Chapter 1
Introduction
General Aspects Regarding Operating Systems

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Colesa

February 24th, 2021

1.1

Part I
Course Presentation and Administrivia

1.2

1 Presentation

Useful Information

1. Lecture classes (2 hours / week): Adrian Colesa
2. Lab classes (2 hours / week / semigroup)
   - Balint Szabo for group 30421/1
   - Andrei Seicean for group 30421/2
   - Lazslo Ciople for group 30422/1
   - Adrian Colesa for group 30422/2
   - David Acs for group 30423/1
   - Istvan Csaszar for group 30423/2
   - Andrei Crisan for group 30424/1
   - Adrian Buda for group 30424/2
3. OS Course's Web page: `https://moodle.cs.utcluj.ro/course/view.php?id=354`
   - create user account, if not having one yet
     - pay attention at what **firstname** and **surname** (familyname) are
   - register for the "*Operating Systems, Spring 2021*" course
   - enrollment keys
     - for semi-group 3042N-M: *os-2021-GR-N-M*, $N \in 1,4$, $M \in 1,2$
     - for students that re-attend the course: *os-2021-Retaken*

1.3

Purpose and Objectives

- general purpose
  - make students understand the fundamental concepts and functionality of modern OSes
- specific objectives

1. understand the OS's role and its components' functionality
2. be familiar with OS' major services (system calls)
3. have general knowledge about the internal mechanisms of an OS

- methods
  - presents the most important components of an OS from two points of view: **external** (interface) and internal (design)
  - problem solving
  - practice (write C programs) on a real OS: ***Linux***

1.4

Contents

1. Introduction. General Aspects Related to Operating Systems
2. The Command Interpreter
3. The File System (2 parts)
4. Process Management
5. Thread Management
6. Synchronization (2 parts)
7. Memory Management (3 parts)
8. Inter-Processes Communication (IPC) Mechanisms
9. Protection and Security

1.5

Docs sources

Books

- **A. Tanenbaum, *Modern Operating Systems*, 4nd Edition, Pearson, 2014**
- A. Silberschatz, P. Galvin, G. Gagne, *Operating Systems Concepts*, 9th Edition, Wiley, 2012
- Michael Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010
- M. Mitchell, J. Oldham, A. Samuel, *Advanced Linux Programming*, New Riders Publishing, 2001

On-line

- Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, online available at `http://pages.cs.wisc.edu/~remzi/OSTEP/`

1.6

Acquired Students' Competences and Skills

1. be able to explain the role of the OS in a computing system
2. be able to define and explain fundamental OS concepts
3. be able to use basic Linux commands
4. be able to write C programs to use Linux system calls

1.7

2 Requirements and Policies

Prerequisites

1. basic knowledge about computing systems
2. C programming

1.8

Students' and Teacher's Responsibilities

1. interest for the subject
   - **trust me**: it is fundamental and really interesting
2. receptiveness to the other side challenges
   - **ask** (*use the forum, chat!*), **answer**, **propose**
3. **perseverance** during the entire semester
4. **balance the effort**, be in time to scheduled deadlines
   - use the night before the deadline / exam for sleeping!

1.9

Attendance and Recovery Policy

1. **this is not about planning "*what and how much to miss (skip)*"**
   - our activities are meetings, based on communication, not possible without the presence of both parties
   - all that I and my colleagues prepare(d) is **for you**
   - a meeting and a discussion is a living action: you are given not just information, but also personal experience
   - reading at home is not communication and could by far be not so effective!
   - though, reading and learning by him/herself is required and important, obviously
2. attendance policy's terms and rules are really very, very, very strict
   - exceptional cases must be discussed in time, not just at the end of semester

1.10

Attendance and Recovery Policy (cont.)

1. attending lecture classes
   - minimum 7 to be allowed to take examinations
   - *below 7 $\Rightarrow$ must retake the course next year!*
   - **there is no possibility to recover missed classes**
2. attending lab classes
   - maximum 2 missing labs to be allowed to take the (summer) lab examination
   - maximum 4 missing labs to be allowed to take the lab re-examinations
   - *more than 4 $\Rightarrow$ must retake the course next year!*

1.11

Attendance and Recovery Policy (cont.)

1. recovery policy for labs
   - **absences are not removed!**
   - missed classes should be recovered in lab rooms (extra time with other groups)
     - send solutions to proposed problems to your lab teacher
   - maximum 2 per exam session, i.e.
     - 2 missed labs could be recovered until one week before the end of semester
     - 2 extra missed labs could be recovered in any re-examination session this year

1.12

Assignment Lateness and Recovery Policy

- 3 lab assignments
- two deadlines for each assignment
  - 1st one is soft and optional (on Tuesday at 1 am)
  - about one week between them
  - assignment could be resubmitted on the 2nd deadline, if not please with received grade
  - 2nd one is hard (on Sunday at 23:55)
- assignment recovery
  - only in re-examination sessions

1.13

Plagiarism and Cheating Policy

- **cheating is not allowed and not accepted!**
- if **anyone** found guilty of something like this will not be allowed to take the exam in any exam session!
- we use an application that **checks against plagiarism** your submitted solutions
  - between them
  - against submissions from all previous years
- this is really very, very, ... strict
- if you need help, ask it in time from us (your teachers)
- **never show, give or make public your code**
  - excepting to your teachers
- read more details in the "Examination and Plagiarism Policy" on the course page

1.14

What I Like

- work with enthusiastic people (students)
- see interested, friendly faces
- have a feedback from those I work with (teach)
- be asked good questions
- learn about interesting (useful) thinks
- ...

1.15

What I Do Not Like

- teach someone things he/she thinks from the beginning (having a prejudice) is of no use or interest
- teach someone things he/she thinks from the beginning he/she knows better and everything about the subject
- on-site classes
  - talk to someone who is not looking at me
    - but in his/her laptop, phone etc.
  - be disturbed during my talk (presentation)
    - have question? ask them!
    - something not clear? ask about!
    - interesting idea related to OS? just say it loudly!
    - anything else? delay it for later!
- on-line classes
  - talk to someone not paying attention
  - ???
- ... not so many though!

1.16

3 Evaluation and Grading

Lab Evaluation

- quiz tests each lab class (about 10 minutes)
  - $\boxed{Lab\_QT_i \in [1,10], i = \overline{2,13}}$
- bonuses: $\boxed{Lab\_B_i \in [8,10], i = \overline{1,13)}}$ for
  - lab activity at each class
  - extra (special) problems

- $\boxed{Lab\_B = \sum_{i=0}^{13} \frac{max(0;6+(Lab\_B_i-8)*2)}{10}}$

- 3 lab assignments: $\boxed{Lab\_A_i \in [0,10], i = \overline{1,3}}$
  - one each 3-4 weeks (see the moodle page for deadlines)
  - consists in one problem needing about working 4-5 hours for being solved
- final test (last week): $\boxed{Lab\_P \in [0,10]}$
  - 2-3 problems to be solved on the computer

1.17

Lab Evaluation (cont.)

- Lab grade formula $\boxed{Lab = 0.10 * Avg(Lab\_QT_i) + 0.60 * Avg(Lab\_A_i) + 0.30 * (Lab\_P + Lab\_B)}$
- Conditions to pass
  - $Avg(Lab\_QT) >= 5$
  - $Avg(Lab\_A) >= 5$
  - $Lab\_P >= 5$
  - $Lab >= 5$

1.18

Lecture Evaluation

- short quiz tests AFTER each class (sort of homework)
  - $\boxed{Lect\_T_i \in [0,10], i = \overline{1,14}}$
  - available ONLY to students that attended the corresponding lecture class (based on a password)
  - during one week after the class
- final written examination in the summer session
  - $\boxed{Lect\_E \in [0,10]}$
  - closed-book exam
  - check *understanding* of fundamental concepts
  - three quiz tests
    - basic-level (63%): must take minimum 6 to pass and minimum 7 to have access to next
    - medium-level (20%): must take minimum 7 to have access to next
    - high-level (17%): essay subjects
- Lecture grade formula
  - $\boxed{Lecture = \frac{1}{6} Avg(Lect\_T_i) + \frac{5}{6} Lect\_E}$
- Conditions to pass
  - $Lect\_E >= 5$
  - $Lecture >= 5$

1.19

Final Grade

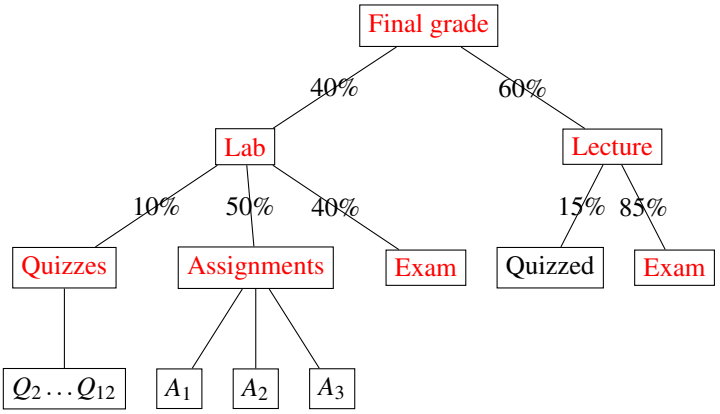
Final grade formula
Final_Grade $= 0,4 * Lab + 0.6 * Lecture$

1.20

Evaluation and Grading Illustration

The red items should be ≥ 5 in order to pass.

1.21



Optional: OS Club

- optional meetings for OS-related subject presentations
  - presentations and hands-on exercises
- OS Club channel on the OS course on Teams
- open to anyone
- every two weeks
- on Monday, from 18:00-20:00
- first meeting on Monday, March 1st 2021
- proposed subjects
  - Python scripting
  - Linux Shell (Bash) scripting
  - debugging with GDB
  - Windows API for OS services
  - remote communication using sockets
  - basic C++ resource management
  - OS-related vulnerabilities

1.22

Part II
OS Definition. Hardware Aspects Review. OS Structure

1.23

Purpose and Contents

The purpose of this part

- Define the OS and some of its basic concepts
- Review few fundamental aspects of computer hardware
- Discuss about two possible OS structures: monolith and micro-kernel

1.24

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 1, pg. 1 – 34
- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 1, pg. 34 – 62

1.25

Contents

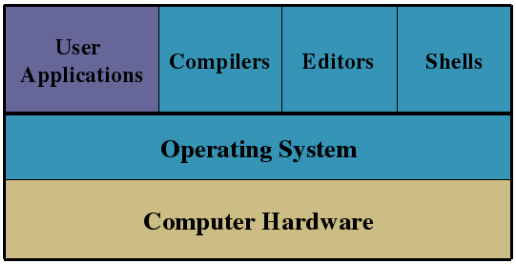


Figure 1: Computing System Structure

1.26

4 OS Definition

What Is An OS?

- A **software**, i.e. a program, an application
  - though, it is a **system** software
  - special from some perspective
- A collection of functions that
  - manage the hardware resources
  - provide the users the environment in which they can
    - run their own applications, in order to
    - use the system resources

1.27

Where Is Placed The OS?

- *Logical perspective*: **between** user applications and hardware ⇒ it is an **inter-mediator**
- *Physical perspective*: in the system's memory (RAM)
  - a (system) program **among other** (user) programs

1.28

Roles of An OS

- **Provider** of the *virtual or extended machine* interface
  - Hides the complexity of using the hardware devices
  - Provides **abstractions**: a more convenient view of the system
  - Purpose: **Convenience**
  - *The external perspective*: the user point of view
  - Example: files and directory for HDD, applications for processors
- **Manager** of the *hardware resources*
  - Brings the hardware resources in a functional state
  - Provides each program with time and space for using resources
  - Purpose: **Efficiency**
  - *The internal perspective*: the designer point of view
  - Example: multiplex one processor among more competing applications

1.29

OS's Interfaces

1. with hardware
   - the kernel and device drivers
2. **with user applications**
   - system calls (special functions)
3. with (human) user
   - shell, usually a user application

1.30

Do We Need An OS?

- **theoretically NO**
  - any application could be placed directly on the hardware
  - managing it in the way it wants
- yet, in practice **we NEED an OS as a helper**
  - we need help, as hardware interaction is difficult
  - we usually need someone specialized
  - we need to be efficient in terms of application development
  - we need to concentrate on our application logic
  - we need portability for our applications, as hardware is variable
- also **we NEED an OS as a mediator**
  - mediate between multiple applications running on the same computer
  - competing for the (same) limited resources
- $\Rightarrow$ OS is a **common body of software** for all applications
- $\Rightarrow$ OS is the **central mediation authority** for all applications

1.31

Desired OS Characteristics

- be **helpful** and **general**
  - provide a rich functionality
  - provide anything needed by any application
- provide **protection**!
  - protect the HW from user applications
  - protect itself from user applications
  - protect applications from one another
- be **invisible**
  - efficient: "no" performance overhead
  - light: "no" resource usage
  - flexible: "no" restrictions in terms of interface and abstractions
  - general: supporting perfectly any type of application

1.32

Though ... there is no "one-size-fits-all" OS!

1.33

Different Types of OSes

- Mainframe operating systems: *OS/390*, *z/OS*
- Server operating systems: *UNIX*, *Windows Server*, *Linux*
- PC (workstation) operating systems: *Windows 10*, *Mac OS*, *Linux*
- Real-time operating systems: VxWorks, QNX
- Mobile and embedded operating systems: *Android*, *PalmOS*, *Windows CE*, *Windows Mobile*, *Symbian*
- Virtualization operating systems (hypervisors): *VMware ESXi*, *MS Hyper-V*, *Xen*

1.34

A History of OSes

- the OS evolution was closely related to hardware evolution (computer generations)
- the more complex the hardware, the more complex and more responsible the OS
  - hides the increased complexity of hardware
  - uses it efficiently
- $\Rightarrow$ **an OS is closely tied to the hardware it runs on**

1.35

5 Computer Hardware Review

5.1 General View

Hardware Components

- CPU (mono- vs. multi-processor)
- Memory
- I/O Devices: *monitor*, *keyboard*, *storage devices* etc.
- BUS-es

1.36

Hardware Organization

1.37

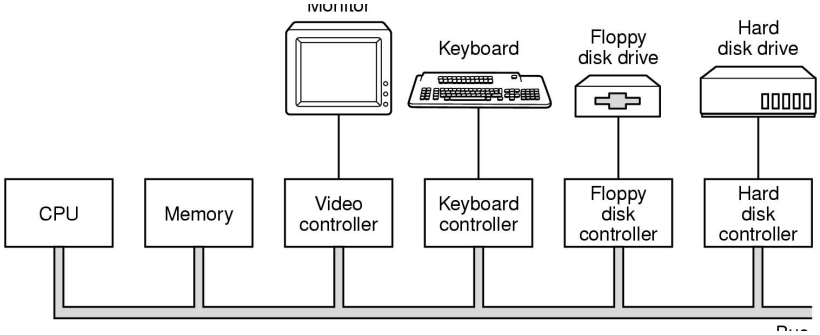


Figure 2: Computing Hardware

5.2 Main Hardware Components' Role and Characteristics

The Processor (CPU)

- Functionality: **executes programs**
  - fetch instructions from memory, decode and execute them
  - execute one instruction flow (i.e. program) at one moment
- Instruction set architecture (ISA)
  - has a specific set of instructions that can be executed
  - $\Rightarrow$ specific executable programs each processor supports and runs
- Registers
  - Program counter
  - Stack pointer
  - Many others – architecture dependent
  - Compose the *machine state* that is saved at *context switch*

1.38

CPU's "Intelligence"

- from a technical point of view, the CPU
  - is very complex
  - is very fast
  - does a good job executing programs
- from a logical (semantical) point of view, the CPU
  - is very "simple" and has no "intelligence"
  - provides support (and work), yet
  - does not understand what it executes
  - does not take (logical) decisions
  - does not manage the system
  - does not know what to run and when

1.39

OS – CPU Relationship (Questions to Answer)

- general question
  - how does the OS control and manage the system?
- specific question
  - how does the OS protect itself and the hardware from user applications?
    - how does the CPU avoid letting an application execute something that can harm the system?
  - when is the OS executed (by CPU)?
    - when does the CPU switch between user applications and OS?

1.40

Different Processor's Modes of Execution

- CPU could execute code in different privilege modes
- OS uses them for protecting itself and the hardware from user applications!
- **Privileged Mode**
  1. access allowed to the complete set of instructions
  2. the OS (kernel) runs in it → ***kernel* mode**
- **Non-Privileged Mode (Less-Privileged Modes)**
  1. restrict the possibility to execute some instructions from the instruction set
  2. does not allow direct access to hardware resources
  3. user applications are run in it → ***user* mode**

1.41

OS Usage of CPU's Execution Modes for Protection

1. when the system starts (boots) it runs in privileged mode
2. first software run is the OS, which "takes control of the system"
3. OS configures the CPU such that to trap into OS code at certain events (see below)
   - such configurations could only by done in privileged mode
4. when giving the processor to an application, the OS switches the CPU to user mode
   - there is no way to switch the CPU to privileged mode and still run the user application
5. CPU is automatically switched back to kernel mode when
   - the application generates exceptions, like illegal instruction, division by zero etc.
   - the application calls the OS (system calls), actually generating a sort of exception
   - hardware interrupts occur

1.42

When does the OS run? Switching From User Applications to OS (Part 1)

- **when the application calls explicitly the OS**
- through a special software mechanism
- called **system call**, which
- is a sort of program exception
- synchronous mechanism
  - triggered voluntarily by applications
  - the application waits for a result
- based on dedicated CPU instructions
  - examples on Intel: *int*, *sysenter / sysexit*, *syscall / sysret*

1.43

System Call Functionality

- the `read` system call takes about 11 steps

1.44

Memory

- functionality: **store programs** that are run by CPU
- should be extremely fast, large and cheap
- hierarchy of layers
  - registers: fastest, no delay, but limited size
  - cache
  - **main memory**: **RAM** (Random Access Memory)
  - external storage: HDDs, SSDs, magnetic tapes
    - few orders of magnitude cheaper and larger then RAM, but many orders of magnitude slower

1.45

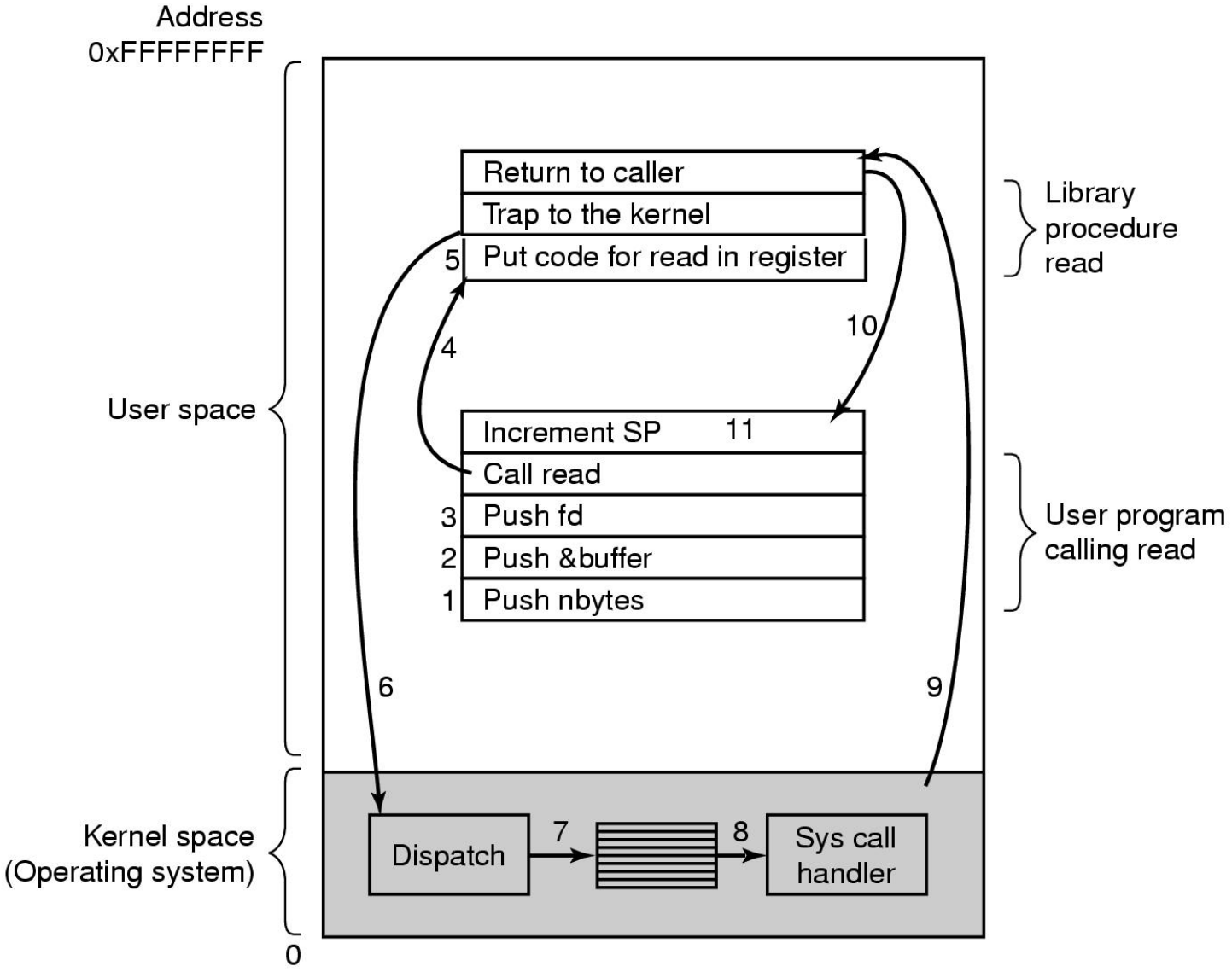


Figure 3: Taken from A. Tanenebaum, Modern OS, p. 46

I/O Devices

- components
  - controller and the device itself
- controller
  - directly controls the physical device
  - receives commands from the OS
- **device driver**
  - (normally) supplied by controller manufacturer
  - inserted into the OS ⇒ **part of the OS** ⇒ runs in kernel mode
- possible way to work
  - busy waiting
  - **interrupts**

1.46

When does the OS run? Switching to OS When Interrupts Occur (Part 2)

- **when external interrupts occur**
- it is a hardware mechanism → generated by hardware devices
  - e.g. timer interrupt, disk interrupts
- **asynchronous** mechanism
  - not triggered by applications
  - occurs at unpredictable moments
  - transparent for the applications
- the CPU is configured to switch to predefined memory locations
- where OS code is placed
- CPU automatically switches to privileged mode
- OS handles the interrupt, i.e. the hardware operations
- when finished, returns to the user interrupted application

1.47

Classes of Interrupts (based on Stallings, *Operating Systems*, p. 34)

Program Exceptions Occur as a result of an instruction execution, such as arithmetic overflow, division by zero, illegal instruction, illegal memory address.

Timer Interrupt Generated by a timer within the processor. This allows the OS to perform certain functions on a regular basis.

I/O Interrupt Generated by an I/O controller, to signal normal completion of an operation or various errors.

Hardware Failures Generated by a failure, such as power failure, memory parity error.

1.48

Do Not Forget!

OS does not run all the time! It is an event-triggered software.

1.49

6 Conclusions

Conclusions

- defined the OS
- identified the OS' roles
  - hide hardware complexity (provide a virtual machine)
  - manage hardware
- reviewed main hardware components: CPU, memory (RAM), I/O devices
- CPU's modes of execution: privileged (kernel) vs. non-privileged (user)
- CPU switch between OS and user applications
  - system calls

1.50

Lessons Learned

- OS is dependent on the hardware it runs on
- OS cooperates with the hardware (use hardware support) to provide its functionality
- there is no "one-size-fits-all" OS

1.51

Optional Sections

The following sections contain optional subjects, related to the presented ones! They are optional, though recommended for reading.

