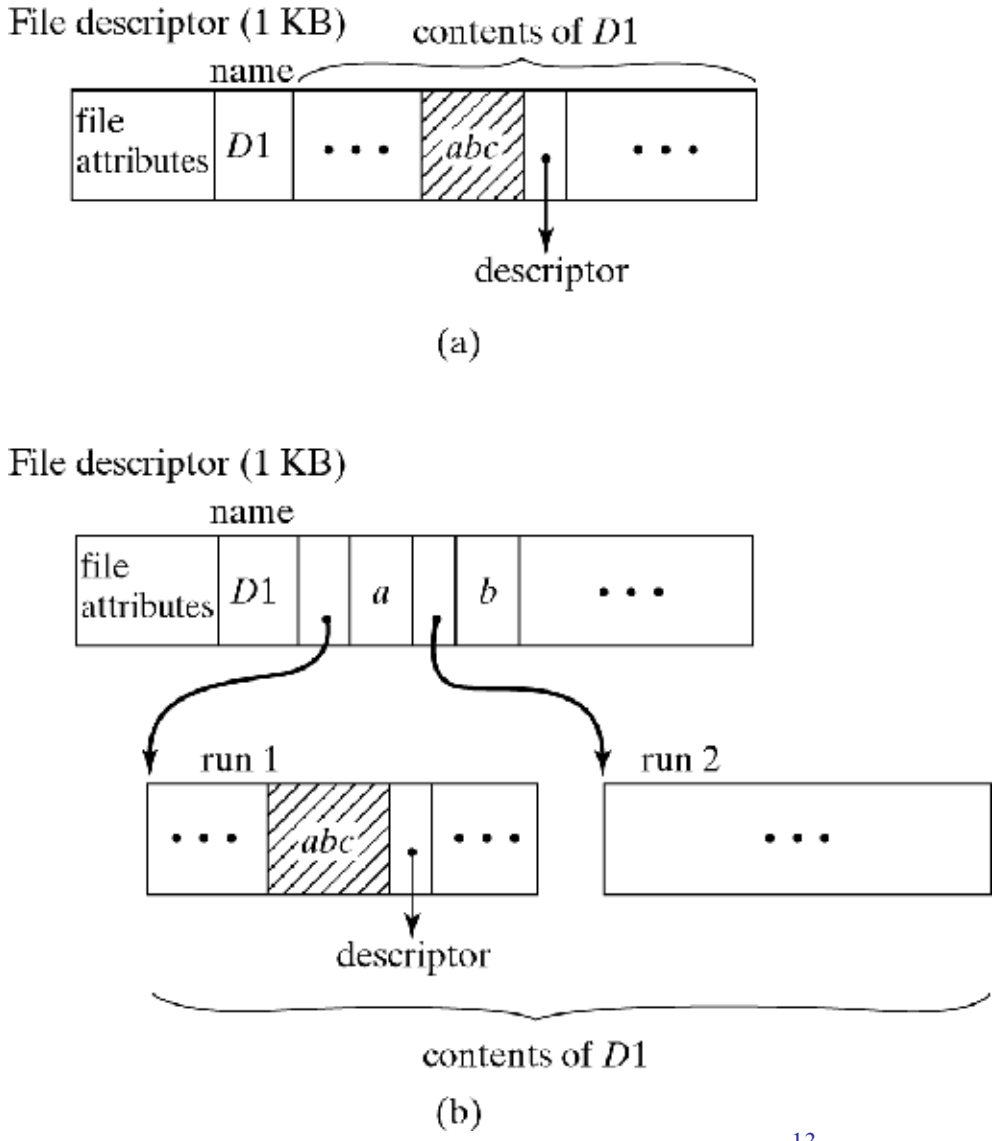
Implementation of Directories

- **What information to keep in each entry**
  - **All** descriptive information
    - ✓**Directory can become very large**
    - ✓**Searches are difficult/slow**
  - Only symbolic **name and pointer** to descriptor
    - ✓**Needs an extra disk access to descriptor**
    - ✓**Variable name length?**



Implementation of Directories

- **How to organize entries within directory**
  - **Fixed-size** entries:
    use array of slots
    - **Variable-size** entries:
      use linked list
  - **Size of directory**:
    fixed or expanding
- **Example: Windows 2000**
  - when # of entries exceeds directory size, **expand** using B^+-trees