1.52

7 Main OS Concepts and Terms (Optional)

User

- the human being using a computing system
- *multiuser* system
  1. authentication mechanism: identify the user
  2. protection mechanisms: e.g. permission rights
  3. accounting mechanisms
- user groups
- *administrator*

1.53

Processes (User Applications)

- Definition
  - a program in execution
  - consists of executable code, data, stack, CPU registers value, and other information
- Processes can be created by
  - the OS: special system processes
  - other processes ⇒ process-child relationship and process hierarchy
- *Multiprogramming* and *timesharing* ⇒ *scheduling*
- Process *synchronization*
- Inter-Process *Communication* (IPC)

1.54

Files and Directories

- *file*
  - user's basic unit of data allocation
  - a collection of related data
- *directory* (*folder*)
  - used for file organization
  - results in a file hierarchy (tree, graph)
- permanent (physical) data space is viewed as a *file system*

1.55

Address Space

- the set of addresses accessible to a process
- virtual vs. physical space
- memory management techniques: segmentation, pagination, swapping

1.56

System Calls

- OS's services (interface)
- involve a special trapping mechanism to "jump" from user space (application's code) to kernel space (OS's code)
- available through normal library functions
- relationship between system call set and API
  - e.g. POSIX is an API, not a system call specification
- Linux examples: `open`, `read`, `fork`, `wait`, `exit`
- Windows examples: `CreateFile`, `ReadFile`, `CreateProcess`, `WaitForSingleObject`

1.57

8 Most Common OS Structures (Optional)

8.1 Monolithic OS Structure

Monolithic OS Structure. Description

- all OS parts linked together in a single piece of code
- the OS code is obtained by
  - compiling all the source files and then
  - linking all the object files in a single final executable
- OS is a collection of functions, each one being
  - visible to all the other ones (no information hiding)
  - able to call any other one, if necessary
  - able to access any data, if necessary
- apparently there is "no structure"
  - though, there is a logical one
- entire OS (all its components and code) runs in kernel mode
- examples: most *UNIX* versions, *Linux*, MS Windows

1.58

Monolithic OS Structure. Advantages vs Disadvantages

- advantages
  - performance $\rightarrow$ very fast
- disadvantages
  - non-modular: could be very big
  - non-modular: could be difficult to extend
  - non-portable: difficult to port on other systems
  - unreliability: a bug in a module can freeze / crash the entire OS
  - unreliability: a vulnerability in a module can compromise the entire OS

1.59

Monolithic OS Structure. Illustration

1.60

Monolithic Structure of Linux

1.61

Monolithic Structure of Windows

1.62

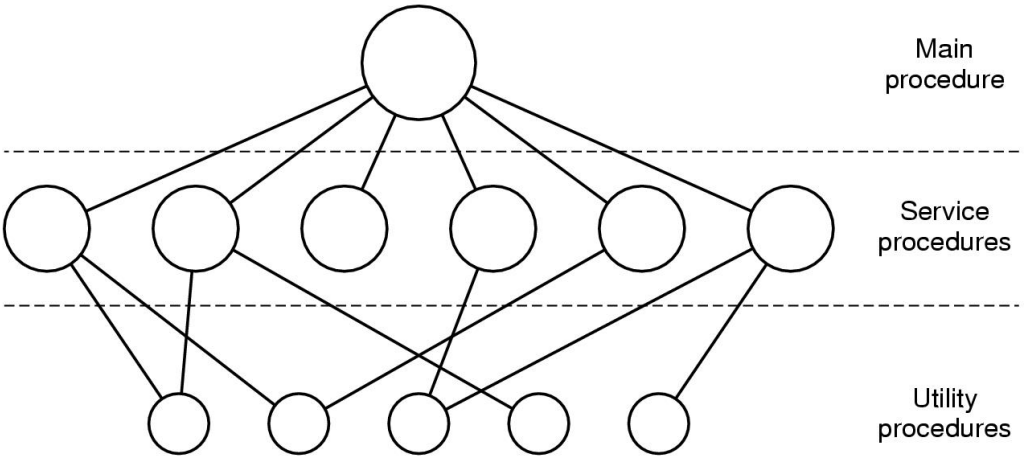


Figure 4: from Tanenbaum, Modern OS, p. 57

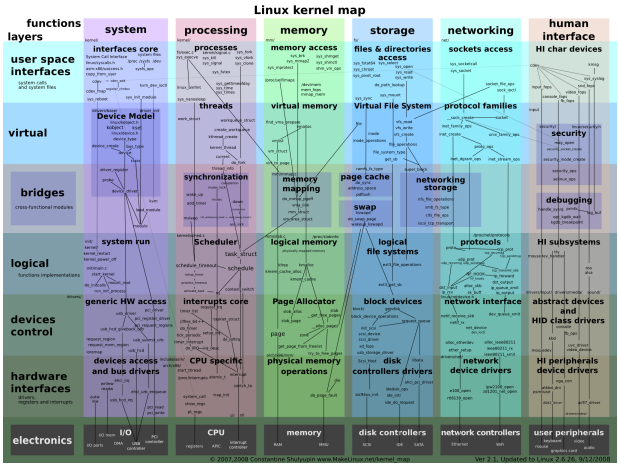


Figure 5: from `http://www.makelinux.net/kernel_map`

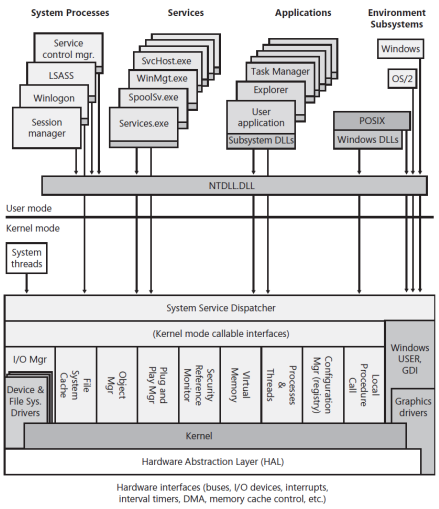


Figure 6: from `MS Windows Internals`

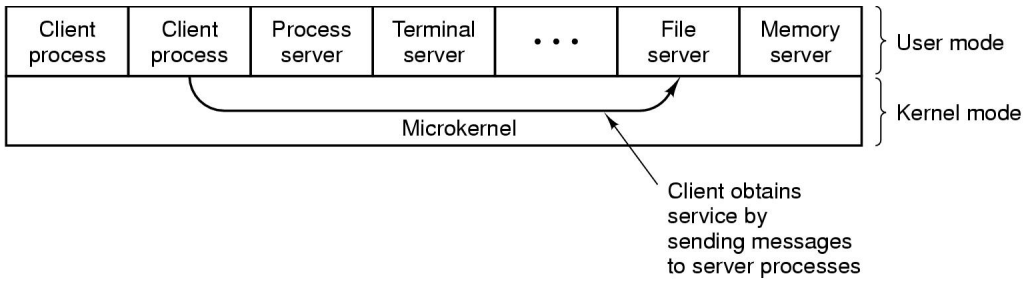


Figure 7: from Tanenbaum, Modern OS, p. 62

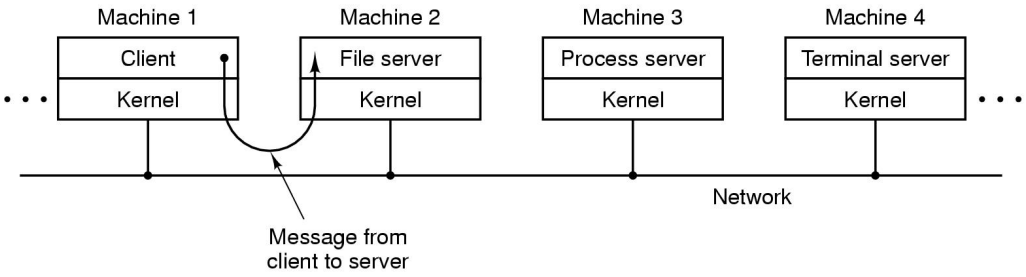


Figure 8: from Tanenbaum, Modern OS, p. 63

8.2 Microkernel OS Structure

Microkernel OS Structure. Description

- based on
  - modularization
  - clear and strict separation of duties and privileges
  - client / server architecture
- a small OS part (the kernel) runs in kernel mode
  - provides basic I/O operation, synchronization and communication between other OS modules
- other OS parts (system servers) run in user space
- examples: *Mach*, *GNU Hurd*, *Minix*, *L3/L4 family*, *seL4*

1.63

Microkernel OS Structure. Advantages vs. Disadvantages

- advantages
  - modularity
  - reliability, protection
  - portability
  - adaptability to distributed environments
- disadvantages
  - performance penalties due to inter-modules communications

1.64

Microkernel OS Structure. Illustration

1.65

Distributed OS Structure

1.66

Modules in Linux

- modules can be added (loaded) to and removed (unloaded) from kernel at run time
- takes advantages of micro-kernel architecture (modularity, efficient memory utilization etc.)
- Linux is still a monolithic OS

1.67

9 OS's Components (Optional)

Main OS's Components

- Process manager
- Memory manager
- Data manager (File system)
- I/O devices manager
- Communication system
- Protection system
- Shell

1.68

Process Manager

- manages user applications (processes and threads): creates, suspends, blocks, terminates, schedules processes
- POSIX system calls: `fork`, `execv`, `wait`, `exit`, `getpid` etc.

1.69

Memory Manager

- manage the system memory, allocate memory to processes, provide virtual memory
- system calls: `mmap`, `munmap`, `shmget`, `shmat`, `shmctl` etc.

1.70

Data Manager

- storage manager: physical data allocation and retrieval
- data presentation (visualization and access) manager: the file system
- POSIX system calls: `creat`, `open`, `lseek`, `read`, `write`, `close`, `link`, `unlink`, `stat`, `ioctl`, `fcntl` etc.

1.71

Shell

- it is a special user applications
  - do not belong to SO; runs in user space
  - each OS has its own shell
- provides the user the interface to interact with the OS
  - use the system
  - launch other applications
- two types of shells
  - text interface – *command interpreter*
  - graphical interface

1.72

Chapter 2

OS Shell: Command Interpreter

Functionality and Command Line Details

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Colesa

March 3rd, 2021

2.1

Purpose and Contents

The purpose of today's lecture

- Presents the general functionality of the command interpreter
- Presents some Linux command line details

2.2

Bibliography

- Lab text about Linux command interpreter
- Linux manual page of `bash` shell

2.3

Contents

2.4

1 General Description

Definition and Role

- the OS shell is a special user application
  - does not belong (entirely) to SO
  - runs in user space

  - each OS has its own shell
  - some OSes could have more shells
- provides the user the interface to interact with the OS
  - use the system
  - launch other applications
- two types of shell
  - text interface – *command interpreter*
  - graphical interface

2.5

Functionality: Description

- displays a command prompt (indicating the command line)
- reads from command line the user's keyboard input
  - command line = a string of characters
  - command line = a string of space separated words (!)
  - command line = a command with its parameters
- executes the command
  - *internal commands* $\rightarrow$ executed by the shell itself
  - *external commands* $\rightarrow$ searches for an executable file having the name of the command and runs it

2.6

Functionality: Pseudo-code

while FOREVER **do**
    displays a prompt
    reads a string from keyboard, i.e. the command line
    tokenize cmd. line $\Rightarrow$ command, its parameters, special chars
    if internal command **then**
        executes the internal command
    else
        searches for the corresponding executable file
        creates a new process to execute the external command
        if in synchronous mode **then**
            waits for the end of the child process
        end if
    end if
end while

2.7

Functionality: Illustration

2.8

Functionality: Execution Modes

- two execution / usage modes
  1. **interactive shell** (the one described above)
  2. **shell script command processor**
- shell script
  - a text file
  - a collection of shell commands (basically one per line)
  - accepts execution **parameters** (arguments)
  - could be **easily run multiple times**
  - could be **executed with different parameters**
  - helps **automatize** different actions
  - helps executing actions **non-interactively**

2.9

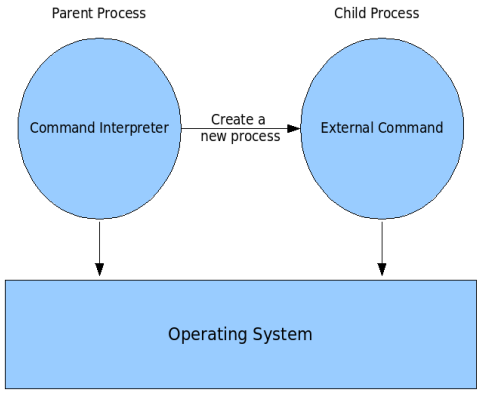


Figure 1: Shell Functionality. External commands are executed by different (child) processes.

Command Line. Simplified Form

```
$ cmd_name options parameters endings
```

- a **string of characters**
  - some of them are **special characters**
  - indicate the shell how to specially interpret the command line
- a space-separated **list of "words"**
  - more correctly **"items"**
  - an item could be a word or more words between quotes (`"word1 word2"`)
- the first word (item): the **command name**
- the other words (items): command **options** and **parameters**
- can end in special characters followed optionally by other words

2.10

Command Line. Complete Form

```
command_line := prompt command_list

prompt := '$' | '>' |
                  .... (any string of printable chars)

command_list := NULL | command |
                                  command cmd_separator command_list

command := cmd_name options parameters endings
cmd_name := WORD | FILE_PATH

options := NULL | '-'short_option [parameter] options |
           '-'options | '--'long_option=[parameter] options
short_option := LETTER
long_option := WORD

parameters = NULL | parameter parameters
parameter = WORD

cmd_separator := "||" | "&&" | '|' | ';'

endings := NULL | endings '&' | terminator FILE_PATH endigs
terminators := '>' | '>>' | '<' | '<<'
```

2.11

Command Types

- Internal (builtin) Commands
  - implemented and handled by the command interpreter
  - examples: `cd`, `read`, `alias`
  - very limited
    - specific to the shell (i.e. the current process)
    - affecting the environment and internal state of the shell
- External Commands
  - correspond to a file name
    - an executable file
    - a script (text file with commands)
  - examples: `/etc/init.d/apache2`, `/bin/ls`, `/usr/bin/passwd` etc.

2.12

Command Names

- a path
  - `/bin/ls`
  - `./my_ls`
- a name (word)
  - `ls`
  - `passwd`

2.13

Searching For an Executable File

- when command name not a path, but just a name
- search it in directories specified in the `PATH` environment variable
  - run "`echo $PATH`" to see `PATH`'s contents
  - example: `/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`
- order is important !
  - choose the first found executable with the searched name
  - run "`which cmd_name`" to see where it is found

2.14

Synchronous vs. Asynchronous Mode of Execution

- **synchronous** mode
  - the default mode
  - command interpreter (parent process) **waits** for termination of the currently running command (its child process)
  - only after that displays the prompt and gets the next cmd line
- **asynchronous** mode
  - activated by specifying '&' char at the end of the cmd line
  - command interpreter (parent process) **does not wait** for termination of the currently running command (its child process)
  - runs simultaneously with its child
    - displays the prompt and get next cmd line
  - the terminal is shared by command interpreter and its child
    - make sense to execute commands asynchronously when they do not display messages on the terminal

2.15

2 Command Line Parameters

Access Command Line Parameters in Shell Scripts

- using special variables
  - `$0`: name of script file (command name)
  - `$1, $2, ..., ${10}, ...`: parameters
- other variables related to command line parameters
  - `$#`: number of parameters in command line
  - `$@`: the string of cmd parameters
- examples

```
echo "Gets cmd line args one-by-one."
echo "Works for args with spaces."
for i
do
  echo $i
done
```

```
$> ./script.sh arg1 arg2
arg1
arg2
$> ./script.sh "arg 1" "arg 2"
arg 1
arg 2
```

```
echo "Gets cmd line args one-by-one."
echo "Doesn't work for args with spaces."
for i in $@
do
  echo $i
done
```

```
$> ./script.sh arg1 arg2
arg1
arg2
$> ./script.sh "arg 1" arg2
arg
1
arg2
```


Access Command Line Parameters in C Programs

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("The prg name: %s\n", argv[0]);

  for (i=1; i<argc; i++)
    printf("The i-th param: %s\n", argv[i]);
}
```

- `argc`: number of items in the command line
- `argv[0]`: command name (first item in command line)
- `argv[1]`: first parameter (second item in command line)
- ...
- `argv[argc-1]`: last parameter (last item in command line)

3 Command's Environment

Description of Application Environment

- **a list of "`name - value`" pairs** related to an application
  - simple association of two strings
  - stored in the **application's memory**
  - could **influence its behavior**
- too see them run one of the commands: "`set`", "`env`"
- an application **inherits from its parent** (i.e. shell) its environment
- **setting** a variable's value (**adding** it to the environment)

  - `VAR=`
  - `VAR=VALUE`
- **setting the inheritable attribute** of the variable
  - `export VAR`
  - `declare -x VAR`
- **setting the non-inheritable attribute** of the variable
  - `export -n VAR`
  - `declare +x VAR`
- **removing** a variable from the environment
  - `unset VAR`

2.18

Access Environment Variables in Shell Scripts

- simply specifying their names, preceded by '$'
- examples
  - `echo $PATH`
  - `echo $USER`
  - `echo $HOME`
  - `echo $PWD`

2.19

Access Environment Variables in C Programs

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

int main ()
{
    char * pPath;
    pPath = getenv ("PATH");

    if (pPath != NULL)
      printf ("The current path is: %s\n",pPath);
}
```

```
#include <stdio.h>

main (int argc, char** argv, char** env)
{
    int i;
    printf("The environment variables of the %s process are:\n", argv[0]);

    for (i=0; env[i]; i++)
      printf("env[%d]: %s\n", i, env[i]);
}
```

2.20

Security Considerations

- an **application should not trust its environment**
  - an inherited environment variable is controlled by the application's user
  - this could be exploited by a malicious user (i.e. attacker)
- especially if the application runs with high privileges
- example
  1. attacker writes a malicious version of a system executable, e.g. "`ls`"
  2. places the malicious program in a writable directory, e.g. "`/tmp/`"
  3. changes the `PATH` variable to include the attacker's directory `export PATH=/tmp:$PATH`
  4. executes the privileged application, which unintentionally launches the malicious executable
- recommendations: **do not trust the user!**
  - check the value of the inherited environment variables
  - establish safe values for them
  - do not add "." (i.e. current directory) to `PATH`

2.21

Secure Code Setting a Trusted PATH

```
#!/bin/bash

export PATH="/bin:/sbin:/usr/bin:/usr/sbin"

# ...
```

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

int main ()
{
    setenv("PATH", "/bin:/sbin:/usr/bin:/usr/sbin", 1);

    // ...
}
```


4 Standard Input and Output Redirection

Standard Inputs and Outputs

- each application (process) is associated a terminal used to
  - get inputs from keyboard
  - display characters on the screen
- each application has three (file) descriptors associated with its terminal
  - **0 for STDIN** (the *keyboard*, by default)
  - **1 for STDOUT** (the *screen*, by default)
  - **2 for STDERR** (the *screen*, by default)

Standard Input Redirection

- redirects the STDIN of a command to a existing file
  - what normally comes from keyboard taken from an existing file
- makes sense only for commands that reads something from STDIN
  - e.g. a C program that calls the `scanf` function
  - which results in a "`read(0, ...);`" system call
- examples

```
read var1 var2 < file_name
```

```
while read line
do
    echo $line
done < file_name
```

```
cat < file_name
```

```
sort 0<file_name
```


Standard Output Redirection

- redirects the STDOUT of a command to a file
  - what normally goes on the screen written in a file
- makes sense only for commands that sends something to STDOUT
  - e.g. a C program that calls the `printf` function
  - which results in a "`write(1, ...);`" system call
- examples

```
ls > file_name
```

```
cat file1 > file2
```

```
cat < file1 > file2
```

```
ls 1>file_name
```

```
sudo sh -c "cd /; ls > file_name"
```

Standard Error Redirection

- redirects the STDERR of a command to a file
  - what normally goes on the screen written in a file
- makes sense only for commands that send something to STDERR
  - e.g. a C program that calls the `perror` function
  - which results in a "`write(2, ...);`" system call
- examples

```
ls -R / > result 2>err_file
```

```
ls -R / 1>/dev/pts/1 2>/dev/pts/2
```


5 Special Aspects

Pipelining Commands

- redirects the STDOUT of a command to the STDIN of another command
- makes sense only for pairs of commands where
  - the first command displays something on STDOUT
  - the second command reads something from STDIN
- the linking between the two commands is made using a special communication file, named *pipe*
- Examples

```
ls -R / | less
```

```
cat file | sort | less
```

```
dpkg -l | grep "string" | less
```


Getting "FS Elements" From The Current Directories

```
for elem in *
do
    echo $elem
done
```

- the code above is equivalent with executing the command "`ls`"

Getting "FS Elements" From A Specified Directory

```
for elem in /home/os/*
do
    echo $elem
done
```

- the code above is equivalent with executing the command "`ls /home/os`"

Identifying Different Types of "FS Elements"

```
for elem in *
do
    if test -f $elem
    then echo File: $elem
    else
        if test -d $elem
        then echo Dir: $elem
        else
            if test -L $elem
            then echo Sym link: $elem
            else echo Other type: $elem
            fi
        fi
    fi
done
```

Dealing With Names Containing Spaces

- it is possible to have file names containing spaces
- for example: `echo something > "a file name"`
- specify them in command line like this
  - `ls a\ file\ name`
  - `ls "a file name"`
  - `ls 'a file name'`

```
for elem in *
    if test -f "$elem"
        rm "$elem"
    fi
done
```

2.31

6 Conclusions

Conclusions

- defined an OS shell
  - usually a user application
  - provides the user the interface with the OS
- types: graphical vs. text interface
- command interpreter
  - executed commands in child processes
  - functionality: synchronous vs. asynchronous
  - command line structure and syntax
- application environment (`PATH`, `HOME` etc.)
  - security aspects regarding untrusted environment
- special command line constructions
  - STDIN / STDOUT / STDERR redirection
  - pipelining

2.32

Lessons Learned

- never trust the user-controlled environment of an application!
  - check for environment variables' values
  - define safe values
- never use current directory "." in PATH environment variable!

2.33

7 Security Considerations (Optional)

PATH Attack on Shell Scripts

- the vulnerable script "`vuln-script.sh`"

```
#!/bin/bash
ls
```

- making the shell script having high (root's) privileges

```
$> sudo chown 0:0 vuln-script.sh    # change owner to "root"
$> sudo chmod +x vuln-script.sh     # make the script executable
$> sudo chmod +s vuln-script.sh     # make the script SUID
```

- the attacker's steps

```
$> cd /tmp
$> echo "cat /etc/shadow" > ls
$> export PATH=.:$PATH
$> vuln-script.sh
... displays /etc/shadow ...
```

- actually the attack does not work on current Linux
  - SUID bit for scripts is ignored
  - ⇒ script run without root's privileges

2.34

PATH Attack on Executables

- the vulnerable C program "`vuln-prg.c`"

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
        // display the program's effective and real UID
        printf("euid = %d ruid = %d\n", geteuid(), getuid());

        // load and execuute code in "ls" executable
        // "ls" is searched in the PATH's directories
        execlp("ls", "ls", NULL);
}
```

- making the vulnerable executable having high (root's) privileges

```
$> gcc vuln-prg.c -o vuln-prg   # compile de C program to get the exe
$> sudo chown 0:0 vuln-prg      # change owner to "root"
$> sudo chmod +x vuln-prg       # make the script executable
$> sudo chmod +s vuln-prg       # make the script SUID
```

- the attacker's code

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char **argv)
{
        int fd;
        char c;

        // open the "/etc/shadow", which is normally readable only by "root"
        fd = open("/etc/shadow", O_RDONLY);
        if (fd < 0) {
                perror("Cannot open file");
                exit(1);
        }

        // displays file's contents
        while (read(fd, &c, 1) > 0)
                printf("%c", c);
}
```

- the attacker's steps

```
$> cd /tmp
$> echo "cat /etc/shadow" > ls
$> export PATH=.:$PATH
$> vuln-prg
... displays /etc/shadow ...
```


8 Special Aspects (Optional)

Getting Hidden "FS Elements"

```
for elem in "/home/os/*" "/home/os/.*"
do
    echo $elem
done
```

- the code above is equivalent with executing the command

```
ls -a /home/os
```

```
for path in /home/os/* /home/os/.*
do
    file_name=`basename $path`
    if test $file_name = "."
    then
        echo Take care of "." element (crt. dir.)
        echo It introduces cycles in file tree
    elif test $file_name = ".."
        then
            echo Take care of ".." element (parent dir)
            echo It introduces cycles in file tree
        else
            echo Do something with $file_name
    fi
done
```

- the code avoids two special hidden elements
  - "." (current directory)
  - ".." (parent directory)

Getting Filtered "FS Elements"

```
for elem in "/home/os/lab*.c" ".*.sh"
do
    echo $elem
done
```

- the code above is equivalent with command

```
ls /home/os/lab*.c .*.sh
```

2.37

Returning An Exit Status

- Specify exit status: `exit n`
  - `0`: succes exit status
  - `n`: error exit status
- Getting the exit status
  - `$?` - the exit status of last executed command
  - use the command in a conditional command, like `if`