NUS National University of Singapore
School of Computing

Revisit file operations

- **Assume:**
  - descriptors are in a dedicated area
  - directory entries have name and pointer only
- **create**
  - find free descriptor, enter attributes
  - find free slot in directory, enter name/pointer
- **rename**
  - search directory, change name
- **delete**
  - search directory, free entry, descriptor, and data blocks
- **copy**
  - like create, then find and copy contents of file
- **change protection**
  - search directory, change entry

Basic File System

- **Open/Close files**
  - Retrieve and set up information for **efficient** access:
    - ✓**get file descriptor**
    - ✓**manage open file table**
- **File descriptor (i-node in Unix)**
  - Owner id
  - File type
  - Protection information
  - Mapping to physical disk blocks
  - Time of creation, last use, last modification
  - Reference counter

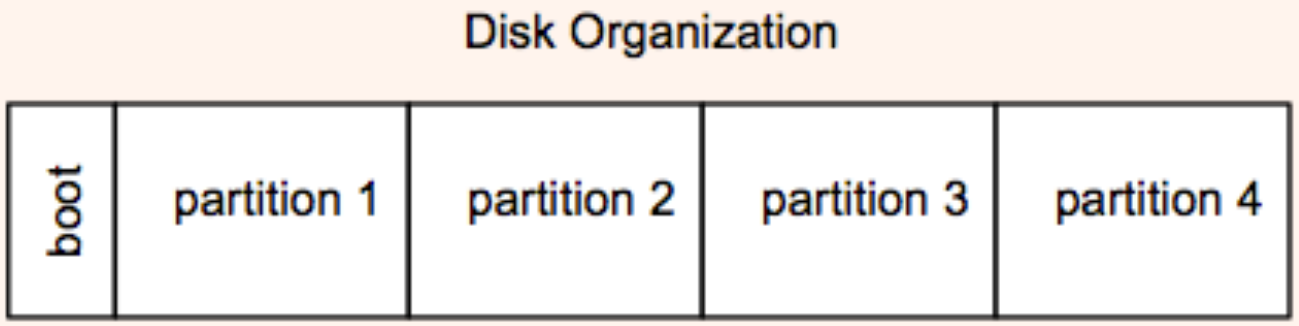
Basic File System

- **Open File Table (OFT) keeps track of currently open files**
- **open command:**
  - Search directory for given file
  - Verify access rights
  - Allocate OFT entry
  - Allocate read/write buffers
  - Fill in OFT entry
    - ✓**Initialization (e.g., current position)**
    - ✓**Information from descriptor (e.g. file length, disk location)**
    - ✓**Pointers to allocated buffers**
  - Return OFT index

Basic File System

- **close command:**
  - Flush modified buffers to disk
  - Release buffers
  - Update file descriptor
    - ✓**file length, disk location, usage information**
  - Free OFT entry

Organizing Data on a Disk



Typical PC hard disk organization

Boot: usually start of disk which contains code to start loading the OS (booting the OS). For PC, boot block also called Master Boot Record (MBR) and contains also the partition table

Partitions: divide up disk into regions for filesystems, each filesystem in one partition. For PCs, only 4 primary partitions are allowed which can be booted

Note: details are PC specific, also can have logical partitions beyond 4 primary partitions, etc.

Organizing Files

- basic unit is a sector/block in a partition (compare with page in VM)
- logical view of file implementation – collection of logical blocks (possibly last block may be partial since file data may be ≤ block size)
- logical blocks need not be same as physical sector size – may be some multiple of sectors (Why? consecutive I/O is the fastest)
- Simple schemes – minimum unit is 1 block – file size < 1 block has internal fragmentation, more complex schemes can share multiple small files in 1 block (we wont look at this)!

Notes: mostly we treat file as collection of bytes, MSDOS logical block is called cluster (which can be large!)

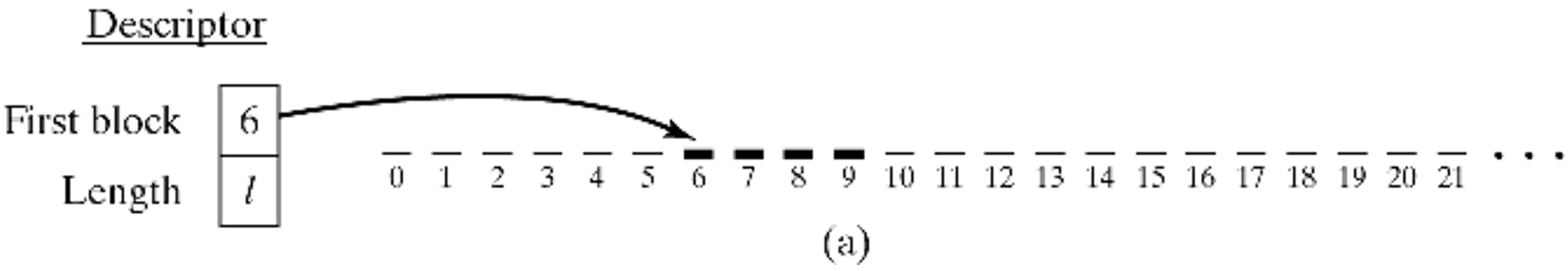
Data Structures on Disk

- need data structure to record which block belongs to which part of file, eg. data at positions 2020-4100 are in which blocks and which part of the block
- How does data in file change?
  - write more data at end of file
  - Unix: write data into holes! But basically means can write anywhere!
  - decreases with truncation operation (unlike mem no free op!)
- data structure must also be stored on disk (persistent)
- typical data structures: versions of list / trees / arrays

Notes: very similar to data structures for dynamic memory management and page tables

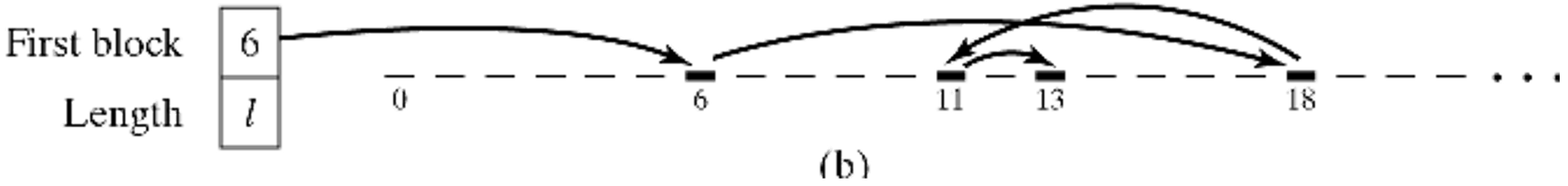
Physical Organization Methods

- **Contiguous organization**
  - Simple implementation
  - Fast sequential access (minimal arm movement)
  - Insert/delete is difficult
  - How much space to allocate initially
  - External fragmentation