```
if command;
then echo Success;
else echo Error;
fi
```

2.38

Returning One or More Results

- specify results displaying them on screen like: `echo result1 result2`
- example: "`sum_dif.sh`"

```
sum=`expr $1 + $2`    # could be written in Bash "((sum = $1 + $2))"
dif=`expr $1 - $2`    # could be written in Bash "((dif = $1 - $2))"
echo $sum $dif
```

- Getting the results

```
results=`sum_dif.sh 3 5`
i=0
for result in $results
do
 if test $i -eq 0
 then
    echo Sum = $result
 elif test $i -eq 1
 then
    echo Dif = $result
 else
    echo Unexpected result: $result$
 fi
 i=`expr $i + 1`   # could be written in Bash "$((i++))"
done
```

2.39

Chapter 3

User Application Interaction With an OS

Dealing With C Programs in Linux

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Cole?a and Ciprian Opri?a

March 11th, 2020

3.1

Purpose and Contents

The purpose of today's lecture

- Review few basic aspects regarding writing and running C programs
  - their relationship with the OS

3.2

Contents

3.3

1 Getting Executable from Source Code

Compiler Role

- **CPU** does not understand "C language", not even "assembly"
  - **understands machine instructions** (its own language), i.e. bytes encoding certain actions
- **a program** (i.e. user application) that could be run should be
  - **a sequence of bytes** organized as **machine instructions**
  - machine instructions map 1:1 to assembly instructions

```
8b 10                   mov    edx,DWORD PTR [eax]
89 d0                   mov    eax,edx
c1 e0 02                shl    eax,0x2
01 d0                   add    eax,edx
c1 e0 02                shl    eax,0x2
01 c8                   add    eax,ecx
89 85 30 ff ff ff       mov    DWORD PTR [ebp-0xd0],eax
```

- the **app developer** "knows" a **higher-level language** (e.g. C)
- **compiler "translates" the source code** into machine instructions
  - ⇒ binary executable

3.4

Calling the Compiler

- **gcc (Linux, Windows, Mac)** `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows) `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- from an Integrated Development Environment (IDE)
  - interact with an interface, e.g. just click some button
  - ⇒ calls transparently commands like that above
- example
  - `gcc -Wall -Werror hellow.c -o hellow`
    - `-Wall` option means *warnings=all*, i.e. displays all warnings
    - `-Werror` option means *warnings=errors*, i.e. report warnings as errors

3.5

Never ignore compiler's warnings!

3.6

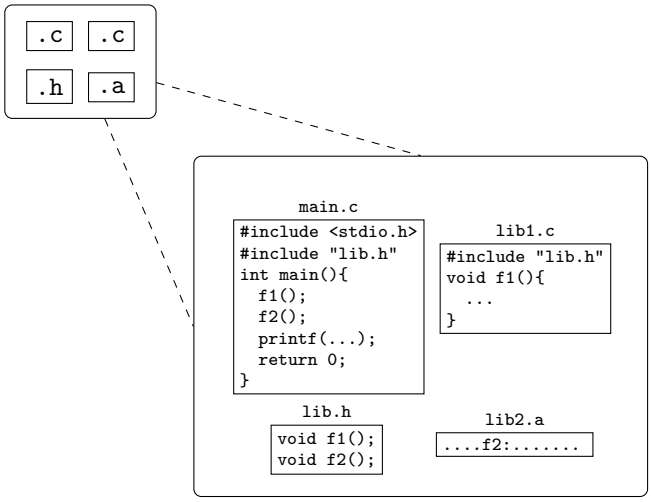
Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / **shared libraries** (`.so` files) or **static libraries** (`.a` files)

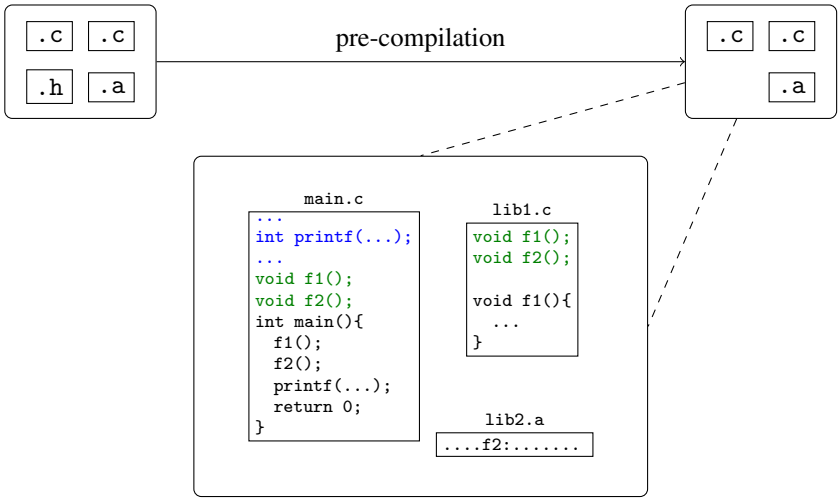
3.7

Compilation

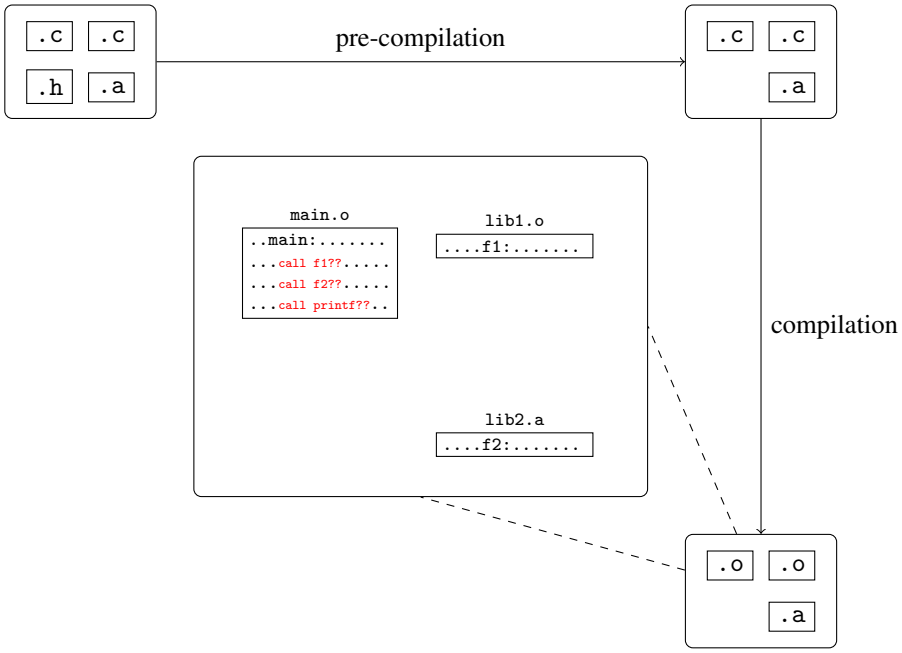
[1] Note the difference between definition and declaration!
[2] See gcc and Visual Studio recommendations


.c
.c
.h
.a
main.c
#include <stdio.h>
#include "lib.h"
int main(){
f1();
f2();
printf(...);
return 0;
}
lib1.c
#include "lib.h"
void f1(){
...
}
lib.h
void f1();
void f2();
lib2.a
....f2:.....

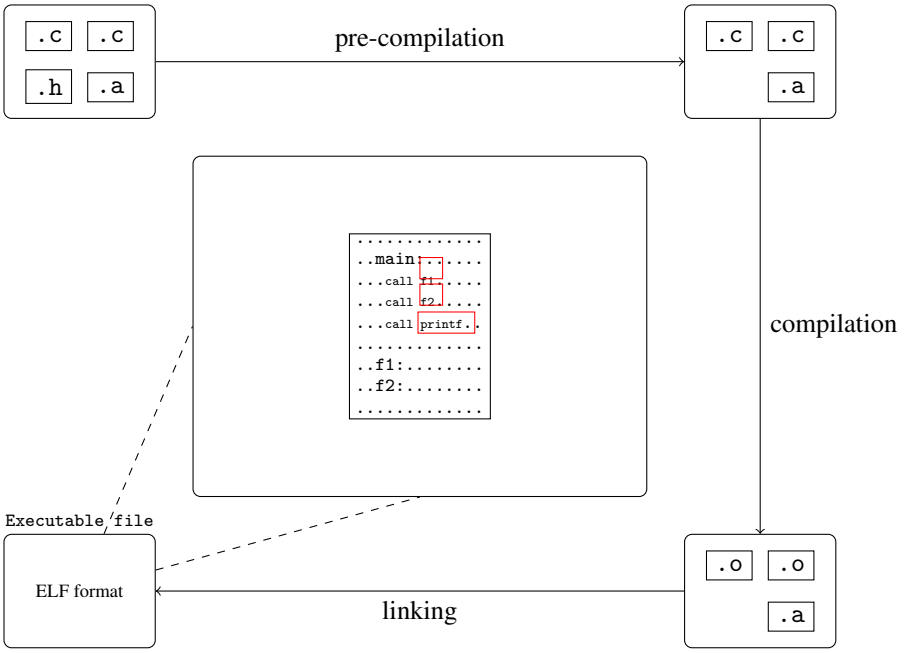
3.8


.c
.c
.h
.a
pre-compilation
.c
.c
.a
main.c
...
int printf(...);
...
void f1();
void f2();
int main(){
f1();
f2();
printf(...);
return 0;
}
lib1.c
void f1();
void f2();
void f1(){
...
}
lib2.a
....f2:.....

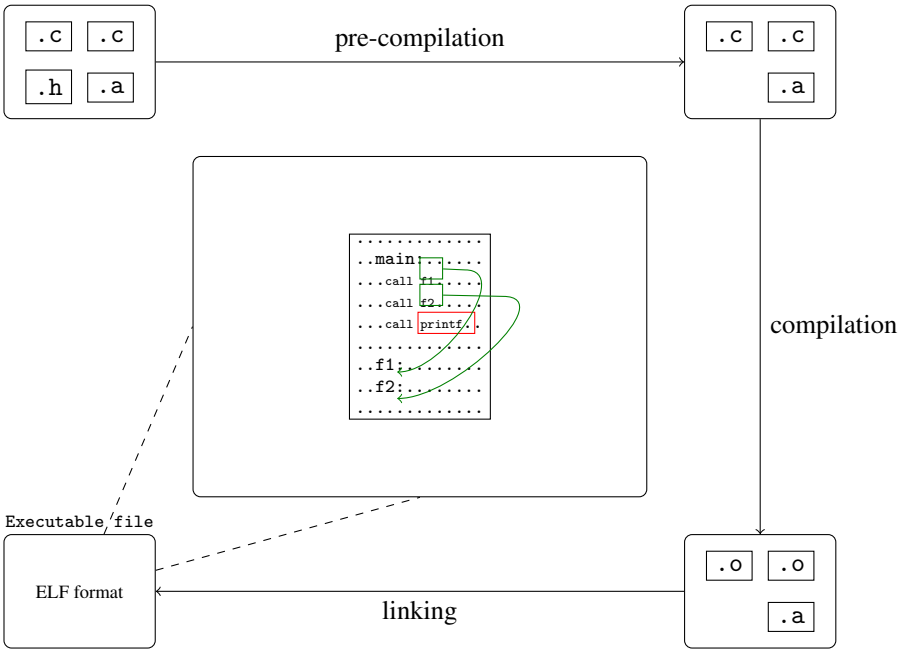
3.9


.c
.c
.h
.a
pre-compilation
.c
.c
.a
compilation
main.o
..main:.....
...call f1??.....
...call f2??.....
...call printf??..
lib1.o
....f1:.....
lib2.a
....f2:.....
.o
.o
.a

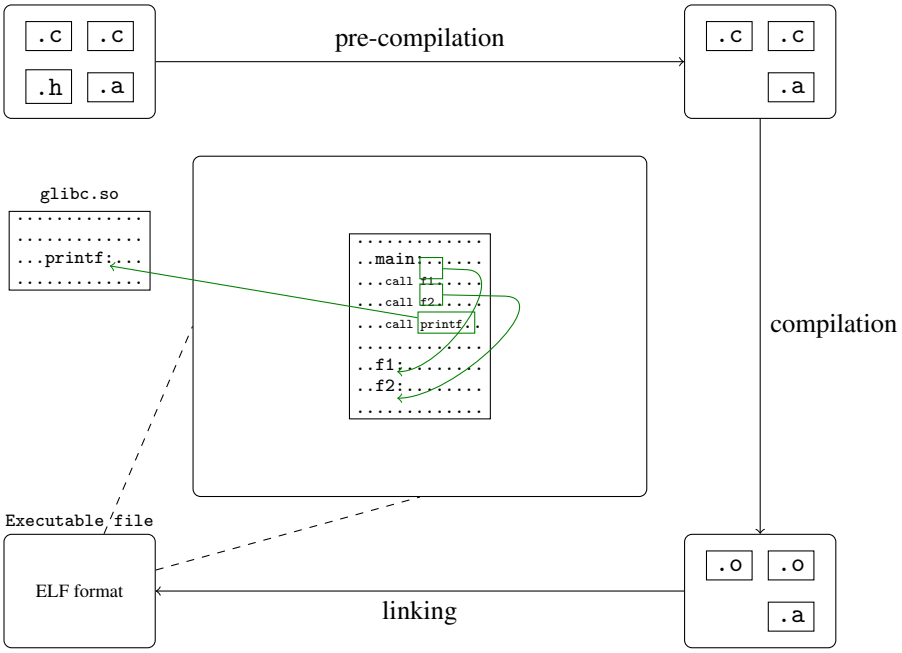
3.10


.c
.c
.h
.a
pre-compilation
.c
.c
.a
compilation
..main.....
...call f1.....
...call f2.....
...call printf..
..f1:.....
..f2:.....
Executable file
ELF format
linking
.o
.o
.a

3.11


.c
.c
.h
.a
pre-compilation
.c
.c
.a
compilation
..main.....
...call f1.....
...call f2.....
...call printf..
..f1:.....
..f2:.....
Executable file
ELF format
linking
.o
.o
.a

3.12



3.13

2 Running the Executable

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- ⇒ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- ⇒ CPU starts running the new application

3.14

Local variables and the stack (1)

test1.c:

```
#include <stdio.h>

int main(void){
    int x = 7;
    printf("x=%d\n", x);
    return 0;
}
```

Compiling: `gcc -Wall test1.c -o test1` Running: `./test1 x=7` Compiling with different

options: `gcc -g -m32 -Wall test1.c -o test1`

- `-g`: add debugging info in the executable
- `-m32`: generate 32-bit code

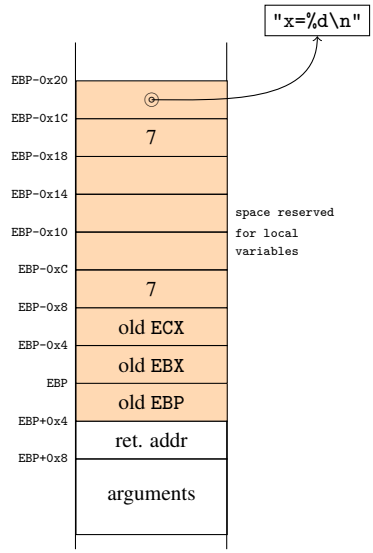
View executable as assembly code: `objdump -D test1 -M intel -S`

- `-D`: disassembly
- `-M intel`: Intel syntax
- `-S`: display source code also

3.15

Local variables and the stack (2)

```
;int main(void){
...
push ebp
mov  ebp,esp
...
push ebx
push ecx
sub  esp,0x10
...
;int x = 7;
mov DWORD PTR [ebp-0xc],0x7
;printf("x=%d\n", x);
...
push DWORD PTR [ebp-0xc]
lea  edx,[eax-0x19e8]
push edx ;"x= %d\n"
mov  ebx,eax
call 3b0 <printf@plt>
...
;return 0;
mov    eax,0x0
```



Array Manipulation (1)

test2.c:

```
int main(void){
    int a[2];
    int b[] = {7, 8};
    a[0] = 3;
    a[1] = 4;
    printf("%d %d\n", a[0], a[1]);
    printf("%d %d\n", b[0], b[1]);
    return 0;
}
```

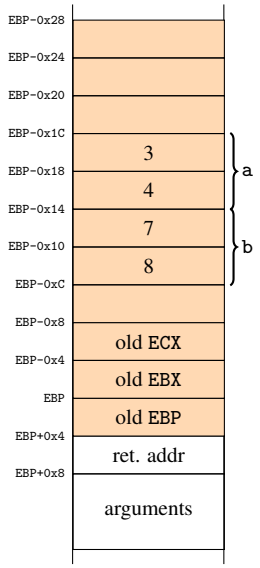
```
$ gcc -Wall test2.c -o test2 $ ./test2  3 4 7 8
```

- could be initialized when declared
- element indexing starts at 0
- when declared as local variables, they are allocated on the stack

Array Manipulation (2)

```
...
push ebp
mov  ebp,esp
push ebx
push ecx
sub  esp,0x20
...
;int a[2];
```

```
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
...
```



Memory Corruption (1)

test3.c:

```
int main(void){
    int a[2];
    int b[] = {7, 8};
    a[0] = 3;
    a[1] = 4;
    a[2] = 5;
    printf("%d %d\n", a[0], a[1]);
    printf("%d %d\n", b[0], b[1]);
    return 0;
}
```

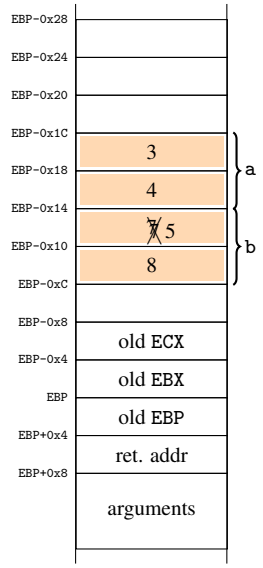
`$ gcc -Wall test3.c -o test3 $ ./test3 3 4 5 8` What happens when access an array out of its bounds?

- Java: an array is an object and all the operations on it are controlled, it knows its bounds
  $\Rightarrow$ throws an exception
- C: the array is just a memory address, where the array starts
- C standard does not define behavior when array accessed out of bounds
  - anything could happen

Memory Corruption (2)

```
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
;a[2] = 5;
mov DWORD PTR [ebp-0x14],0x5
...
```

Question
What happens if change `a[8]`?



3.20

Pointers (1)

test_ptr.c:

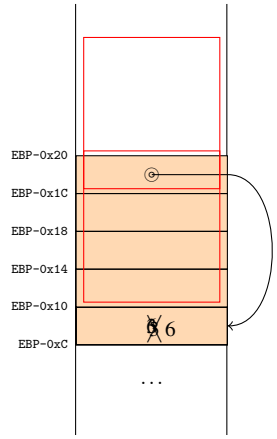
```
void f1(int x){
    x = x * 2;
}
void f2(int *x){
    *x = *x * 2;
}

int main(void){
    int x = 3;
    f1(x);
    printf("%d\n", x);
    f2(&x);
    printf("%d\n", x);
    return 0;
}
```

- a pointer is a variable that contains a memory address, e.g. the address of another variable
- memory addresses are just integers (on 32 or 64 bits)
- useful when need referring some data, not copying it (e.g. reference parameters of a function)
- & - reference (get the address of a variable)
- * - derenference (get memory contents from a memory address)

3.21

Pointers (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>
add esp,0x10
;printf("%d\n", x);
...
```



Arrays Are Pointers (1)

test_array_ptr.c:

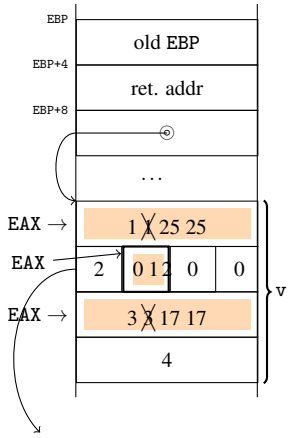
```
#include <stdio.h>
void f(int *v) {
    v[0] = 25; // *v = 25
    *(v + 2) = 17; // v[2] = 17
    *((char*)v + 5) = 1; // !!!
    3[v] = 44; // v[3] = 44
}
int main() {
    int v[] = {1, 2, 3, 4};
    int i, n = sizeof(v)/sizeof(v[0]);
    f(v);
    for(i=0; i<n; i++){
        printf("%d ", v[i]);
    }
    printf("\n");
    return 0;
}
```

- v variable (i.e. the array name)
  - is a pointer to the beginning of the array
  - points where the array starts
  - it is a pointer of the same type as the array elements
- adding N (an integer) to a pointer ⇒ add "`N*sizeof(ptr. type)`"

```
$ ./test_array_ptr
25 258 17 44
```


Arrays Are Pointers (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



$2 \cdot 256^0 + 1 \cdot 256^1 = 258$

Note: due to the **little-endian representation**

- the value of v[1], i.e. `0x00000002`, stored in memory as bytes `0x02 0x00 0x00 0x00` (from smaller to bigger addresses)
- ⇒ v[1] becomes `0x02 0x01 0x00 0x00`

Data Structures and Pointers

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Assigning values to structure fields

```
    struct MyStruct s;
    s.a = 7;
    s.b = 12;
    s.p.x = 150;
    s.p.y = -11;
    s.c[1] = 10;
    s.c[2] = 'a';
```

Initialization when declared

```
    struct MyStruct s = {
        .a=7, .b=12,
        .p={.x=150, .y=11},
        .c={0, 0, 0, 0, 0}
    };
```

Pointers to data structures:

```
    struct MyStruct s = {...};
    struct MyStruct *ps;
```

How do we access the a field?

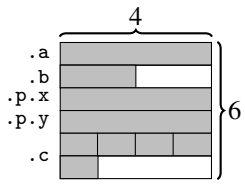
- with s: `s.a`
- with ps (v1): `(*ps).a`
- with ps (v2): `ps->a`

Recommended style: (->).
How do we access (sub)field x of field p?

- `s.p.x`
- `ps->p.x`

Data structure's size

```
    printf("Point size = %d\n",
        sizeof(struct Point));
    printf("MyStruct size = %d\n",
        sizeof(struct MyStruct));
```




```
Point size = 8
MyStruct size = 24
```

Each data structure field is aligned at `DWORD` (4 bytes).

Data Structure Alignment

How to force `BYTE`-alignment, if needed?

gcc (Linux)

```
struct __attribute__((packed)) MyStruct {
    ...
};
```

Visual Studio (Windows)

```
#pragma pack(push, 1)
struct MyStruct {
    ...
};
#pragma pack(pop)
```

Cast to data structures

```
unsigned char v[] = {
     7, 1, 0, 0, 12, 0, 0, 0,
    150, 0, 0, 0, 11, 0, 0, 0,
    'a', 0x62, 'c', 100, 0, 0, 0, 0
};
struct MyStruct *ps = (struct MyStruct*)v;
printf("a=%d b=%d p=(%d, %d), c='%s'\n",
    ps->a, ps->b, ps->p.x, ps->p.y, ps->c);
```

What will be displayed?