Physical Organization Methods

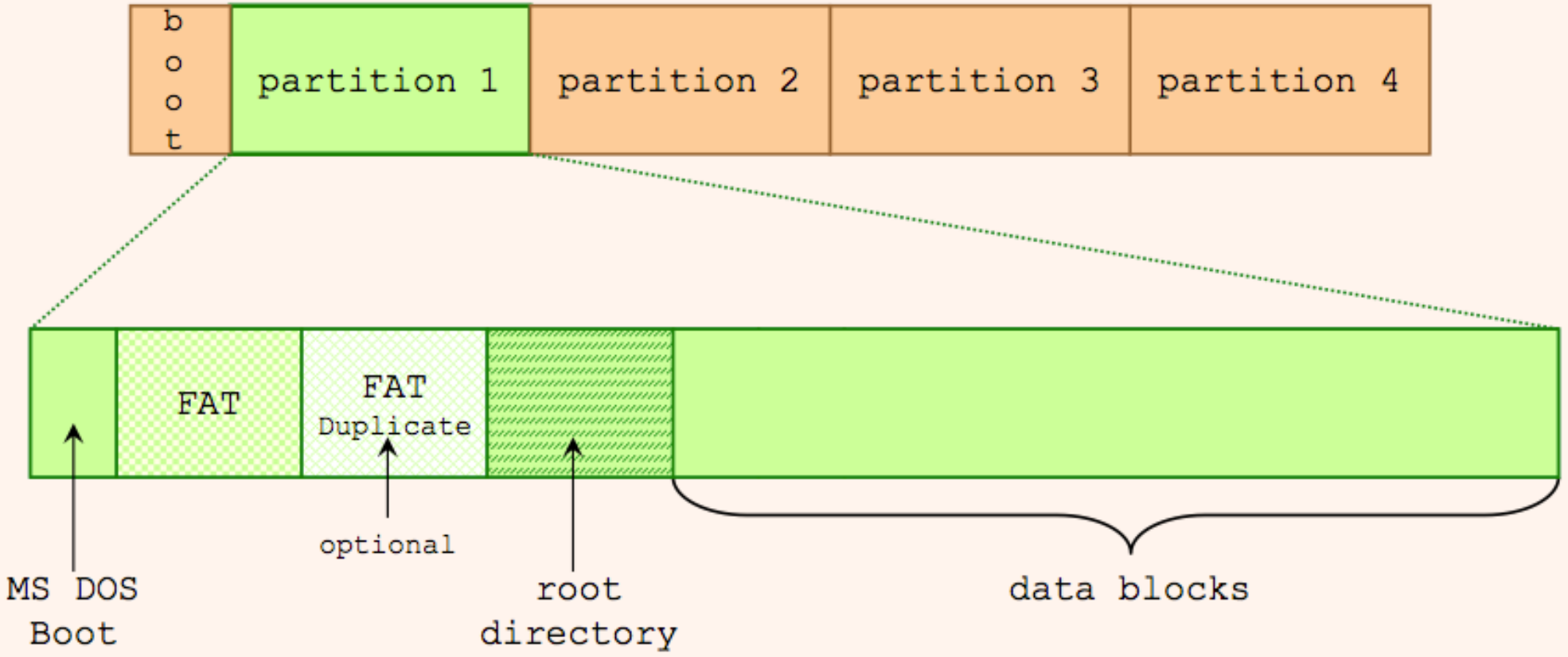
- **Linked Organization**
  - Simple insert/delete, no external fragmentation
  - Sequential access less efficient (seek latency)
  - Direct access not possible
  - Poor reliability (when chain breaks)



Linked List Allocation - FAT

- MSDOS uses File Allocation Table (FAT)
- Linked allocation but stored completely in FAT (after reading from disk)
- FAT kept in **RAM** (stored in disk but duplicated in RAM) – gives fast access to the pointers
- FAT table contains either: **block number** of next block, **EOF** code (corresponds to NULL pointer), **FREE** code (block is unused), **BAD** block (block is unusable, i.e. disk error) – combines bitmap for free blocks with linked allocation for list of blocks
- FAT table is 1 entry for every block. Space management becomes an array method (but bigger than bitmap)

MSDOS File System Layout

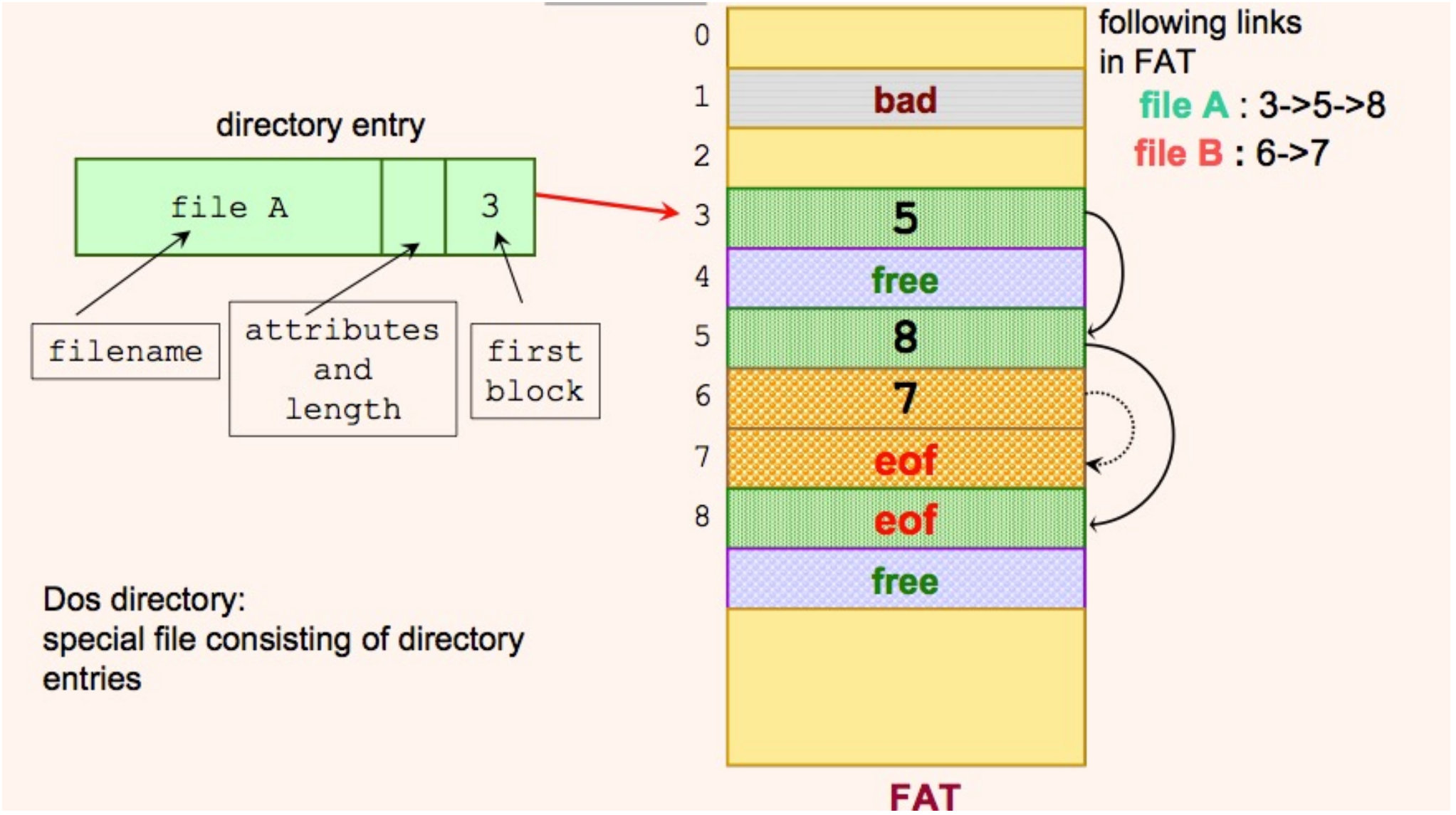


Notes: FS corresponds to 1 volume (logical drive, eg. C:), stored in 1 partition, eg: another drive (D:) in another partition etc. We dont cover extended partition

DOS Directory

- special file (type directory) containing directory entries (32 byte structures, little endian)
- **fat directory entry**: **filename + extension** (8+3 bytes), file **attributes** [1 byte: readonly, hidden, system, **directory flag** (distinguish directories from normal files), archive, volume label (disk volume name is dir entry)], time+date created, last access (read/write) date, time+date last write (creation is write), **first block (cluster) number**
- root directory is special (already known, FAT16 has limited root dir size, 512 entries), other directories distinguished by type

FAT Organization



FAT Organization

- Linked list implemented as array of FAT entries:

  4 types: block number (part of a linked list), EOF type (end of linked list), special FREE type, special BAD type – some numbers are block numbers, some are reserved block numbers for EOF,FREE,BAD
- Blocks for file linked together in FAT entries, eg. file A: 3 -> 5 -> 8
- Free blocks indicated by FREE entry
- Bad blocks (unusable blocks due to disk error) marked in FAT entry: BAD cluster value

MSDOS: Deleting a File

How MSDOS deletes file/directory:

- Set first letter in filename to 0xE5 (destroys first byte of original filename)
- Free data blocks: set FAT entries in link list to FREE (tags all the data blocks in file free)

Can attempt to **undelete** – some information lost but can guess!