`a=263 b=12 p=(150, 11), c='abcd'`

3.25

Dynamic Memory Allocation (1)

Functions `malloc` and `free`

```
int *v = (int*)malloc(10000 * sizeof(int));
struct MyStruct *ps = (struct MyStruct*)
                      malloc(sizeof(struct MyStruct));
...
free(v);
free(ps);
```

- `malloc` allocates memory area on the *heap* and returns a pointer to the allocated memory
- `free` releases an allocated memory area having a pointer to it

3.26

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: $8MB$
  - default stack size on Windows: $1MB$
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

3.27

Dynamic Memory Allocation (3)

Memory leaks

- any dynamically allocated memory should be explicitly released with `free()`
  - otherwise it remains allocated until the program ends
- C language has **no garbage collector**
- sometimes memory is not release in the same function where it is allocated
  - *example*: a function allocates memory (for storing some results) and returns a pointer to that memory
  - the function getting such a pointer should release the memory, when not needing it anymore
- example of a classical memory leak (**don't do like this!**)

```
int x[1000];
int *p = (int*)malloc(1000 * sizeof(int));
p = x; // the only pointer to the dynamically allocated memory is lost
```

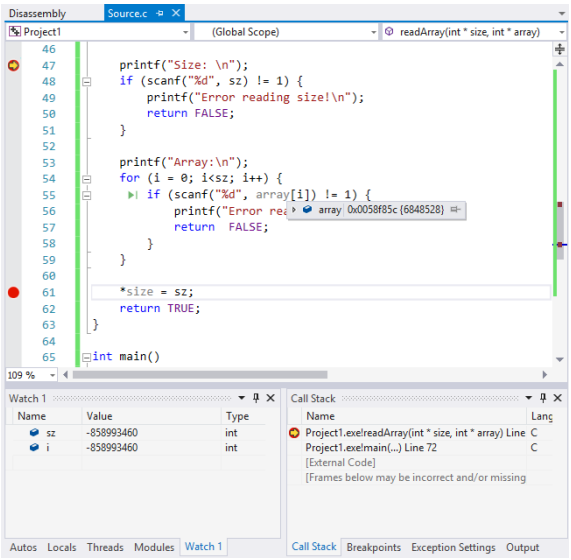

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use `strcpy()`!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - ⇒ do not use them when strings not terminated with NUL

3 C Programs Debugging

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the `-g` option of *gcc*
  - debugging info: variable names, correlation to the source code



Debugging with a GUI (1)

- very convenient, simple and efficient
- e.g. set breakpoints by clicking in front of line of source code
- e.g. view variable value with *mouse over* or right-click → *Add Watch*
- trace / debug a program
  - *step into*: execute and go forward one instruction entering into functions
  - *step over* execute and go forward one line (even if a function, so not entering in such a function)
  - *continue*: execute and go forward to the next breakpoint or program's end

3.31

Debugging with a GUI (2)

3.32

Debugging with GDB

Running as: `gdb <executable_file>`
Displays an interactive shell where we can type commands like:

- `break test.c:12`
- `run`
- `continue`
- `step` (step into)
- `next` (step over)
- `bt` (*backtrace*: shows the stack frames)
- `print myvar`: displays the variable's contents

3.33

Postmortem Debugging

- provides us the way to investigate (debug) a crashed program's state
- useful when our programs run on other remote systems
- steps
  - activate core dumping on the remote system `ulimit -c unlimited`
  - run the program until its crash ⇒ a `core` file
  - loads the core dump into the `gdb` debugger `gdb <executable_file> core`

3.34

Using *valgrind* to Detect Memory Leaks

- allows analyzing a program during its runtime
- for finding out memory leaks it intercepts `malloc` and `free` (and other related functions) and keeps evidence of the allocated memory areas
- at program termination displays a report with memory leaks

```
$ valgrind ./<executable\_file> [<arguments>]
...
==9347== LEAK SUMMARY:
==9347==    definitely lost: 55 bytes in 5 blocks
==9347==    indirectly lost: 0 bytes in 0 blocks
==9347==      possibly lost: 0 bytes in 0 blocks
==9347==    still reachable: 43 bytes in 3 blocks
==9347==         suppressed: 0 bytes in 0 blocks
```

- when there are leaks run it again for detailed info `-leak-check=full -show-leak-kinds=all`

3.35

4 Recommendations About Writing C Programs

There Are No Perfect Programs

- it is almost impossible to write a complex bug-free program
  - at least from first try
- every day new bugs are found in real-life "professional" (commercial) software run by millions of users
- there are metrics that account bugs per thousands of lines of code

A good programmer could be allowed to not write perfect programs, but (s)he is required to be able to test / debug them, identify bugs and fix them!

- testing and debugging a program has same importance like writing code
  - understanding a written code is not at all the same like writing it by yourself from scratch

3.36

Using Code From Other Sources

- **Do not copy-paste code you do not understand!**
- actually, **never copy-paste any code**, at least for your school projects!
- there is (almost) nothing to learn from others' code
- if really need to use someone else' code (in real life)
  - make sure you understand and control that code
  - mention the source of your code (at least in a comment)

3.37

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

3.38

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- ⇒ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them**!
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - ⇒ **dangling pointer**
  - **after releasing the memory a pointer points to, assign it NULL value!**

3.39

Watch the Global Variables

- global variable not explicitly assigned values, are initialized with 0
- global variables are visible from any function of your program
  - ⇒ if not a constant, anyone can change them
- global variables can be changed concurrently from multiple threads
  - ⇒ unexpected, unpredicted values
  - such a code is called thread-unsafe
- **do not use global variable, when not really needed!**
  - better and safer to give a function the needed context information as parameters

3.40

Do not use uninitialized local variables!
Avoid using global variables!
Do not use dangling pointers!

3.41

Warning Handling

- compiler warns us when something in the compiled program is confusing regarding the use of some variables or functions
  - a logical expression (e.g. a condition in an `if`) using a single '`=`' instead of two
  - code with no effect
  - usage of uninitialized variables
  - calling `printf()` with a strange or incorrect format string
- when we want to ignore some types of warnings, so not be reported about them, we can specify certain compiling options
  - be aware that too many irrelevant warnings, could hide from us the important ones
- for the release version of a program is recommended the `-Werror` option
  - just to report an error for any warning

3.42

5 Conclusions

What We Talked About!

- basic aspects related to C programs
- getting an executable from the source code
- local variables on the stack
- pointers, arrays, data structures
- debugging aspects
- coding recommendations

3.43

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

3.44

Chapter 4.1
The File System
The User Perspective

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

March 17th, 2021

4.1.1

Purpose and Contents

The purpose of today's lecture

- General Overview of The File System Module
- File Concept
- Directory Concept

4.1.2

Bibliography

- Andrew Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 6, pg. 380 – 398.

4.1.3

Contents

4.1.4

1 File System (FS) Overview

Context

- users need to **store** and **retrieve**
  - **persistent data**
  - **large amount of data**
- users need to **share data**
- ⇒ OS should
  - **provide services** for such needs
  - **manage** storage devices (area) and stored data

4.1.5

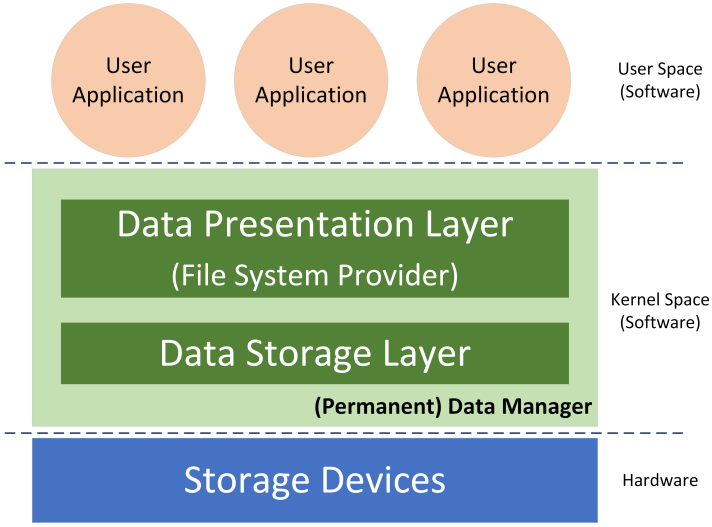


Figure 1: Data Manager's Layers. What we call "File System" is the way the "Data Manager" makes visible the data storage space to users.

Definition

- **an OS's component**
  - ⇒ part of OS
- the **interface** between the user and the physical storage devices
  - *provides* the users access to the storage area
  - *manages* the information on the storage area
- ⇒ the (permanent) **data manager** 4.1.6

Role

- **provides** the users a simplified, uniform and **abstract view of the storage area**
  - hides the complexity of physical storage devices
  - abstract concepts: *file* and *directory*
  - ⇒ *its name: File System*
- **manages** data on the storage area
  - interacts directly with storage devices
  - ⇒ contains a hardware dependent layer (code modules) 4.1.7

File System Architecture. General View 4.1.8

File System Architecture. Detailed View 4.1.9

2 Fundamental Concepts

2.1 File

Definition. User Perspective

- the basic **storage unit** of information
  - anything the user wants to store must be placed in a file
- provides a (convenient) way to
  - **store** the information on storage devices
  - **retrieve** the information back later
- **a container**, i.e. **a box**
  - a collection of related information defined by its creator 4.1.10

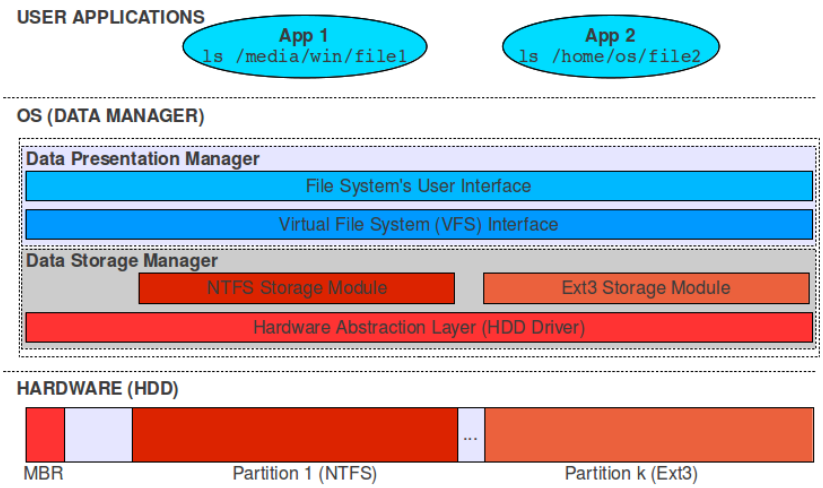
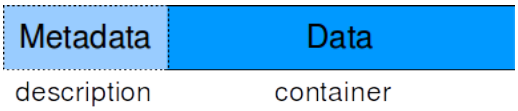


Figure 2: Data Manager's Components



The file is the basic abstract concept for storing user data!

4.1.11

Definition. OS (Internal) Perspective

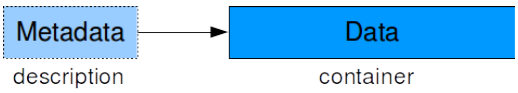
- an abstraction mechanism: a *container*
  - models the way data is provided to users
  - does not necessarily correspond to the way data is stored (on physical devices)
- OS must
  - **store** the container's contents on storage area
  - **retrieve** back later the container's contents
- the file maps the user view of data to the physical data
  - identifies on physical devices and links together physical data (i.e. chunks of bytes) belonging to a file

4.1.12

File's Components

- **data**: user's useful information placed in the container
- **metadata**: description associated to stored data
  - needed, maintained and sometimes imposed by OS
  - some fields also used by user
- the two components **can be stored in different parts** on a storage area
  - there is a link maintained from metadata to data
  - $\Rightarrow$ FS always **starts from a file's metadata** in order to access that file

4.1.13



File Metadata: Name

- the user's way to **identify** (refer to) a file
- **must be unique** (there are though some exceptions!)
- each file must have (at least) a name
- consists of a **string of characters**
  - upper vs. lower letters, char sets
  - its length is generally limited
- file name's **extension**
  - normally **not imposed by OS**
  - *rarely*: may be recognized and used by the OS
  - *usually*: **a user-space convention** to provide hints about file's contents
  - required by some application (e.g. *gcc* requires .c)
  - examples: `book.`*`docx`*, `letter.`*`odt`*, `setup.`*`exe`*, `arch.tar.`*`gz`*, `progr.c`, `image.`*`jpg`*

4.1.14

From the OS perspective every filename is just a string of (unrestricted) characters!

4.1.15

File Metadata: Type

- Regular files
  - Contain user data (text or binary)
- Directories
  - System files used to organize file space
- Links
  - System files used to redirect the access to other files
- Special files
  - Model I/O devices
  - *Character*: terminals, mouse, etc.
  - *Block*: disks
- Pipes
  - Inter-process communication (IPC) mechanisms

4.1.16

File Metadata: Structure

- **file's structure = the way the file's contents is organized and interpreted**
- possible strategies
  1. **no structure**: just a *sequence of bytes*
     - used by most OSes for normal files
  2. **specialized structures**: used normally for system files
     - sequence of (fixed-length) records
     - tree of records (e.g. B-Trees)

4.1.17

File Metadata: Type vs. Structure – Pros and Cons

- different **file types** ⇔ different **file structures**
- that is **used for system files** (directories, links, pipes etc.)
  - managed by OS transparently from the user
- **But . . .** should the OS support **types and structures for regular files**?
  - **if, yes**
    - +: **efficient** / particular file's contents manipulation
    - +: convenient for novice users (not much / complex code)
    - -: additional OS code
    - -: restrictions, rigidity
    - -: file not portable
  - **if, no**
    - +: **flexible** ⇒ convenient for advanced users
    - +: no additional OS code
    - -: no OS support, additional application code
    - ⇒ **each application must manage / interpret itself the contents of files it works with**

4.1.18

File Metadata: Type vs. Structure – Real Life OSes

- **no structure for regular files**
  - *text* vs. *binary* files: just a user convention
  - basically, **all files are binary**, i.e. each file's contents must be handled specifically by applications using it
  - ⇒ *text file* is just a particular file type
- ⇒ **flexibility** (*separate mechanism by policy*)
- **yet**, *every OS recognizes its own executable files*

4.1.19

From the OS perspective every file is just a sequence of bytes!

4.1.20

Question

Yet, from the user perspective . . . how would you classify the following files in terms of **text** vs **binary**, based on their filename extension?

1. `program.c`
2. `archive.zip`
3. `paper.pdf`
4. `index.html`
5. `persons.xml`
6. `app.java`
7. `cv.docx`

4.1.21

File Metadata: Attributes

- system attributes
  - type
  - length
  - owner
  - access permissions (rights): basically *read*, *write* and *execute*
  - time stamps (e.g. creation time, last access time, last modification)

  - addresses of allocated blocks for that file
    - named during that course *Block Addresses Table* (BAT)
    - the link between metadata and data
  - ...
- user attributes (if supported)
  - anything the user wants
  - examples: *source Web address* (for an html file), *place* (for an image file), *author* (for a document) etc.

4.1.22

File Access Method

- **sequential** access
  - **impose an order** on the way the bytes are accessed
  - could be imposed by the storage device (e.g. tapes)
  - could be imposed by the nature of the file's contents
    - e.g. **specific for directories**
- **random** access
  - accesses the bytes or records **out of order**
  - specific to storage devices like hard disks
  - normally, **specific to regular user files**

4.1.23

Operations On Files

- Access file data
  - *open*
  - **write**, i.e. *store* data
  - **read**, *get back* stored data
  - position (seek), i.e. *randomly* accessing stored data
  - *close*
- Manipulate files/Access file metadata
  - create
  - get/set attributes
  - rename
  - truncate
  - delete

4.1.24

2.2 Directory

Definition. User Perspective

- the way to **organize / classify files**
  - when too many, difficult to find them based just on their name
- a **collection / class** of related elements (files)
  - a file could be placed in a directory (or more?)
  - $\Rightarrow$ directory's name is a sort of additional (user) metadata associated to the file
- reduces the size of the filename space
  - files in different directories could have the same name
- imposes a structure / **hierarchy** on the file space
  - helps the user locating files easier in huge file spaces
  - **finding files visually** in the hierarchy is based on **navigation**
    - refine filters gradually, based on subdirectory names
  - does not help (too much) the **user applications locating a particular file**
    - if not knowing in advanced a file's complete path
    - $\Rightarrow$ **file must be searched** in the entire FS tree

4.1.25

The directory is the abstract concept for organizing user files!

4.1.26

Definition. OS (Internal) Perspective

- an abstraction mechanism of grouping files
- a **container**: data (bytes) that must be stored
  - a **special file managed by OS** transparently from user
  - i.e. directory's bytes are interpreted by the OS and user-applications are provided directory's elements
  - usually organized as specialized searching data structures, e.g. B-trees
- it also **consists of data and metadata**
  - very similar to a regular file
  - $\Rightarrow$ just a FS-element from the OS perspective

4.1.27

Directory Hierarchy

- types
  - single-level directory systems
  - two-level directory systems
  - **hierarchical** directory systems
    - the most general form: **tree** or **graph**
- **file paths**
  - the way of **identify a file in a hierarchy**
  - a list of consecutive nodes ending with the file name

4.1.28

File Paths

- **absolute paths**
  - starting node is the **root directory**
  - examples
    - `c:\Program Files\Application\run.exe` (Windows)
    - `/home/students/adam/program.s` (Linux)
  - usage: access files at known fixed places
- **relative paths**
  - not staring from the root directory
    - starting node is the **current directory**
  - each application has its own current directory associated to it
  - examples
    - `Application\run.exe` (Windows)
    - `students/adam/program.s` (Linux)
  - special notations (directories)
    - current directory: '`.`'
    - parent directory: "`..`"
  - usage: access files in a subtree with a known structure, but located to an unknown location in the overall FS tree

4.1.29

The directory concept imposes a hierarchy on the file space, requiring a path in order to identify a file!

4.1.30

Question

Supposing the current working directory of an application is "`/home/os`", which will be the corresponding absolute paths for the following relative paths?

1. `file_1`
2. `./file_2`
3. `project/file_3`
4. `../file_4`
5. `../../file_5`
6. `../../../../file_6`
7. `../../etc/file_7`
8. `./../../etc/apache2/./../../../home/os/./file_8`

4.1.31

Operations On Directories

- create
- delete
- rename
- opendir
- readdir
  - **sequential access**
  - positioning not allowed anywhere
- rewind
- "write" (**not directly allowed**)
  - *add* elements (create directories and files)
  - *delete* elements (delete directories and files)
  - *rename* elements (change files' and subdirs' names)
- close

4.1.32

3 Conclusions

What We Talked About

- **file system (FS)** as an OS component
  - role
  - general structure
- **file** concept
  - used for **storing user data**
  - components: **data** (container component) and **meta-data** (description component)
  - meta-data fields: name, types, attributes etc.
  - **structure**: sequence of bytes, specialized collection of elements
- **directory** concept
  - used for **organizing the file space**
  - a system file: its bytes are interpreted by the OS and provided as a collection of elements
  - imposes a **hierarchy**
  - FS elements (e.g. files) specified by a **path**
  - absolute vs relative path

4.1.33

Lessons Learned

- the basic storage unit for the user (applications) is the file
  - $\Rightarrow$ storing just a single byte of data needs placing it in a file
- OS usually not involved in the interpretation of a file's contents
  - $\Rightarrow$ file structure (format) is the user-application business
- text vs binary files is just a user convention
  - from the OS perspective all files consist just in a sequence of bytes
- FS organization, i.e. classifying files, is a real need
  - helps the user to navigate visually in the FS space in order to find her files
  - does not help (too much) the user applications finding files

Chapter 4.2

Linux and Windows File Systems

Permission Rights and System Calls

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Colesa

March 25, 2020

4.2.1

Purpose and Contents

The purpose of today's lecture

- Presents and compare *permission rights* in Linux and Windows
- Presents and compare *system calls for files* in Linux and Windows

4.2.2

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 10. Case Study 1: Unix and Linux, pg. 732 - 744, p. 753 - 757
- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 11. Case Study 2: Windows 2000, pg. 830 - 833, p. 844 - 847
- Lab texts related to Linux's and Windows' file system and their system calls.
- From `http://msdn.microsoft.com/en-us/library/aa364407(VS.85).aspx` about File management and Directory Management (following the lecture slides)

4.2.3

Contents

4.2.4

1 Permission Rights

1.1 Linux Permission Rights

Basic Permission Rights

- defined for three classes of users
  - owner or user (`u`)

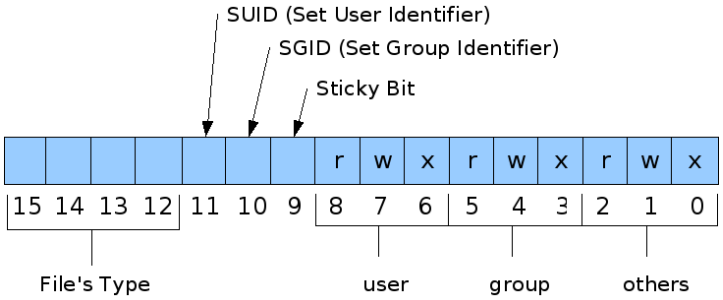


Figure 1: The way permission rights are stored

  - groups (g) the owner belongs to
  - others (o)
- operations (types)
  - *read* (r)
  - *write* (w)
  - *execute* (x)
- See examples running `ls -l` command on different files and directories

4.2.5

I-node Field Structure

- bit value: 0 / 1 → denied / allowed
- example
  - string: `rw-r--r--`
  - binary: `110100100`
  - octal: `0644`

4.2.6

Permissions on Regular Files

- `read`: read file's contents
- `write`: write (modify, append to, truncate) file's contents
- `execute`: execute file

4.2.7

Permissions on Directories

- `read`: read (list) directory's contents
- `write`: write (*modify, add and remove elements*) file's contents
  - confusing and too limited
- `execute`: **traverse directory**, i.e. search for an element in the directory
- ⇒ **Read and/or Write without Execute not so useful**
  - but . . . **Execute without Read and/or Write makes sense**
  - when we want a directory to be traversed, but its contents not be visible
  - commonly used in practice for the `/home` directory

4.2.8

Basic permission rights for FS elements in Linux are `r`, `w`, `x` for `u`, `g`, `o` `rwxrwxrwx`

4.2.9

Questions (1)

Give the equivalent permission right representation for the following cases?

1. `rwxr-xr--`
2. `r--r--r--`
3. `0765`

4.2.10

Questions (2)

Which of the following operations

(O1) `ls /home/os`
(O2) `cat /home/os/file.txt`
(O3) `rm /home/os/file.txt`

could be performed by the "`os`" user, supposing the directory "`/home/os`" is its home directory and has the following permission rights (all the file in "`/home/os`" have `r--r--r--` permissions)?

(P1) `r-xr--r--`
(P2) `--x--x--x`
(P3) `rw-r--r--`

4.2.11

Special Techniques: `SUID` and `SGID` Bits

- SUID
  - has effect only for executable files
  - the process resulting from the corresponding executable file will have the effective UID that of the owner of the file
  - see the classical example of `/usr/bin/passwd` executable file, that can be run by any *non-privileged user*, modifying the `/etc/passwd` file belonging to *root*
- SGID
  - similar to SUID but applies to files's GID

4.2.12

Special Techniques: Sticky Bit

- has effect only for directories
- allows elements to be removed only by their owner
- see `/tmp` directory

4.2.13

Extended Attributes and ACL-like Permission Rights

- extended attributes
  - extra attributes like "append-only"
  - see "man chattr"
- Access Control List (ACL)
  - list of complementary permission rights per user / group
  - see "man getfacl"

4.2.14

Security Considerations: Weak Permissions

- **permission rights are essential** for
  - file protection
  - **system security**
- pay attention to
  - **not allow** read / write permission on vital system files
  - **not allow** write permission on system directories
  - **not trust** files / directories writable by regular users (possible attackers)
- attack scenarios
  - readable "`/etc/shadow`" file allows for brute-force password guess
  - writable "`/etc/passwd`" allows for creation of new users
  - writable "`/etc/sudoers`" allows getting root (admin) permissions
  - writable "`/sbin`" allows replacement of system executables
  - ...

Security Considerations: Study Case (Bad Code)

Consider the following C program "`fopen-permissions.c`"

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main( void)
{
    FILE *f;

    umask(0000);

    if ((f=fopen("TEST", "w+")) == NULL) {
        perror("File creation error");
        exit (1);
    }

    printf("File TEST created with the following permission rights\n");

    system("stat --format=%A TEST");

    unlink("TEST");

    return 0;
}
```

When run, the program displays

```
$ gcc -Wall fopen-permissions.c -o fopen-permissions

$ ./fopen-permissions
File TEST created with the following permission rights
-rw-rw-rw-
```


Security Considerations: Study Case (Good Code)

Consider the following C program "`fopen-permissions.c`"

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main( void)
{
    FILE *f;

    umask(0022);

    if ((f=fopen("TEST", "w+")) == NULL) {
        perror("File creation error");
        exit (1);
    }

    printf("File TEST created with the following permission rights\n");

    system("stat --format=%A TEST");

    unlink("TEST");

    return 0;
}
```

When run, the program displays

```
$ gcc -Wall fopen-permissions.c -o fopen-permissions

$ ./fopen-permissions
File TEST created with the following permission rights
-rw-r--r--
```

4.2.17

Never trust the users! Protect files using the appropriate permission rights!

4.2.18

1.2 Windows' Permission Rights

Strategy

- Permission rights are defined for each user using ACLs (*Access Control Lists*)
- Basically, they are: *read* (`r`), *write* (`w`), *execute* (`x`)

4.2.19

Simplified (Synthesized) View

- Files
  - `read`: permits viewing or accessing of the file's contents
  - `write`: permits writing to a file
  - `read&execute`: permits viewing and accessing of the file's contents as well as executing of the file
  - `modify`: permits reading and writing of the file; allows deletion of the file
  - `full control`:permits reading, writing, changing and deleting of the file
- Directories
  - `read`: permits viewing and listing of files and subdirectories
  - `write`: permits adding of files and subdirectories
  - `read&execute`: permits viewing and listing of files and subdirectories, as well as executing of files
  - `modify`: permits reading and writing of files and subdirectories; allows deletion of the directory
  - `full control`: permits reading, writing, changing, and deleting of files and subdirectories

4.2.20

Advanced View

- Traverse Folder/Execute File
- List Folder/Read Data
- Read Attributes
- Read Extended Attributes
- Create Files/Write Data
- Create Folders/Append Data
- Write Attributes
- Write Extended Attributes
- Delete Subfolders and Files
- Delete
- Read Permissions
- Change Permissions
- Take Ownership

4.2.21

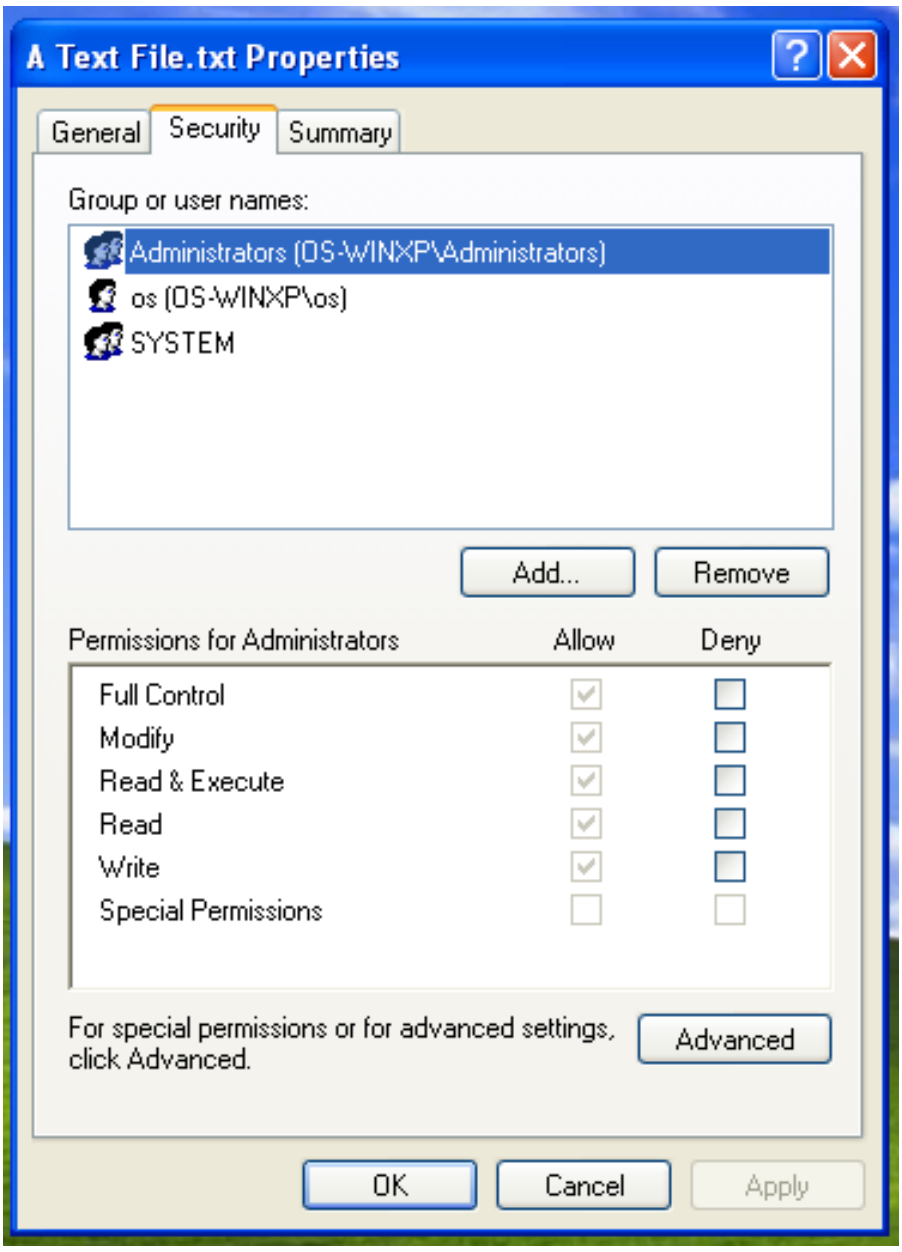


Figure 2: The Simplified View of Permission Rights

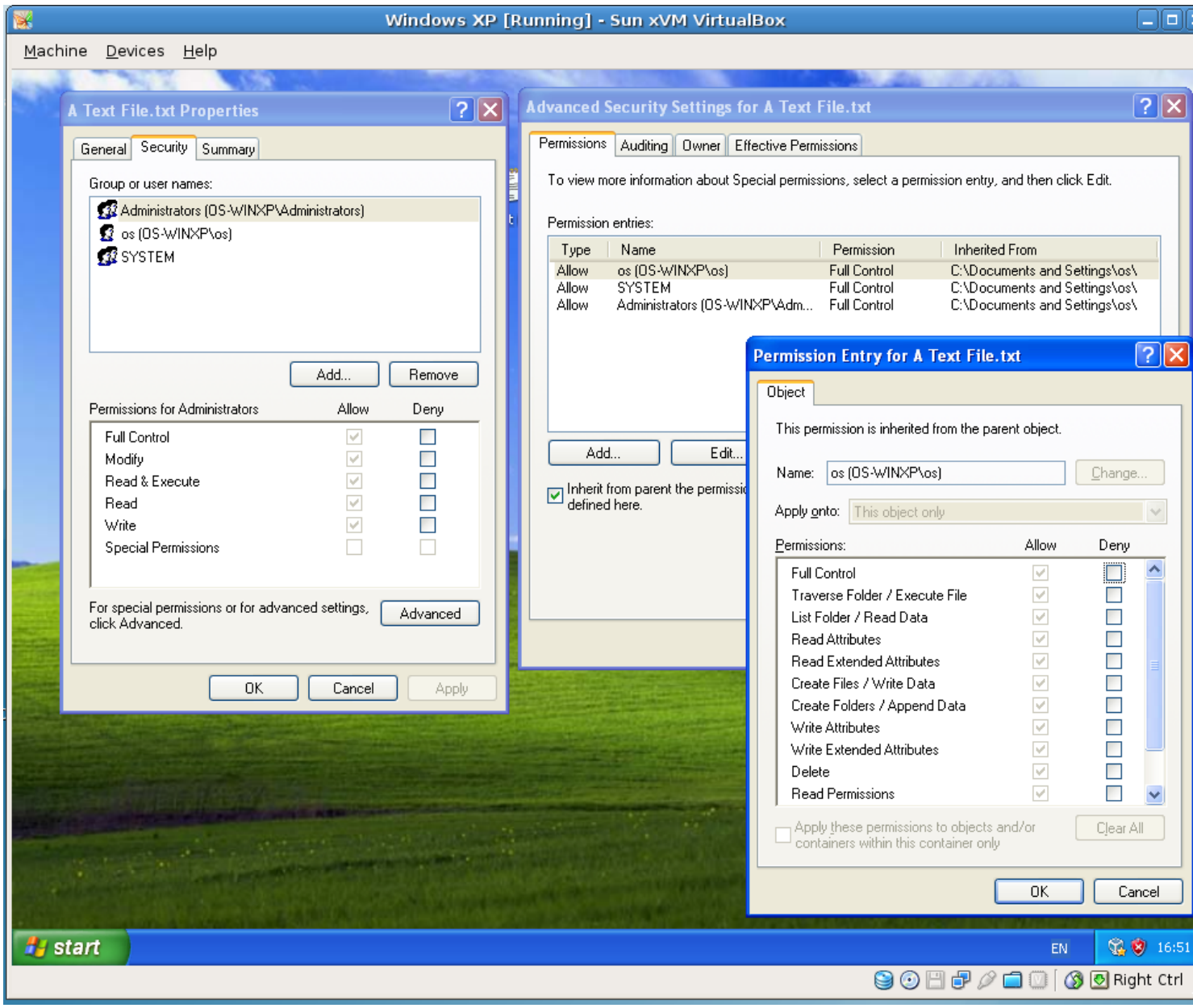


Figure 3: The Way of Setting Detailed Permission Rights (Windows XP)

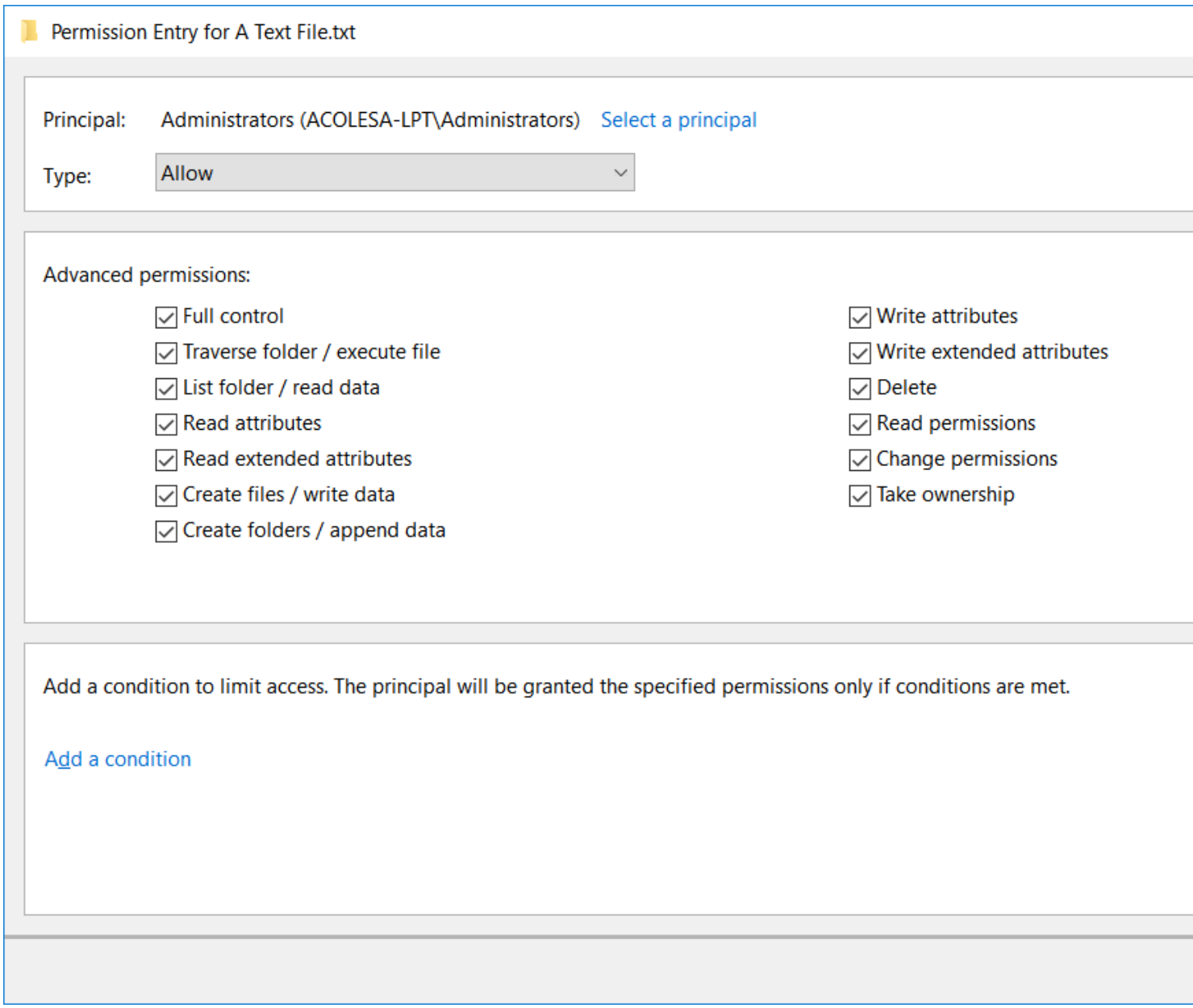


Figure 4: The Advanced View of Permission Rights (Windows 10)

	Full control	Modify	Read & Execute	List folder contents	Read	Write	Special permissions
Full control	X						
Traverse folder/Execute file	X	X	X	X			
List folder / Read data	X	X	X	X	X		
Read Attributes	X	X	X	X	X		
Read extended attributes	X	X	X	X	X		
Create files/Write data	X	X				X	
Create folders/Append data	X	X				X	
Write attributes	X	X				X	
Write extended attributes	X	X				X	
Delete subfolders and files	X	X					X
Delete	X	X					
Read permissions	X	X	X	X	X		
Change permissions	X						X
Take ownership	X						X

Relationship Between Synthesized and Advanced Permission Rights

4.2.22

2 Basic System Calls on Linux and Windows

2.1 Linux

Access File Data

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int close(int fd);

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

4.2.23

Manipulate Files

```
int creat(const char *pathname, mode_t mode);
int rename(const char *oldpath, const char *newpath);
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

4.2.24

Manipulate Directories

```
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);
int symlink(const char *oldpath, const char *newpath);
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
off_t telldir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
int closedir(DIR *dir);
```

4.2.25

Create and Remove a File

```
int fd;

  // create a new file
fd = creat("/home/os/file", 0600);
    // file size = 0 (no space allocated)
    // only the i-node (metadata) allocated
    // if file exists, it is truncated
    // the new file is opened for WRONLY
    // note permissions: rw-------

  // remove the file (remove a link to the file)
unlink("/home/os/file");
```

4.2.26

File Structure (Format): What Is The File Provided Like?

- no (special) structure
- just a sequence (stream) of bytes
  - each byte has its fixed position (offset)
- **no special byte**(s) inside the file to mark the end of file
  - every byte in the file could have any possible (user-provided) value
  - the file size kept as an i-node field

4.2.27

Relationship Between the OS and User Views of A File

- user (application) view
  - unstructured sequence of bytes
  - $\Rightarrow$ a logically contiguous area
- OS logical view
  - sequence of blocks
  - $\Rightarrow$ a logically contiguous area
- OS physical view
  - collection of blocks
  - $\Rightarrow$ a collection of more physical contiguous areas

4.2.28

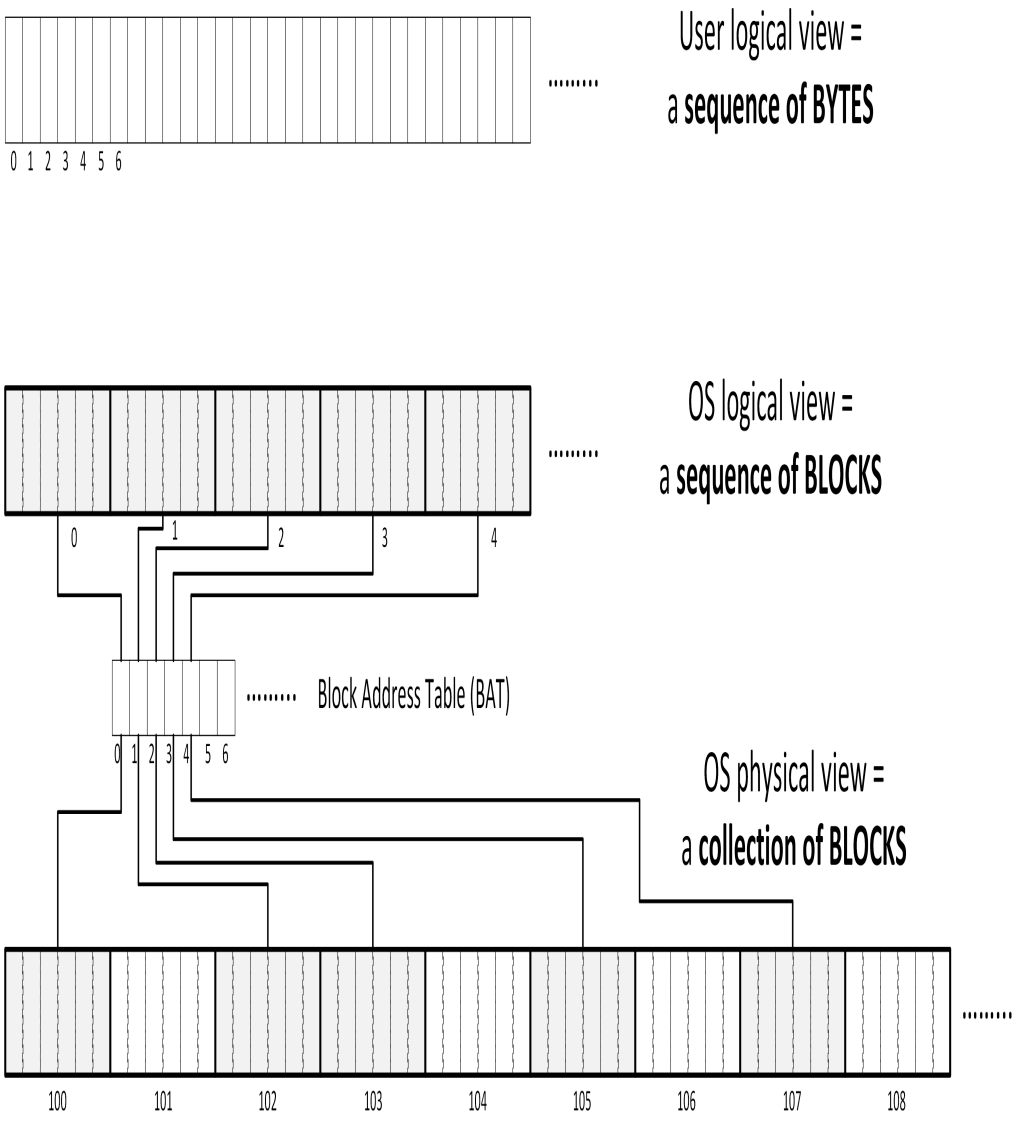
Illustration of Different File Views

4.2.29

Open Files Management

- accessing a file is done using open files
  - opening a file let the OS prepare for the next read/write operations
  - let the OS be very efficient
- OS maintains three types of tables (i.e. internal structures)
  - **i-node table (IT)**: one per system
  - **open file table (OFT)**: one per system
  - **file descriptor table** (FDT): one for each process
- **every** `open` $\Rightarrow$ a new *open file* $\Rightarrow$ **different entry in OFT**
  - read operations are independent
  - write operations are independent, but ...
  - write effect is immediately visible (*one-copy semantic*)

4.2.30


User logical view =
a sequence of BYTES
0 1 2 3 4 5 6
OS logical view =
a sequence of BLOCKS
0
1
2
3
4
Block Address Table (BAT)
0 1 2 3 4 5 6
OS physical view =
a collection of BLOCKS
100
101
102
103
104
105
106
107
108
HDD (storage area)

Open a File

```
int fd;

  // open an existing file
fd = open("/home/os/file_1", O_RDWR);
    // file must exist
    // opened for both RD and WR (if allowed by persmission rights)

  // ALWAYS CHECK return values of I/O operations!
if (fd < 0) {
    // -1 returned in case of error
    // e.g. file does not exists (wrong filepath)
    // e.g. permission denied
    perror("File cannot be opened");  // display the system err msg.
    exit(1);                          // terminate program
}

  // create a file with "open"
fd = open("/home/os/file_2", O_CREAT | O_EXCL | O_RDWR, 0600);
    // O_EXCL check if file does not already exist
```

4.2.31

Reading From/Writing To A File

- operations are performed relative to the *current position*
  - most of the cases at beginning of file after open
- current position is advanced (increased)
  - with the number of bytes successfully read or written
  - by each `read` and `write`, respectively
- both `read` and `write` syscalls
  - use *memory address* where bytes are
    - written to (after being *read*)
    - taken from (to be *written*)
  - return the number of bytes successfully read or written
- ***end of file (EOF)* detected when `read` returns 0 (zero)**

4.2.32

Example: Opening Files, Reading From Them, Creating Duplicates

```
// Process 1
int fd1, fd2;
char buf[100]="1234567890";

fd1=open("file1", O_RDWR);       // fd1 = 3
fd2=open("file1", O_RDONLY);     // fd2 = 4
write(fd1, buf, 10);             // write "1234567890"
read(fd2, buf, 5);               // read "12345"

// Process 2
int fd1, fd2, fd3;
char buf[100];

fd1=open("file1", O_RDWR);       // fd1 = 3
fd2=open("file2", O_RDWR);       // fd2 = 4
fd3=dup(fd2);                    // fd3 = 5
write(fd2, buf, 100);            // write first 100 bytes
write(fd3, buf, 100);            // write next 100 bytes
```

4.2.33

Example: Open File Tables Illustration

4.2.34

Each "`open()`" leads to a new, independent open file, i.e. a new entry in OFT and FDT.

4.2.35

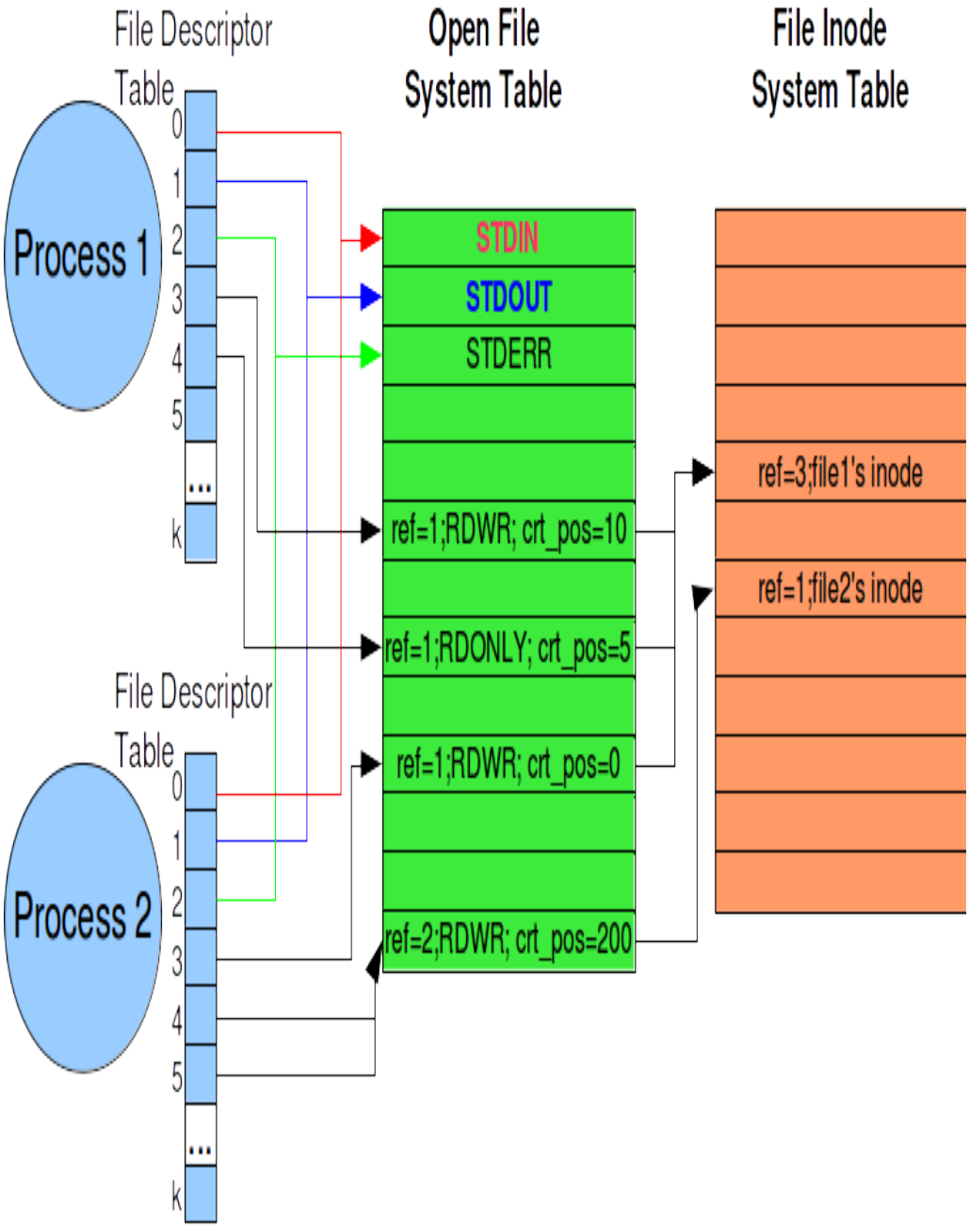


Figure 5: The Effect of More Open and Read Operations

File Format: Text Files

- each byte has its own interpretation
  - contains the code of a printable character
- special characters: new line (`0x0A`)
- OS knows nothing about (ANY) file's format
  - ⇒ `read` has no notion of reading until some special char
  - i.e. `read` cannot read a text line
  - **end of line has to be detected by application**

4.2.36

Example: Read the First Text Line