FAT16 Cluster Size

- FAT16: fat entry block # is **16 bits** (16 bit numbers in FAT entries), sector size=512, how to deal with disks bigger than 64K*512 (32M)
- Logical block size = multiple of sectors. MSDOS calls this the **cluster size**
- Maximum cluster size = 32K (normally)
- Maximum file system size for FAT16 = 64K*32K = 2G
- Maximum file size: slightly less than 2G
- large cluster size means large internal fragmentation!

Note: can use 64K cluster size, so limits become 4G

VFAT

- VFAT: adds long filenames (255 chars, introduced in Win95)
  - compatible with FAT16 by tricking it!
  - short FAT16 filename for FAT16 and long version, short filename created from long one (e.g. `longna~1.doc`)
  - for compatability: long name stored as multiple directory entries using illegal attributes (not used by FAT16)
  - trick: have to manage 2 kinds of names, old SW uses short names, aliasing issue due to 2 names

FAT32

- Increase FAT size to 28 bits, cluster numbers 28-bits
- max filesystem size increased: Win98 ~127G limit, Win2K can only format up to 32G limit
- decreases internal fragmentation (cluster size can decrease)
- FAT table size increases
- FAT16 root directory size limit removed – normal directory
- Maximum file size 2^32 - 1

Notes: compare with direct paging, changing size of page and effect on page table given different sized virtual addresses

Disk Fragmentation

Fast disk access:

- contiguous blocks (from geometry/processing viewpoint)
- blocks in same cylinder

Suppose currently FS is optimally allocated (fresh install). Is this sustainable?

1. Delete files/blocks
2. Insert new files/blocks

After some operations – block ordering becomes more **random**!

Disk Fragmentation: logical contiguous blocks are "**far apart**" on disk (this is different from memory fragmentation)

Notes: internal fragmentation still exists from block size

Disk Fragmentation

FAT:

- affects FAT FS
- fragmentation effect less with large cluster size (but **large** internal fragmentation!)
- MSDOS solution: run defragmentation (like compaction) on entire FS – move all used blocks to be contiguous .. one big free space chunk after defragmentation. May take a long time to defrag! (Windows: Disk Defragmenter)

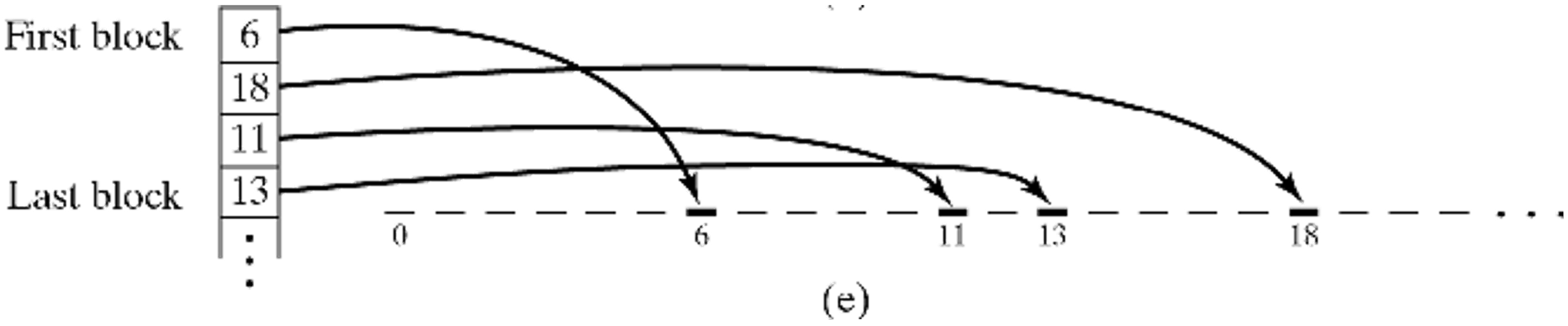
Unix S5FS:

- also has disk fragmentation
- may be worse than DOS since smaller logical block size

Alternative (not covered): FS with **fragmentation resistance** (not necessarily optimal but dont need to defragment).