```
#define MAX_LINE 1024

int fd, i;
char c;
char line[MAX_LINE+1];

fd = open("file.txt", O_RDONLY);
if (fd < 0) {
    perror("Cannot open the file");
    exit(1);
}

i=0;
while ( (i < MAX_LINE) && (read(fd, &c, 1) > 0) && (c != '\n')) {
    line[i] = c;
    i++;
}

line[i] = '\0';
printf("The read line is: %s\n", line);
```

4.2.37

File Format: Binary Files

- any non-text file is a binary file
- actually, **any file is a binary file**
  - just a stream of bytes
- ⇒ **applications have to know**
  - formats of files they work with (e.g. `pdf`, `docx` etc.)
  - i.e. **the way the bytes must be grouped** for good interpretation
  - i.e. the offsets real information (interpretable groups of bytes) is placed

4.2.38

Example: Write / Read into / from Binary Files

- process 1

```
int fd;
int number = 10;
char c = 'A';

if ((fd = creat("intfile.bin", 0644)) < 0) {
    perror("Cannot create the file");
    exit(1);
}

// write a char on the first byte
write(fd, &c, sizeof(c));

// write an integer's representation on the next four bytes
write(fd, &number, sizeof(number));
```

- process 2

```
int fd;
int number;

if ((fd = open("intfile.bin", O_RDONLY)) < 0) {
    perror("Cannot open the file");
    exit(1);
}
```

```
// position where WE (MUST) KNOW the integer is
// i.e. one byte after beginning of file
lseek(fd, sizeof(char), SEEK_SET);

// read four bytes from crt position
// i.e. an integer's representation
read(fd, &number, sizeof(number));
```

4.2.39

Files With Holes (Gaps). Description

- **`lseek` is allowed to position after the end of file**
- **when write** to that position
  - **a gap results in file**
  - i.e. unwritten space
- Linux does not allocate physical space for gaps
- **`read` returns zeros from a gap**
  - there is no difference between reading previously written zeros or reading from a gap
  - it is the application's responsibility to remember where gaps are (manages them)
- usage
  - core dumps for crashed processes
  - virtual HDD files (dynamically allocated)

4.2.40

Files With Holes (Gaps). Example

- Example 1: Command line

```
dd if=/dev/zero of=file bs=1024 count=1 seek=1024
```

- Example 2: C Program

```
int fd;
  // create a zero-sized file
fd = creat("file.with.gaps.bin", 0644);
if (fd < 0) {
    perror("Cannot create file");
    exit(1);
}

  // position 4KiB after the end of file
lseek(fd, 4096, SEEK_END);
        // has no effect: only modify the crt position

  // write at crt position
write (fd, "END", 3);
        // create a gap (unwritten bytes) of 4KB,
        // followed by 3 written bytes
        // file's size is now 4096 + 3 = 4099
```

4.2.41

File format (text, binary, holes etc.) is the business of user applications — OS is not involved!

4.2.42

Input/Output Redirection. Example

```
int fd1, fd2, n1, n2;

fd1 = open("input.txt", O_RDOLNY);
fd2 = creat("output.txt", O_WRONLY);

  // reads an integer from STDIN
scanf("%d", &n1);   // calls read(0, ...);

  // redirects STDIN
close(0);           // breaks the initial association between 0 and STDIN
dup(fd1);           // associates 0 with the same open file like fd1

  // reads an integer from STDIN
scanf("%d", &n2);   // calls read(0, ...);
    // actually reads from "input.txt"

// writes an integer to STDOUT
printf("%d\n", n1); // calls write(1, ...);

// redirects STDOUT
dup2(fd2, 1);       // makes 1 a duplicate for fd2

// writes an integer to STDOUT
printf("%d\n", n2); // calls write(1, ...);
    // actually writes to "output.txt"
```

4.2.43

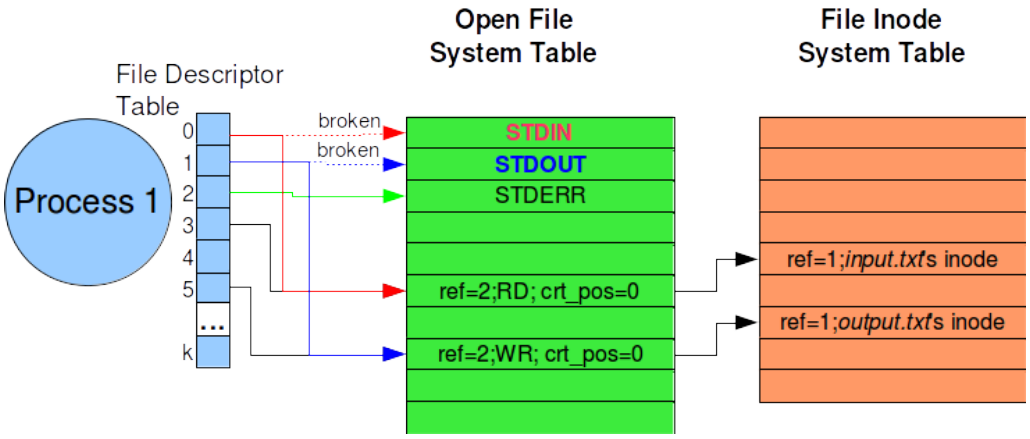


Figure 6: The effect of redirecting STDIN and STDOUT

Input/Output Redirection. Illustration on Open File Tables 4.2.44

Standard channel (STDIN, STDOUT, STDERR) redirection is possible because any resource in the system is modeled as a file!

4.2.45

Changing Permission Rights

- specify permission rights for all three groups of users
  - user: `rwx` → 111 (7)
  - group: `r-x` → 101 (5)
  - others: `--x` → 001 (1)
- example

```
chmod("file", 0751);
```

4.2.46

Getting Information (Metadata) About a File

- Linux metadata = **i-node (information node)**
- **each file and directory has its own, unique i-node**
- all i-nodes
  - have the same fixed size
  - placed together in one HDD area (inode area)
- ⇒ **i-node numbers** used as an index to identify an i-node
- i-node contents
  - file type, size, owner, group, permissions etc.

4.2.47

Example: Get a File's I-Node

```
int res;
struct stat inode;

  // gets file's inode
res = lstat("/home/os/input.txt", &inode);
if (res < 0) {
    perror("Cannot get file inode");
}

  // identify file's type
if (S_ISREG(inode.st_mode)) {
```

```
    printf("It is a file\n");
    printf("File's size [bytes]: %d\n", inode.st_size);
}

if (S_ISDIR(inode.st_mode))
    printf("It is a directory\n");

if (S_ISLNK(inode.st_mode))
    printf("It is a symbolic link\n");
```

4.2.48

Reading Directory Contents

- directory provided as **a collection of elements**
  - named **directory entries**
- the internal structure of directory (linked-list, B-tree) not visible
  - $\Rightarrow$ the only way to **read a directory** is **entry by entry**
  - i.e. **sequential access**
- a directory entry contains (at least)
  - name (e.g. file or subdirectory)
  - i-node number (not of real interest)
- take care of "." and ".."
  - "." points to the current directory
  - ".." points to the parent directory
  - exists as real elements in a directory
  - they could induce cycles in applications that traverse a file tree

4.2.49

Reading Directory Contents

```
DIR* dir;
struct dirent *entry;
char path[MAX_PATH];
struct stat file_info;

  // open directory
if ((dir = opendir("/home/os")) == NULL) {
    perror("Cannot open the directory");
    exit(1);
}

  // read one-by-one dir entries until NULL returned
while ( (entry = readdir(dir)) != NULL) {
      // avoid "." and ".." as they are not useful
    if (strcmp(entry->d_name, ".") &&  strcmp(entry->d_name, "..")) {

          // build the complete path = dirpath + direntry's name
        sprintf(path, "%s/%s", "/home/os", entry->d_name);

          // get element's inode
        stat(path, &file_info);

          // identify type
        if (S_ISREG(file_info.st_mode))
            printf("%s is a file\n", path);
        else
            if (S_ISDIR(file_info.st_mode))
                printf("%s is a dir\n", path);
    }
}
```

4.2.50

Searching for an Element in a Directory

- naive, inefficient way (does not benefit from the specialized directory structure)

```
DIR* dir;
struct dirent *entry;
char path[MAX_PATH];
struct stat file_info;

  // open directory
if ((dir = opendir("/home/os")) == NULL) {
    perror("Cannot open the directory");
    exit(1);
}

  // read one-by-one dir entries until NULL returned
while ( (entry = readdir(dir)) != NULL) {
    if (strcmp(entry->d_name, SEARCHED_NAME) == 0) {
          // build the complete path = dirpath + direntry's name
        sprintf(path, "%s/%s", "/home/os", entry->d_name);
        printf("Found %s\n", path);
        break;
    }
}
```

- efficient way (benefit from the specialized directory structure)

```
char path[MAX_PATH];
struct stat file_info;

// build the complete path = dirpath + SEARCHED_NAME
sprintf(path, "%s/%s", "/home/os", SEARCHED_NAME);

// try getting file info
// succeds if file exists, fails otherwise
if (stat(path, &file_info) > 0) {
    printf("Found %s\n", path);
} else {
    perror("Cannot get file's inode");
    exit(1);
}
```

4.2.51

A directory is presented and interacted with as a collection of elements! Searching, traversing, creating, changing could be performed only at directory element level.

4.2.52

Security Considerations: File-Path Traversal

- context
  - an application wants to **confine access** (of its users) **to a subdirectory** (subtree)
  - very common to Web applications
- problem
  - when user controls (specifies) parts of the file path
  - **".." could be used to evade** the restricted directory
- solution
  - check for ".." in user-controlled file paths

4.2.53

File-Path Traversal Illustration

- vulnerable code

```
char filepath[MAX_PATH];

scanf("%s", filename);  // <--- provided by the user (a possible attacker)!!!

snprintf(filepath, MAX_PATH, "/home/restricted_user/%s", filename);

fd = open(filepath, O_RDONLY);
...  // display the file's contents
```

- malicious value for "`filename`"

```
"../../../../../../etc/passwd"
```

- secure code

```
char filepath[MAX_PATH];

scanf("%s", filename);
if (strstr(filename, "..") != NULL)
    return;
snprintf(filepath, MAX_PATH, "/home/restricted_user/%s", filename);

fd = open(filepath, O_RDONLY);
```

4.2.54

Never trust the users! Check for ".." in the given file paths, if want avoiding path traversal!

4.2.55

Trace System Calls of An Application

- Can be done using
  - the `strace` command
  - the `ptrace` system call
- Example
  - create a file of 8 KB

    ```
    dd if=/dev/zero of=file1 bs=4096 count=2
    ```

  - trace the `cp` command's execution

    ```
    strace cp file1 file2
    ```

  - main part of the output

    ```
    stat64("file1", {st_mode=S_IFREG|0644, st_size=8192, ...}) = 0
    open("file1", O_RDONLY|O_LARGEFILE)     = 3
    stat64("file2", 0xbf850e94)             = -1 ENOENT (No such file or directory)
    open("file2", O_WRONLY|O_CREAT|O_LARGEFILE, 0100644) = 4
    fstat64(4, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
    fstat64(3, {st_mode=S_IFREG|0644, st_size=8192, ...}) = 0
    read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
    write(4, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
    read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
    write(4, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 4096) = 4096
    read(3, "", 4096)                       = 0
    close(4)                                = 0
    close(3)                                = 0
    ```

4.2.56

Questions (3)

- Show the **current position** for each open file (i.e. file descriptor) and the **contents of "buf"** after the execution of **each instruction** from the code below, supposing they are ALL executed successfully.

```
1  char buf[100];
2
3  int fd1 = open("file1", O_RDONLY);
4  int fd2 = open("file1", O_RDWR);
5  int fd3 = open("file2", O_RDWR);
6  int fd4 = dup(fd3);
7  write(fd2, "This is funny, isn't it?", 10);
8  read(fd1, buf, 4);
9  write(fd3, "1234567890", 10);
10 lseek(fd3, 0, SEEK_SET);
11 read(fd3, buf, 5);
12 read(fd4, buf, 5);
```

4.2.57

2.2 Windows

Access File Data

```
HANDLE WINAPI CreateFile(LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode,
        LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDisposition,
        DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);

BOOL WINAPI ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead,
        LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);

BOOL WINAPI WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite,
        LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped);

DWORD WINAPI SetFilePointer(HANDLE hFile, LONG lDistanceToMove,
                        PLONG lpDistanceToMoveHigh, DWORD dwMoveMethod);
```

4.2.58

Manipulate Files

```
BOOL WINAPI MoveFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName);

BOOL WINAPI SetEndOfFile(HANDLE hFile);

DWORD WINAPI GetFileAttributes(LPCTSTR lpFileName);

BOOL WINAPI GetFileInformationByHandle(HANDLE hFile,
        LPBY_HANDLE_FILE_INFORMATION lpFileInformation);

BOOL WINAPI GetFileSecurity(LPCTSTR lpFileName, SECURITY_INFORMATION RequestedInformation,
        PSECURITY_DESCRIPTOR pSecurityDescriptor, DWORD nLength,LPDWORD lpnLengthNeeded);

BOOL WINAPI SetFileSecurity(LPCTSTR lpFileName, SECURITY_INFORMATION SecurityInformation,
        PSECURITY_DESCRIPTOR pSecurityDescriptor);
```

4.2.59

Manipulate Directories

```
BOOL WINAPI CreateDirectory(LPCTSTR lpPathName, LPSECURITY_ATTRIBUTES lpSecurityAttributes);

BOOL WINAPI CreateHardLink(LPCTSTR lpFileName, LPCTSTR lpExistingFileName,
         LPSECURITY_ATTRIBUTES lpSecurityAttributes);

BOOLEAN WINAPI CreateSymbolicLink(LPTSTR lpSymlinkFileName, LPTSTR lpTargetFileName, DWORD dwFlags);

BOOL WINAPI DeleteFile(LPCTSTR lpFileName);

HANDLE WINAPI FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA lpFindFileData);

BOOL WINAPI FindNextFile(HANDLE hFindFile, LPWIN32_FIND_DATA lpFindFileData);

BOOL WINAPI RemoveDirectory(LPCTSTR lpPathName);
```

4.2.60

Sparse Files

```
LARGE_INTEGER   FileSize;


FileSize.QuadPart = 8 * 1024 * 1024 * 1024;

FileHandle = CreateFile("file",
               GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE,
               NULL, CREATE_NEW, FILE_FLAG_NO_BUFFERING, NULL);


DeviceIoControl(FileHandle, FSCTL_SET_SPARSE, NULL, 0,
               NULL, 0, &BytesReturned, NULL);

SetFilePointerEx(FileHandle, FileSize, 0, FILE_BEGIN);

SetEndOfFile(FileHandle);
```

4.2.61

Alternate Data Streams. Command Line Examples

- Example 1

  ```
  echo hello > file.txt:alternatestream.txt

  more < file.txt:alternatestream.txt

  notepad file.txt:alternatestream.txt
  ```

- Example 2

  ```
  type c:\windows\systems32\calc.exe > file.txt:calc.exe

  start .\file.txt:calc.exe
  ```

- Getting alternate streams

  ```
  http://www.microsoft.com/technet/sysinternals/default.mspx

  streams [-s] [-d] <file\_name>
  ```

4.2.62

Alternate Data Streams. C Program Example

```
HANDLE inhandle, outhandle;
char buffer[BUF_SIZE];
int count, s;
DWORD ocnt;

inhandle = CreateFile("sursa.txt", GENERIC_READ, 0,
                      NULL, OPEN_EXISTING, 0, NULL);

outhandle = CreateFile("dest.txt:file.txt", GENERIC_WRITE,
                       0, NULL, CREATE_ALWAYS,
                       FILE_ATTRIBUTE_NORMAL, NULL);

/* copy the file  */
do {
      s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);

      if (s && count > 0)
          WriteFile(outhandle, buffer, count, &ocnt, NULL);

} while (s>0 && count>0);
```

4.2.63

3 Conclusions

What We Talked About

- permission rights
  - similar on both Linux and Windows
  - read (r), write (w), execute (x)
  - for both files and directories
- system calls and how they provide access to files and directories
  - creat, unlink, rename, truncate
  - open read, write, lseek, close
  - opendir, readidir, closedir
- security considerations
  - weak permission rights
  - path traversal

4.2.64

Lessons Learned

- the “file” is a sequence of bytes
  - each application should manage its own files’ format
- the “directory” is a collection of elements
  - traverse it element by element
  - search for elements
- do not trust users!
  - set strong permission rights!
  - check for path traversal elements!

4.2.65

Chapter 4.3

File System's Physical Data Layer

Implementation, Fragmentation, Links and Backup

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

March 24, 2021

4.3.1

Purpose and Contents

The purpose of today's lecture

- Presents details about the way files and directories are implemented.
- Presents related strategies and problems like: fragmentation, links, backup.

4.3.2

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 6, pg. 399 – 421

4.3.3

Contents

4.3.4

1 File's Data Allocation

Context

- files are
  - **provided to user application as sequences at bytes**
    - a logical view
    - a contiguous area
  - **allocated in terms of blocks**, i.e. group of bytes, on HDD
    - a physical view
    - not necessarily a contiguous area
- we are interested in
  - how blocks of a file are allocated on HDD
  - how does the allocation strategy influence the user applications

4.3.5

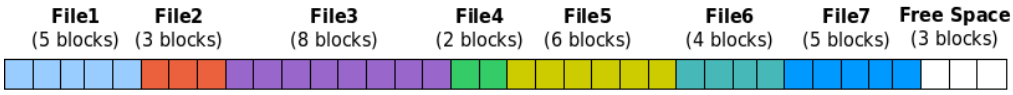


Figure 1: HDD Partition Structure

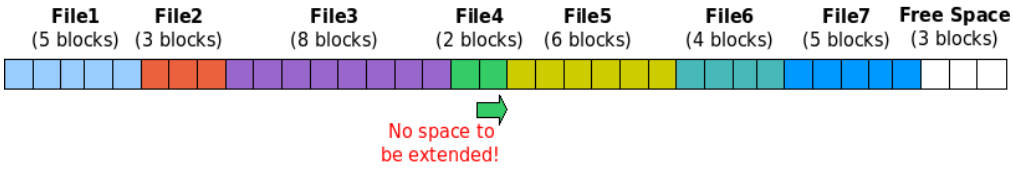


Figure 2: HDD Partition Structure

Contiguous Allocation

- Files are allocated in **just one contiguous area**
- Advantages
  - Reading large areas from the file is very fast
  - Keeping track of allocated blocks (BAT) is very simple: starting block and the number of allocated blocks
- Disadvantages
  - Difficult to increase the file size: see for example **File 4**
  - Leads to **external fragmentation**
  - Complex allocation strategies: **first fit**, **best fit** etc.

4.3.6

Any-Free-Block Allocation

- The file can be allocated any free block
- Advantages
  - there is no external fragmentation; any free block can be used
  - file size can be easily extended
  - could be combined with contiguous allocation: the file is allocated more contiguous areas (as large as possible)
- Disadvantages
  - data access (e.g. read entire file) not so efficient
  - BAT structure more complicated
  - still suffers from **internal fragmentation** and **data fragmentation**

4.3.7

Fragmentation Types: External Fragmentation

- context
  - **free space scattered in small areas** over the entire HDD
  - alternating with allocated areas

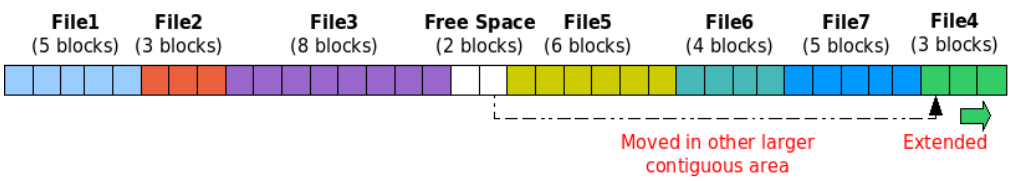


Figure 3: HDD Partition Structure

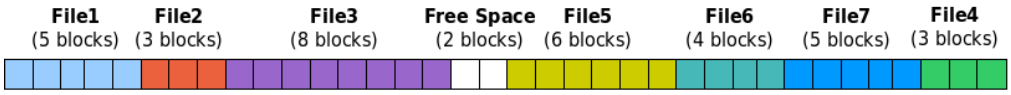


Figure 4: HDD Partition Structure

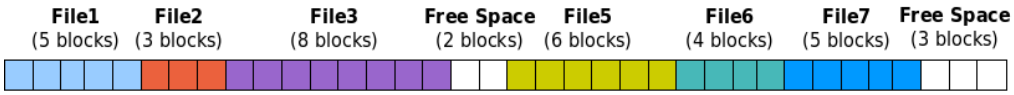


Figure 5: HDD Partition Structure

- problem
  - **free space cannot be used** (in some situations)
  - example
    - need to allocate a contiguous area of a give size
    - but no free contiguous area could be available
    - even if total free space would be enough
- specific to
  - contiguous allocation strategies
  - where data can be allocated **only in a single contiguous area**
- solution (inefficient, i.e. time consuming)
  - **defragmentation**: move all allocated space at one end of the HDD
  - $\Rightarrow$ free space in a single contiguous area at the other end of HDD

4.3.8

Fragmentation Types: Internal Fragmentation

- context
  - any free block could be allocated when new space needed
  - BUT . . . **allocation is done in terms of predefined units**, i.e. blocks
  - AND . . . **needed space not a multiple of block size**
- problem
  - some (internal) “free” space cannot be used
  - **unused space in blocks allocated to files** is lost
- specific to
  - allocation strategies that allocate data in terms of fixed-size blocks
- solutions
  - **smaller block size** to reduce internal fragmentation
  - **fragment blocks**, i.e. share the same block for tails of multiple files

4.3.9

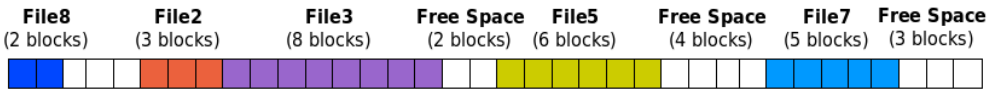


Figure 6: HDD Partition Structure

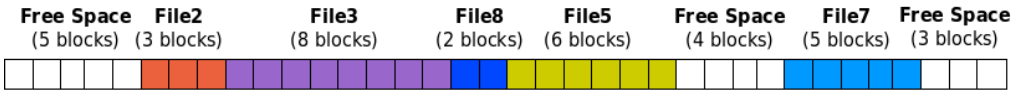


Figure 7: HDD Partition Structure

Fragmentation Types: Data Fragmentation

- context
  - the allocated space of a file is distributed in different non-consecutive blocks
  - i.e. more contiguous areas
- problem
  - **file access could suffer performance penalties**
- specific to
  - **any-free-block allocation strategy**
  - that do not impose the file to be in a single contiguous area
- solution
  - **defragmentation**: reallocate file's blocks in consecutive ones

4.3.10

File System Block Size

- The problem: How large the block should be?
- Alternatives
  - Larger
    - efficient read of file data
    - increase internal fragmentation (waste HDD space)
  - Smaller
    - reduce internal fragmentation
    - increase data fragmentation and data access time (many disk accesses)
- **no good-for-all solution**
  - *performance* and *space utilization* are inherently in conflict
  - the block size should be chosen knowing the way and for what the HDD partition will be used
  - a compromise should be chosen in a general usage case

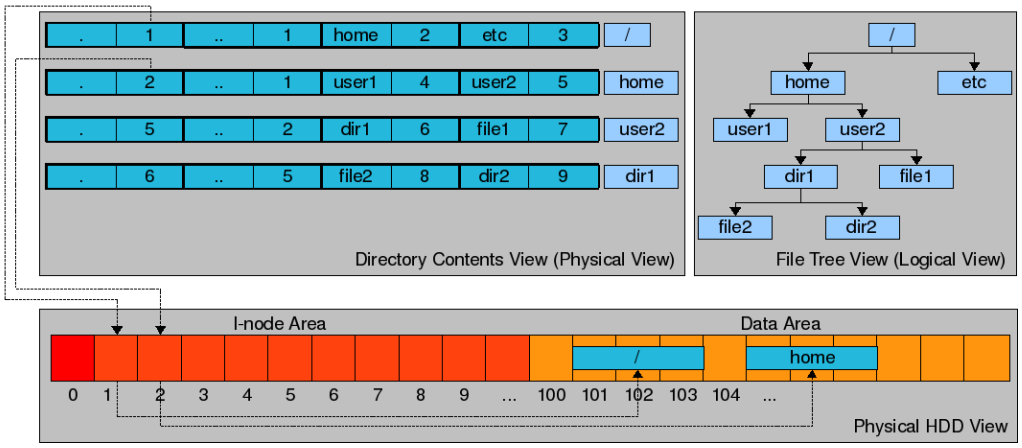
4.3.11

2 Directory Implementation

Directory Contents

- a system "file"
- stored as a stream of bytes, but interpreted by the OS
- organized as a collection of records (elements), called **directory entries**
- a directory entry contains
  - the (file, directory etc.) name
  - file's metadata or a reference to them

4.3.12



File "I-node" (Record)

- a physical space (element) and a corresponding data structure used to store information about a FS element (file, directory etc.)
- stores file meta-data, like
  - file type
  - size
  - owner
  - permission rights
  - time stamps
  - the BAT (Block Addresses Table)
  - etc.

4.3.13

The Relationship Between The Directory Entry and The I-node: Illustration

4.3.14

3 Hard and Symbolic Links

Sharing Data Between Directories

- make a file (directory) appear in different directories
- the operation is called linking files
- two kinds of links
  - hard (physical)
  - soft (symbolic)

4.3.15

Hard Link: Creation and Usage