Physical Organization Methods

- **Indexed Organization**
  - Index table: sequential list of records
  - Simplest implementation: keep index list in descriptor



  - Insert/delete is easy
  - Sequential and direct access is efficient
  - Drawback: file size limited by number of index entries

Physical Organization Methods

- **Variations of indexing**
  - **Multi-level** index hierarchy
    - ✓**Primary index points to secondary indices**
    - ✓**Problem: number of disk accesses increases with depth of hierarchy**
  - **Incremental** indexing
    - ✓**Fixed number of entries at top-level index**
    - ✓**When insufficient, allocate additional index levels**
    - ✓**Example: Unix, 3-level expansion (see next slide)**

UNIX File Systems

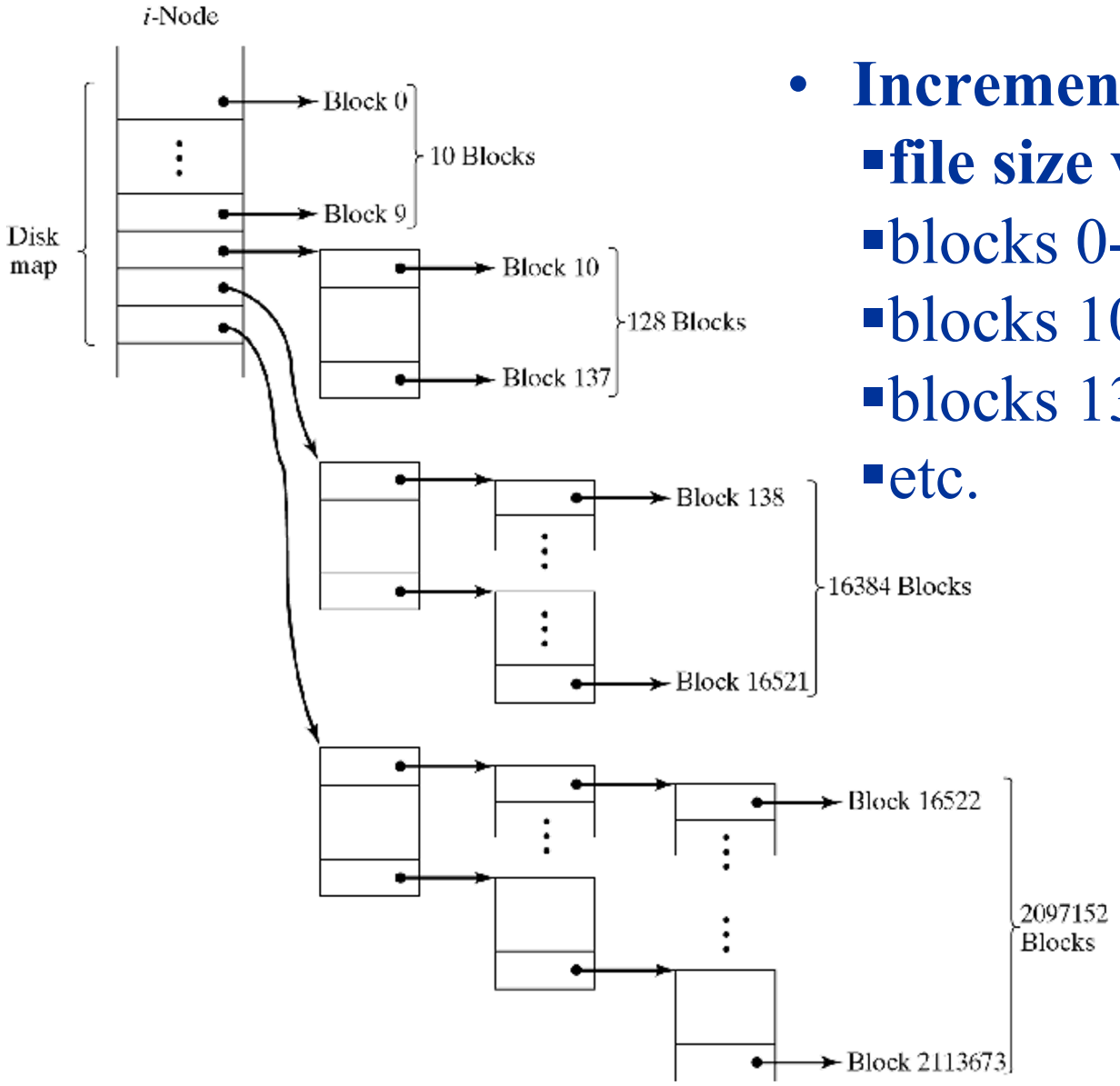
- Look at System V File System (s5fs) – simpler than modern FS implementations (rather old but some of the design remains in current implementations)
- inodes: represents every file
- directories: contain names of files (recall maps filename to eventual file (hard link) or pathname (symbolic link))
- file allocation using a multi-level tree index