```
create("/home/os/dir1/f1", 0600);    // only allocate i-node; no data
link("/home/os/dir1/f1", "/home/os/dir2/f2"); // hard link
link("/home/os/dir1/f1", "/home/os/dir2/f1"); // hard link
open("/home/os/dir1/f1", ...);  // open file with i-node 5
open("/home/os/dir2/f2", ...);  // open file with i-node 5
open("/home/os/dir2/f1", ...);  // open file with i-node 5
stat("/home/os/dir1/f1", ...);  // read i-node 5 contents
stat("/home/os/dir2/f1", ...);  // read i-node 5 contents
stat("/home/os/dir2/f2", ...);  // read i-node 5 contents
```

4.3.16

Hard Link: Illustration

4.3.17

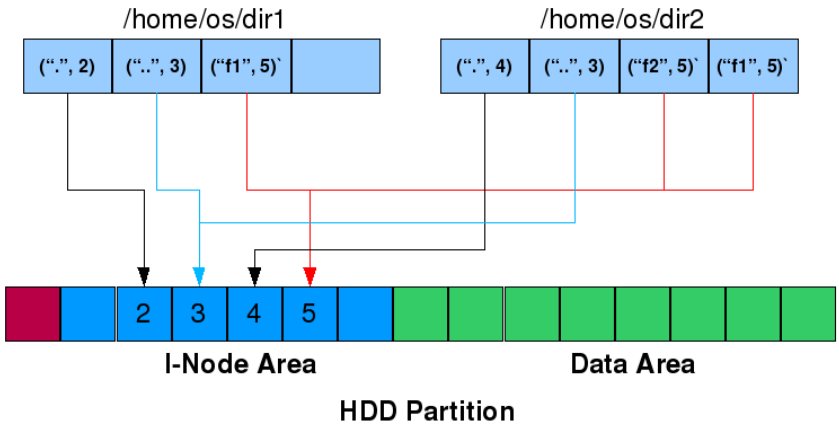


Figure 8: Hard Link Implementation

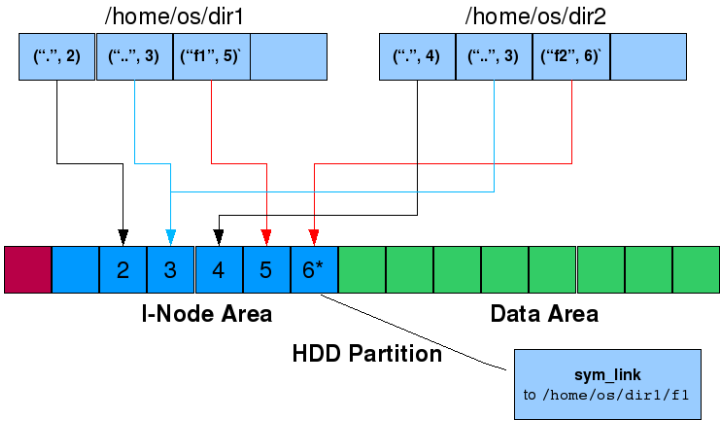


Figure 9: Symbolic Link Implementation

Hard Link: Discussion

- Advantages
  - a file really belong to the two or more directories, when a path is removed the physical file (space) is not removed until all hard links are removed
  - very transparent; there is no difference and distinction between different hard links to the same file
- Disadvantages
  - cannot be established between different partitions

4.3.18

Symbolic Link: Creation and Usage

```
create("/home/os/dir1/f1", 0600);    // only allocate i-node; no data
symlink("/home/os/dir1/f1", "/home/os/dir2/f2"); // hard link
open("/home/os/dir1/f1", ...);  // open file with i-node 5
open("/home/os/dir2/f2", ...);  // open file with i-node 5
stat("/home/os/dir1/f1", ...);  // read i-node 5 contents
stat("/home/os/dir2/f2", ...);  // read i-node 5 contents
lstat("/home/os/dir2/f2", ...);  // read i-node 6 contents
```

4.3.19

Symbolic Link: Illustration

4.3.20

Symbolic Link: Discussion

- Advantages
  - can be created between different partitions
- Disadvantages
  - once the referenced path is removed, the link become invalid

4.3.21

Chapter 5.1

Process Management

General Presentation and Linux System Calls

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Colesa

March 31st, 2021

5.1.1

Purpose and Contents

The purpose of today's lecture

- Presents general aspects related to process management
- Give examples and details about Linux system calls for processes

5.1.2

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, pg. 71 – 100, pg. 132 – 151

5.1.3

Contents

5.1.4

1 General Aspects

Process Definition

- Longman dictionary's definition of **process**
  - *a series of actions that are done in order to achieve a particular result*
- **a program in execution** $\Leftrightarrow$ an user application
  - a **sequential** stream of **execution** in its own **memory address space**
  - including the current **values of CPU's registers** (e.g. IP)
- **OS abstraction for using the computer**
  - composed by all that is needed to run a program: CPU, memory, I/O devices etc.
  - it is a **virtualization concept** $\rightarrow$ **virtualizes an entire system** (computer)
  - $\Rightarrow$ **isolation mechanism**, i.e. isolates one execution (process) by another

5.1.5

Process vs Program

- program's source code $\xrightarrow{compilation}$ program's executable $\xrightarrow{launching}$ process
- **program = static (inactive) entity**
- **process = active entity**
- a process
  - is an activity of some kind
  - is created from a program loaded in memory
  - is allocated system resources (memory, file descriptors, CPU etc.)
  - has input, output, and a state
- the two **parts of a process**
  - **sequential execution**: no concurrency inside a process; everything happens sequentially
  - **process state**: everything that process interacts with (registers, memory, files etc)

5.1.6

The Considered Context

- single-processor systems
- **multiprogramming** and time (processor) sharing, i.e. illusion that **multiple processes** run simultaneously
  - pseudo-parallelism
  - switching CPU among processes
  - scheduling algorithm

5.1.7

Process Creation

- **automatically** (i.e. implicitly) by the OS (**uncommon case**)
  - at system initialization
  - reacting to different events
  - usually run as background processes (vs. foreground processes)
- **at user request** (i.e. explicitly), **by another, existing process** (**common case**)
  - there is a **system call** provided for process creation
  - situations
    - when a process needs help doing some computation
    - when a user action occurs, e.g. interaction with the shell
  - leads to a **process hierarchy** based on the **parent-child** relationship

5.1.8

Process termination

- **voluntarily**, using a special **system call**
  - **normal exit**, i.e. end of program's execution
  - **error detection exit**, like: nonexistent files, insufficient or incorrect input etc.
- **involuntarily**, being forcefully terminated
  - initiated by the system due to a "fatal error", like: illegal instructions, division by zero, segmentation fault etc.
  - initiated by another process (i.e. "killed")

5.1.9

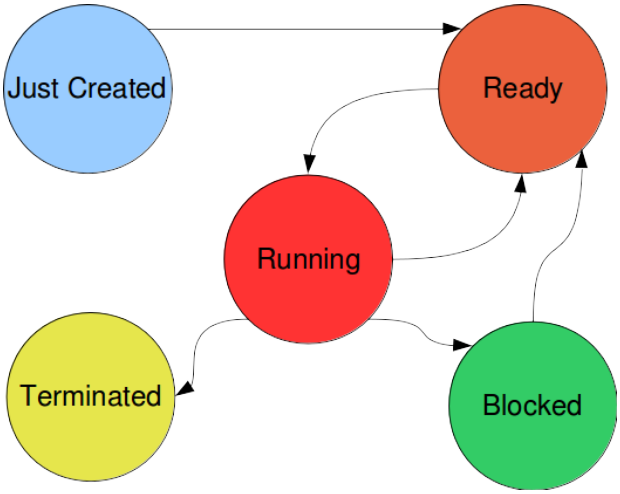


Figure 1: Process States Transition

Process states

- **running**
  - **executed by the CPU**, i.e. using the CPU, at that moment
  - only one process in that state / CPU, actually as many as the number of system's CPUs
- **ready**
  - ready to be executed, but no CPU available
  - so **wait for a CPU to become available**
  - **transparent to the program**
- **blocked**
  - **wait for an event to occur**, a resource to become available
  - **triggered by the application explicitly** through **blocking system calls**
- just created (optional)
  - waiting for some resources to be allocated
- **terminated** (optional)
  - keeping information about the exit state

5.1.10

Process States Transitions

5.1.11

2 Linux Processes

2.1 System Calls

Process Creation: `fork()`

- system call used to **create a new process**
- signature: `fork()`
  - no parameter!
  - **What to do (run) in the new process?**
- child process' **contents is identical** with that of **its parent**
- *still, two distinct and independent processes*
  - remember, **processes provide isolation**
  - $\Rightarrow$ the two processes are scheduled independently on the CPU

- after new process creation ...
  - parent process
    - continues its execution returning from `fork`
    - `fork` returns a positive value (child's PID)
  - child process
    - starts its execution returning from `fork`
    - `fork` returns zero

5.1.12

`fork` Usage Example

```
int x;
static int y;
int *px;

int main(int argc, char **argv)
{
    int pid;

    x = 0;
    px = &x;
    y = 0;

    // up to this point only the parent exists
    // now parent calls fork() to create a new process
    pid = fork();
    if (pid < 0) {
        // error case: no child process created
        perror("Cannot create a new process");
        exit(1);
    }
    // from now on there are two processes: parent and child

    // executed by both processes
    printf("x=%d, px =%p, *px=%d, y=%d\n", x, px, *px, y);
       // parent: x=0, px =0x601050, *px=0, y=0
       // child:  x=0, px =0x601050, *px=0, y=0

    if (pid == 0) { // executed only by the child
        y = 20;
        *px = 200;
    } else {        // executed only by the parent
        y = 10;
        *px = 100;
    }

    // executed by both processes
    printf("x=%d, px=%p, *px=%d, y=%d\n", x, px, *px, y);
       // parent: x=100, px=0x601050, *px=100, y=10
       // child:  x=200, px=0x601050, *px=200, y=20
}
```

5.1.13

`fork`'s Effect Illustration

5.1.14

fork() syscall creates an *independent* child process, which starts as a *copy of its parent!*

5.1.15

Let's practice!

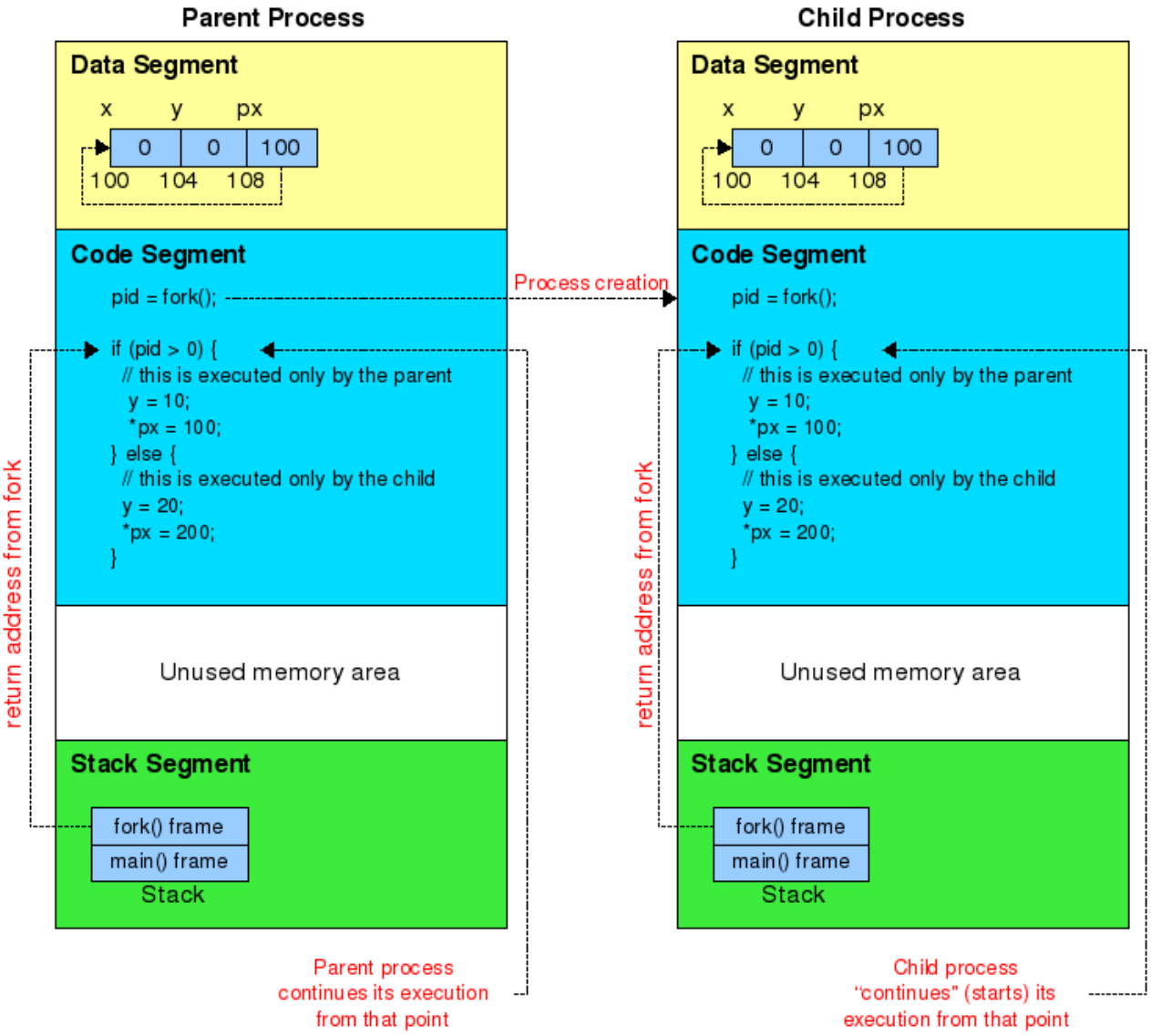
Have you really understood how fork() works?
If you have, try solving the following problems:

1. You are given the following code:

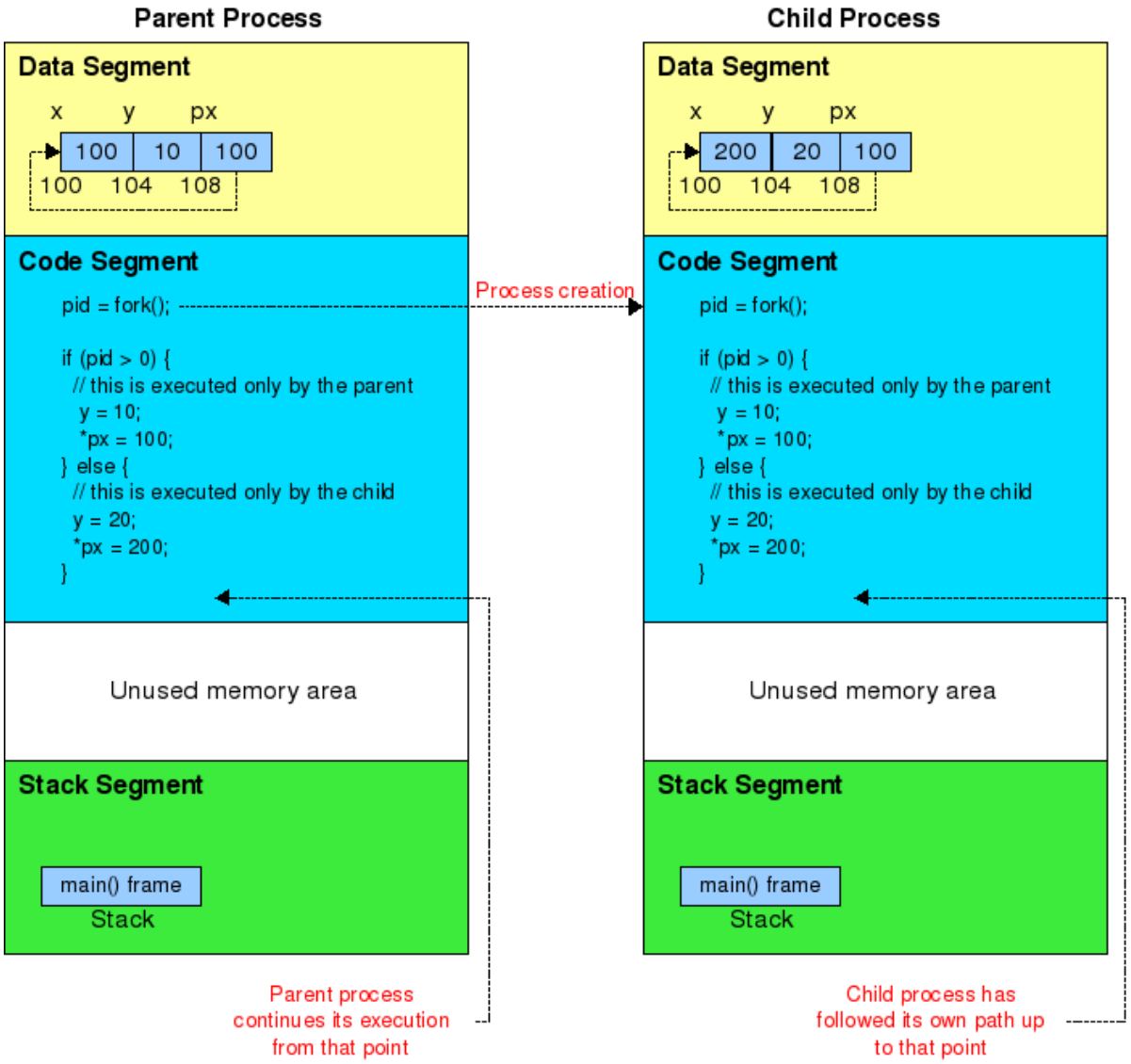
   ```
   fork();
   fork();
   ```

   - How many processes does the following code creates?
   - Draw the resulted process hierarchy.

2. You are given the following code:



Processes Status Immediately After Fork.
The Child Contents Is Identical With That of Its Parent



Processes Status After Some Time.
The Two Processes Evolve Independently

```
for(i=1; i<=100; i++)
    fork();
```

- How many processes does the following code creates?
- Draw the resulted process hierarchy.

5.1.16

Code Execution: exec Family

- system call used to **load a new code into the calling process**
  - replace the calling process' contents, but not its identity
- there are more exec system calls
  - `execl`, `execlp`: with variable number of arguments
  - `execv`, `execvp`: with a fixed number of arguments
- the `exec`'s parameters **similar to a command line**
  - the first argument is **always the path** to the executable file
  - the next argument(s) describe the command line, *starting with command name*

5.1.17

execl and execlp Usage Example

```
// first parameter is the EXPLICIT path to the executable file
execl("/bin/ls", "ls", "-l", NULL);
execl("./myprg.exe", "myprg.exe", "param1", "param2", NULL);

// first parameter is the IMPLICIT path to the executable file
// the path is searched in the directories stored
// in the PATH environment variable
execlp("ls", "ls", "-l", 0);
```

5.1.18

execv and execvp Usage Example

```
char cmdline[10][100]; // equiv. to char *cmdline[];
                       // equiv. to char **cmdline;

// build the command line
strncpy(cmdline[0], "ls", 99);
strncpy(cmdline[1], "-l", 99);
cmdline[2] = NULL;

// call the exec

// first parameter is the EXPLICIT path to the executable file
execv("/bin/ls", cmdline);

// first parameter is the IMPLICIT path to the executable file
// the path is searched in the directories stored
// in the PATH environment variable
execvp("ls", cmdline);
```

5.1.19

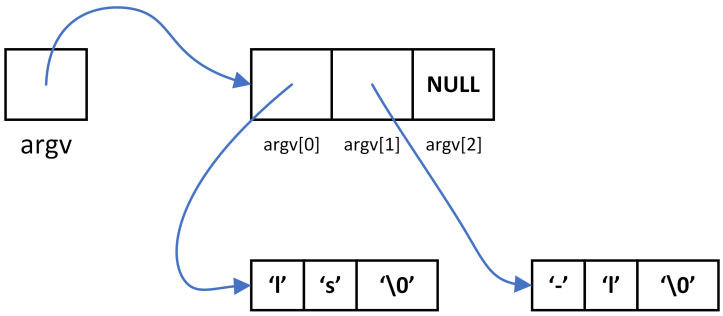
The Structure of the "argv" Array

- command line: `"ls -l"`
- main parameters: `"int main(int argc, char **argv)"`
  - `argc = 2`
  - argv: see below

5.1.20

exec() syscalls loads a new code in the calling process! There is no return from exec() if successfully executed!

5.1.21



Security Aspects About `execlp()` and `execvp()`

- context
  - a **`p` variant** of `exec()` is used
  - only **the name** of executable is given, **not a path**
  - example: "`execlp("ls", "ls", NULL);`"
- ⇒ the corresponding **executable file is searched** in the directories listed in the **`PATH` environment variable**
  - a complete path is built
  - example: "`/bin/ls`"
- security: **application vulnerable** to `PATH` manipulation attack
  - attacker changes the `PATH` to include an attacker writable directory
  - attacker creates a malicious executable searched by the vulnerable application
  - attacker runs the application
  - ⇒ application will run the malicious executable
  - a real problem when the vulnerable application runs with high privileges (e.g. as *root*)

5.1.22

Secure Code Not Trusting the `PATH`

```
// solution 1: set a trusted PATH
setenv("PATH", "/bin:/sbin:/usr/bin:/usr/sbin")
execlp("ls", "ls", "-l", 0);

// solution 2: do not base on PATH
execlp("/bin/ls", "ls", "-l", 0);
```

5.1.23

Relationship Between `fork` and `exec`

- used to
  - create a child process
  - executing something else than its parent
- **Why to have two separated steps instead of just one?**
  - between them the parent "has control" over its child (see standard input and output redirection below)
  - the **parent is released** by the burden of (i.e. time spent) **loading a new code in child**
  - ⇒ **better performance** for the parent

5.1.24

`fork` and `exec` Usage Example

- parent code

```
int main()
{
    pid = fork();

    if (pid > 0) {
        // parent doing something
    } else {
        // child loading and executing a new code
        execl("./child.exe", "child.exe", "p1", "10", 0);
        perror("execl has not succeded");
    }
}
```

- child code

```
int main(int argc, char **argv)
{
    int p;
    for (p=0; p<argc; p++)
        printf("argv[%d]=%s\n", argv[p]);
}
```

5.1.25

Process Termination: `exit`

- system call used to **terminate voluntarily a process**
- terminate the calling process
- specify an exit code
  - 0 (zero) exit code considered successfully termination
  - anything else considered erroneously termination
- exit code is kept until the parent process asks for it
- example
  - `exit(0);`
  - `exit(1);`

5.1.26

Wait For Termination: `wait` and `waitpid`

- system calls used by a process to **wait for the termination of its children**
- return the exit code of the terminated child
- example

```
int child_status;

wait(&child_status);

printf("Child process terminated with code %d\n",
        WEXITSTATUS(status));
```

5.1.27

Relationship Between `wait` and `exit`

- a way to **synchronize two processes**' execution (parent and child)
- a simple way to **communicate between processes** (parent and child)
- when a (parent) process terminates
  - all its children get as their new parent a system process
    - on older systems: the *init* process, having pid = 1
    - on newer systems: a per user *init* process
- when a (child) process terminates before its parent
  - its state is said to be **zombie** and
  - its exit state is maintained by OS until its parent process asks for it or terminates

5.1.28

Let's practice!

You are given the following C code and are required to:

1. Draw the process hierarchy corresponding to the processes created by the code below.
2. Specify the number of times each *printf* is executed, supposing every instruction is executed successfully.