Note: s5fs is the original Unix v7 FS. It's much simpler and less sophisticated than more modern Unix FS.

s5fs: Inodes

- actual file object
- every file has one inode (many to one mapping because of hard links)
- contains all meta-data about file **except filename**
- contains reference count i.e. # hard links, reference count = 0 means file can be deleted [subjected to no open files condition - all file descriptors to file object are closed]
- meta-data in inode includes Table of Contents (TOC) which gives mapping of file data to disk blocks (TOC is per file - contrast with MSDOS which has only **global** TOC (FAT))

Unix Table of Contents (TOC)



- **Incremental indexing in Unix**
  - **file size vs. speed of access**
  - blocks 0-9: 1 access (direct)
  - blocks 10-137: 2 accesses
  - blocks 138-16521: 3 acc.
  - etc.

s5fs: TOC Index Blocks

- Direct block pointers:

  used for small files, no extra disk overhead, efficient direct access. VM analogy: TLB (however not a cache)

- Single indirect block:

  files bigger than direct blocks & smaller than double indirect. Disk overhead is 1 block. File access slightly slower than direct. VM analogy: direct mapped page table

- double + triple indirect blocks:

  files bigger than single indirect block (for sufficiently big block size, triple indirect usually not needed) . More disk overhead but is small fraction of file size. Random file access requires looking up the indirection blocks – slower than indirect. VM analog: 2-3 level page tables

File sizes: direct << single indirect << double indirect << triple indirect

s5fs: File System Parameters

- # of direct blocks (eg. s5fs: 10, ext2: 12)
- # indirection levels (eg. max is 2 or 3)
- logical block size - determines efficiency of disk I/O, can be composed of several contiguous physical blocks, space wastage from internal fragmentation, determines # block pointers in index blocks and max indirection levels (eg. ext2 can select 1,2,4K when creating filesystem)
- block pointer size - affects indexing, determines max addressable disk block)

s5fs: File System Parameters

- small files only use the direct blocks in TOC, number of direct blocks can vary
- larger file requires first indirect block (this is like 1-level page table)
- even larger file uses double indirect block which points to indirect blocks (similar to 2-level page table). Even larger may have triple indirect (but the number of indirection levels may vary)
- like page tables, can allow blocks which do not exist (**logical zero filled holes** in the file) – set the block pointer to NULL in the TOC, return zeroes for the logical block when read

s5fs: Directories

- a file of type directory – accessing directory is like any other file
- only special directory operations allowed (read directory, link + unlink filenames)
- s5fs directory is array of 16 byte entries (inode: 16 bits, filename: 14 bytes), simplifies directory management (note dont need null pointer if filename length = 14)
- deleted file has inode 0
- note: most modern implementations allow long file names (ext2: 255-byte filenames), so space issues like in memory allocation

Inode Number	Filename
25	.
71	..
100	file1
200	file2

s5fs: Link + Unlink

- Create new file: new directory entry with new inode
- Hard link: new directory entry (in appropriate dir) with inode of the linked file: eg. `link(path1, path2)`

  i1 = inode for file with path1;

  *let path2 = dir2/***file2**; // dont allow linking directories

  add new directory entry to dir2: **[i1, file2]**; // assume file2 // not there

- Deleting (deleting is unlink since graph is DAG): remove directory entry, decrement inode link count, free file object when link count = 0 (plus open fd's condition)

NUS National University of Singapore
School of Computing

Free Storage Space Management

- **Similar to main memory management**
- **Linked list organization**
  - Linking **individual** blocks -- inefficient:
    - ✓**no block clustering to minimize seek operations**
    - ✓**groups of blocks are allocated/released one at a time**
  - Better: Link **groups** of consecutive blocks
- **Bit map organization**
  - Analogous to main memory
  - A single bit per block indicates if free or occupied