```
1  printf ("[1] Hello world!\n");
2
3  pid = fork();
4
5  printf ("[2] Hello world!\n");
6
7  pid = fork();
8
9  printf ("[3] Hello world!\n");
10
11 if (pid == 0) {
12     execlp("ps", "ps", 0);
13     printf ("[4] Hello world!\n");
14 }
15
16 fork();
17
18 printf ("[5] Hello world!\n");
```

5.1.29

2.2 Examples

Shell Basic Code (Functionality)

```
char **cmdline; // it must be build like argv param of main
while (TRUE) {
    display_prompt_on_screen();
    cmdline = read_cmd_line();

    pid = fork();      // creates a new process
    if (pid < 0) {
        perror("canot creat a new process");
        continue;
    }

    if (pid == 0)
        execvp(cmdline[0], cmdline);
    else
        waitpid(pid, NULL, 0);
}
```

5.1.30

Standard Input Redirection

- command line

  ```
  cat < file.txt
  ```

- STDIN redirection in C program

  ```
  pid = fork();

  if (pid > 0) {
      // parent
  } else {
      // child
      fd = open("file.txt", O_RDONLY);
      dup2(fd, 0);
      close(fd);

      execlp("cat", "cat", 0);
      perror("execl has not succeded");
  }
  ```

5.1.31

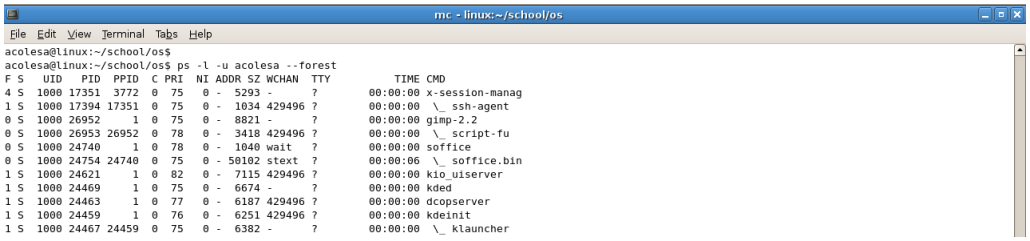
Standard Output Redirection

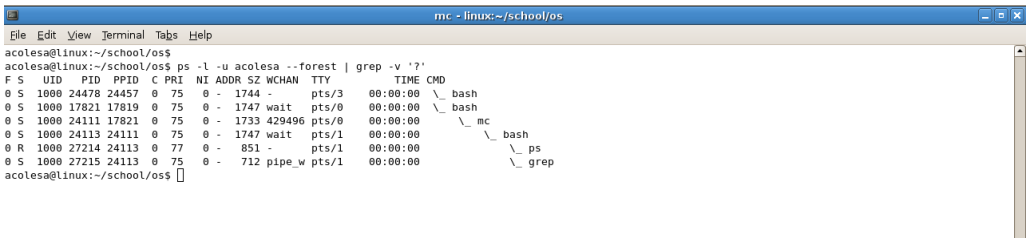
- command line

  ```
  ls > file.txt
  ```

- STDIN redirection in C program

  ```
  pid = fork();

  if (pid > 0) {
      // parent
  } else {
      // child
      fd = creat("file.txt", 0600);
  ```






```
        dup2(fd, 1);
        close(fd);

        execlp("ls", "ls", 0);
        perror("execl has not succeded");
    }
```

5.1.32

2.3 Relates Issues

ps Command

- displays a snapshot of the active processes in the system
- `ps -l -u acolesa --forest`
- `ps -l -u acolesa --forest | grep -v '?'`
- `ps -l -e --forest | head -n 50`

5.1.33

top and htop Commands

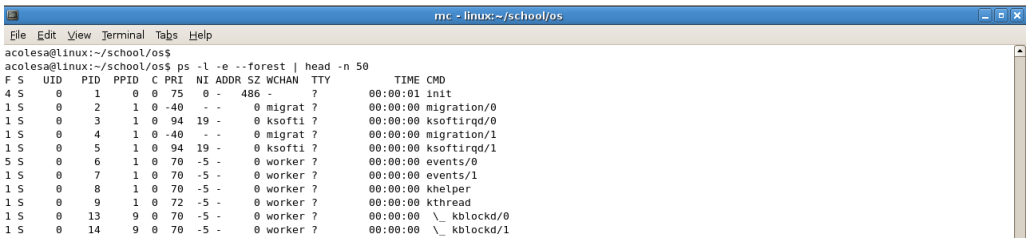
- display a continuously updated list of processes and their on-line scheduling

5.1.34

proc File System

- It is a pseudo file system
- It is mounted in `/proc`
- It is used by the OS to display information about processes
  - each process has a directory named with the process id
  - reading this information is similar to reading any other files and dirs

5.1.35



3 Conclusions

What we talked about

- process definition
- process states and state transitions
  - running, ready, blocked, terminated
- system calls to create and terminate a process
- Linux system calls related to processes
  - fork
  - exec
  - exit
  - wait

5.1.36

Lessons Learned

- process is a virtualization and isolation concept
  - virtualize an entire compute for a program's execution
  - isolate one execution by another
- process states
  - running: the desired one
  - ready: exists due to limited no of CPUs; is transparent to processes
  - blocked: triggered explicitly by a process due to a blocking syscall
- create a new process
  - `fork()` called by the parent to create an identical child process
  - `exec()` called by the child to load new code
- every process terminates with `exit()`

5.1.37

Chapter 5.2

Process Management

Thread Model. Process Scheduling

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

April 7th, 2021

5.2.1

Purpose and Contents

The purpose of this subchapter

- Defines threads and the way they can be used
- Presents scheduling principles and algorithms

5.2.2

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, pg. 71 – 100, pg. 132 – 151

5.2.3

Contents

5.2.4

1 Threads

1.1 Definition of Concepts

Thread Definition

- describes a **sequential**, **independent execution** within a process
  - execution = the memory path followed in the code by the IP register
- **there could be more**
  - simultaneous and independent executions in the same process
  - ⇒ **threads in a process**

5.2.5

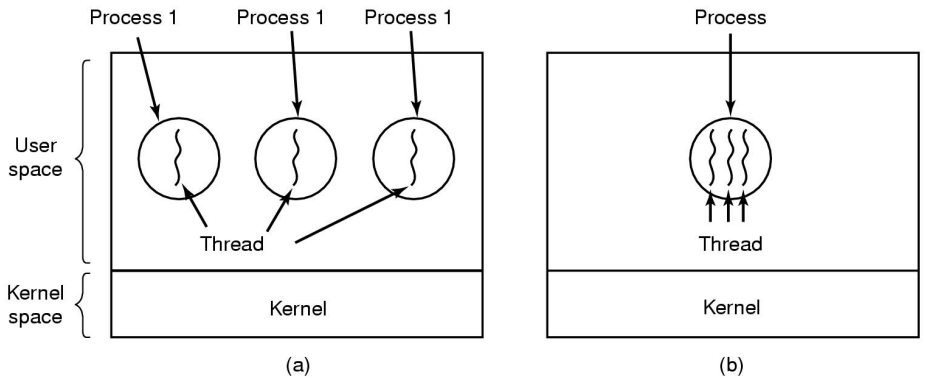


Figure 1: Taken from Tanenbaum

Modeling The Execution

- **process** model
  - models "**an entire computer**" used for executing some user application
  - composed by both
    - process' resources
    - process' execution
  - **isolates resources** of one process by that of others
- **thread** model
  - models **the processor(s)** given to a process
  - makes distinction between the two components of a process
    - the process describes the resources
    - the thread describes the execution
  - if multiple independent executions could be identified in a process
    - there could be more threads of that process
    - one thread $\leftrightarrow$ one execution
  - threads of a process **share resources** of that process
  - threads of a process (normally) **not visible to other processes**

5.2.6

Multiprogramming and Multithreading

- multiprogramming = multiple processes managed simultaneously
- multithreading = multiple threads in the same process managed simultaneously

5.2.7

Thread's Components. Description

- **threads** of the same process
  - **share all the resources of** that **process**: memory, open files etc.
  - $\Rightarrow$ anything (i.e. inside the process) is accessible to all threads (of that process)
- each thread is an independent execution
  - $\Rightarrow$ **each thread has also its own resources**
  - that describe the corresponding independent execution
- thread's specific resources
  - **machine registers** (e.g. the IP register)
  - **the stack**

5.2.8

Thread's Components. Illustration

5.2.9

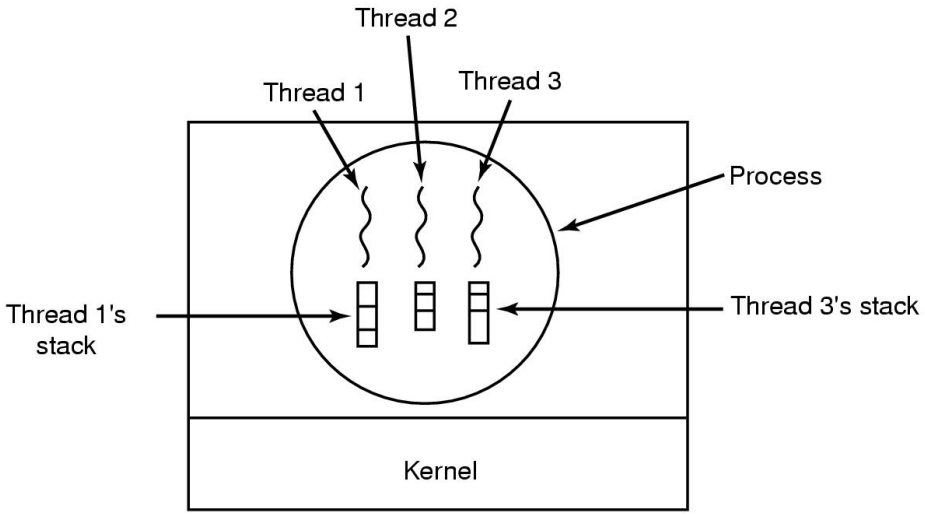


Figure 2: Taken from Tanenbaum

Paraphrase a rule from "Animal Farm" by George Orwell

5.2.10

All threads are equal, but the "main" thread is "*more equal than the others!*"

5.2.11

- **there is no protection between threads of the same process**
  - there is no need, actually
  - created by the application itself to execute code inside the application
- **there is no restriction on them regarding their scheduling**
  - any one could be scheduled any time, depending on the system's scheduling policy
  - we have no control on thread scheduling
  - if needed, **we must synchronize the threads**

5.2.12

- **... but the main one is "*more equal than the others*"**
  - it is the first one created: **executes the *main()* function**
  - it is the ancestor of all other threads, i.e. initiates creation of the other threads
  - if it terminates returning from the "`main()`" function
    - the entire process (and all threads) terminates
    - $\Rightarrow$ it make sense for it to **wait for the termination of all other threads**
  - **though**, if it terminates using the thread termination syscall, the other threads survive
    - see difference between `exit()` and `pthread_exit()` in Linux
    - see a discussion about this at `https://devblogs.microsoft.com/oldnewthing/20100827-00/?p=13023`

5.2.13


ALL ANIMALS ARE EQUAL.
BUT SOME ANIMALS ARE MORE EQUAL THAN OTHERS.

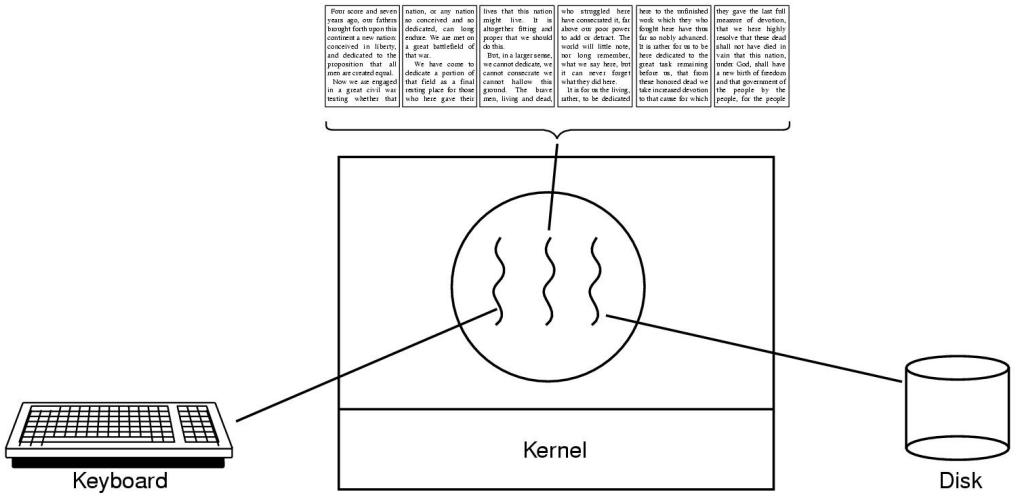


Figure 3: Taken from Tanenbaum

1.2 Thread Usage

When Do Threads Prove Themselves Useful?

- **useful** in applications
  - where **different actions can happen concurrently** (in the same time) → **logical parallelism**
  - and use **different "independent" system resources** (e.g. CPU, HDD, etc.) → **physical parallelism**
- **NOT useful** in applications
  - using only the single CPU (our considered context)
    - though useful, if multiple CPUs are available
  - in which possible concurrent executions need all the time (compete for) the same single shared resource

5.2.14

Examples of Multi-threaded Apps: A Word Processor

5.2.15

Examples of Multi-threaded Apps: A Concurrent Server

5.2.16

Threads vs. Processes

- which are better and for which kind of applications
- **processes** → **isolation**, i.e. totally separated by one another → protection
  - ⇒ use them when isolation and protection needed
  - e.g. tabs in an Internet browser
  - e.g. document tabs in a PDF reader
  - e.g. Web server handling client sessions
- **threads** (light-wight processes) → **resource sharing** (of the same process)
  - ⇒ use them when working on the same process' data is needed
  - e.g. parallel, cooperative computations (see examples above)

5.2.17

Questions (1)

On a computing system with **3 logical processors**, specify the **number of threads** needed to **get the best performance** for:

1. checking if N given numbers are prime or not, using an algorithm that divide the N numbers in equal subsets among the threads.
2. calculating the N-th element of the array generated by the rule: x(n) = x(n-1) + x(n-2), x(0)=0, x(1)=1.

5.2.18

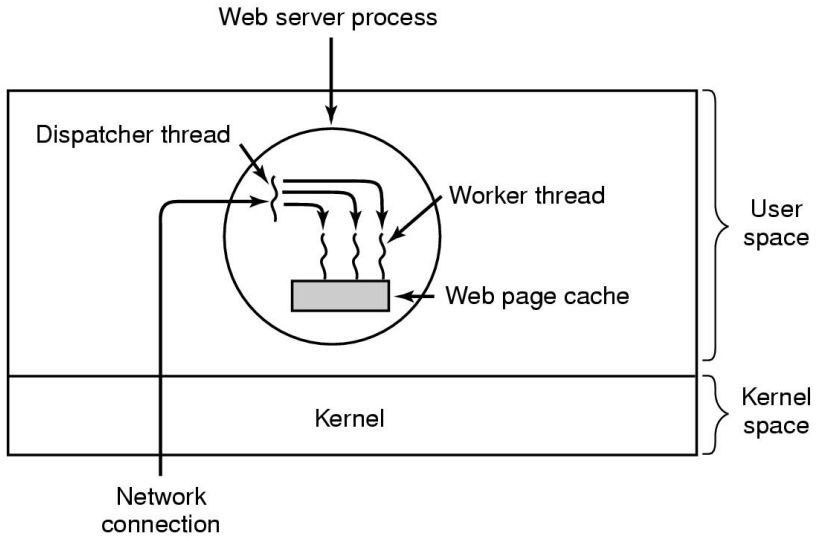


Figure 4: Taken from Tanenbaum

Questions (2)

On a computing system with **1 logical processor**, **a HDD** and **a network card** that can act independently of the system's processors, specify the **number of threads** needed to **get the best performance** for:

1. copying a file from the local HDD to another file on the same HDD;
2. encrypting a file from the local HDD into another file on the same HDD;
3. uploading a file from the local HDD on a remote Web site.

5.2.19

2 Scheduling

2.1 Concepts

Definitions

- **scheduler**: the OS component that **decides**
  - **what process / thread will be run next** and
  - **for how long**
- scheduling policy / algorithm
  - the policy (rules, requirements) / algorithm used by the scheduler
- mainly related to CPU(s)
  - but also consider other system resources
- scheduler's role (motivation)
  - mediates between the more competitors of a limited set of resources
- general classes of processes
  1. compute-bounded (CPU-bounded), i.e. CPU intensive
  2. I/O-bounded, i.e. I/O intensive

5.2.20

Classes of Processes

5.2.21

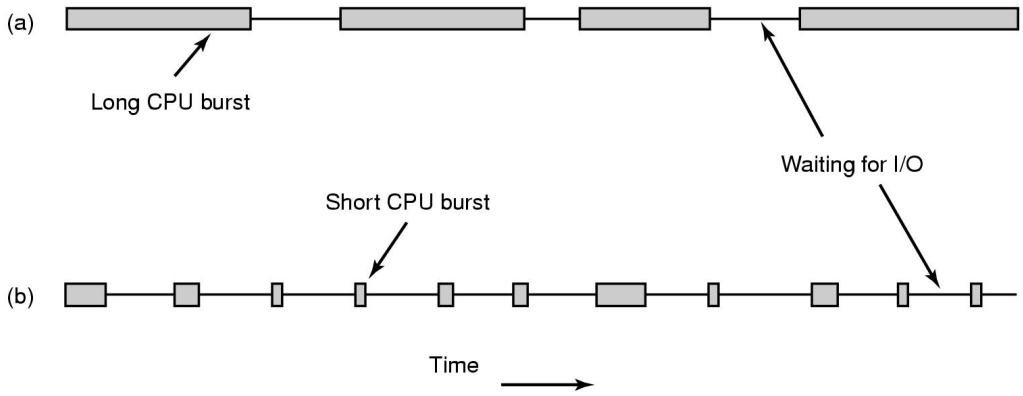


Figure 5: Taken from Tanenbaum

DEMO: CPU-bound vs. I/O-bound Processes/Threads

- create few I/O bound threads, just to see they let the CPU idle a lot (see the percentage they use from CPU)
  - e.g. threads that display very often something on the screen
  - e.g. threads that mostly all the time read something from a file
- create just one CPU-bound thread, just to see that the CPU will be in use all the time
  - e.g. a simple, unrealistic "`while(1);`" thread

5.2.22

Questions (3)

Which threads in the following code are CPU-bound and IO-bound respectively?

```
thread_1 (int n)
{
    int i, x;
    for (x =1, i = 1; i <= n; i++)
        x = x * i;
}

thread_2 ()
{
    char c;
    while (1) {
        read  (0, &c, 1);
        write (1, &c, 1);
    }
}
```

5.2.23

Scheduling Situations

- any time something is changed regarding the CPU's competitors
  - **process creation**
  - **process termination**
  - **process blocking**, e.g. sleeping for a while, waiting for an event etc.
- external events
  - generally, **interrupt occurrences**
  - particularly, the **clock interrupt occurrence**
    - used to implement CPU (time) sharing between processes

5.2.24

Preemptive vs. Non-preemptive Scheduler

- depends on the way the clock interrupt occurrence is used by scheduler
- clock interrupt helps OS keeping track of time passage
- from this perspective a scheduler could be
  - **non-preemptive**
    - lets a process continue its execution until completion/blocking once CPU is given to it
    - i.e. a process could keep CPU if needed
  - **preemptive**
    - suspends a process after a while to switch the CPU to another process
    - i.e. takes the CPU "by force" (yet transparently) from the current process even if it still needs it
    - resumes later the previously suspended process

5.2.25

Categories of Scheduling Algorithms (Examples)

- **batch systems** (processes)
  - do not interact with the users, usually performing long running jobs
  - their users normally expect **reasonable termination time**
  - **fit better non-preemptive scheduling algorithms**
  - or preemptive algorithms with long time quanta
  - reduce the no of process switches $\Rightarrow$ increase overall performance
- **interactive systems** (processes)
  - interact with users
  - their users expect **good response time**
  - **fit only preemptive algorithms**
- **real-time systems** (processes)
  - must perform certain actions in limited time
  - i.e. they must meet their deadlines
  - e.g. react to an external event with a maximum delay
  - **need specific scheduling algorithms** and system characteristics

5.2.26

2.2 Scheduling Algorithms

First-Come First-Served (FCFS)

- it is a non-preemptive algorithm
- simple to understand and implement
- the average waiting time (a.w.t.) depends on the thread order and execution time
  - see Gantt diagram, i.e. graphical illustration of chronological time intervals each thread was scheduled

5.2.27

First-Come First-Served (FCFS). Example 1

- `T1:23, T2:2, T3:3` (`thread : execution time`)
- waiting time for T1 is 0 ms, for T2 is 23 ms, and for T3 is 25 ms
- $a.w.t. = \frac{0+23+25}{3} = 16ms$

5.2.28

First-Come First-Served (FCFS). Example 2

- `T2:2, T3:3, T1:23` (`thread : execution time`)
- waiting time for T1 is 5 ms, for T2 is 0 ms, and for T3 is 2 ms
- $a.w.t. = \frac{5+0+2}{3} = 2.33ms$

5.2.29

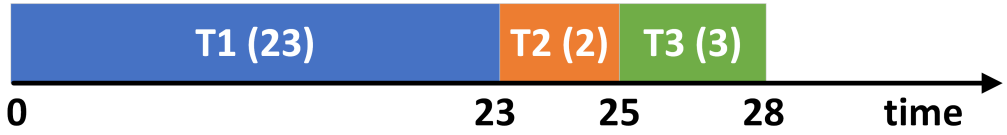


Figure 6: Gantt diagram of process scheduling

Figure 7: Gantt diagram of process scheduling

Shortest Job First (SJF)

- runtime should be known in advance
- is the optimal algorithm when all the ready threads are available simultaneously
- non-preemptive
  - `0:P1:8,1:P2:4, 2:P3:9, 3:P4:5` (`submit time : process : run time`)
  - `a.w.t. = (0 + (8-1) + (17-3) + (12-2))/4 = 7.75`
- preemptive (*shortest remaining time next*)
  - `0:P1:8,1:P2:4, 2:P3:9, 3:P4:5`
  - `a.w.t. = ((10-1)+(1-1)+(17-2)+(5-3))/4 = 6,5`

5.2.30

Round-Robin (RR)

- each thread is assigned a time interval
  - time quantum or slice
- it is a preemptive algorithm
- maintain a FIFO list for ready threads
  - like FCFS, but preemptive, based on time quanta
- the length of the time quantum
  - too short ⇒ many CPU switches ⇒ lower the CPU efficiency, i.e. a lot of CPU is wasted for running the context switching code
  - too large ⇒ longer wait times in ready queue ⇒ poor response time for interactive threads
  - 20–100 msec is a reasonable compromise

5.2.31

Priority Based

- each thread has a priority assigned
  - usually a number that describes the thread "importance" (from some perspective)
  - e.g. running time could be such a number: the smaller the higher thread's priority⇒ SJF
- rule: **the ready thread with the greatest priority is always run**
- give a chance also to smaller priority threads
  - i.e. **avoid starvation**
  - by changing periodically the priority of processes
- priority-based policies
  - fixed priorities

  - dynamically adjusted priorities
- priority classes
  - there could be more processes with the same priority
  - use priority scheduling between classes
  - use another scheduling algorithm with each class, e.g. round-robin

5.2.32

Questions (4)

Let us suppose that an OS' scheduler uses the **Round-Robin** algorithm with **200 ms** time-quantum. Illustrate on a **Gannt diagram** for the time interval **0 – 1 seconds** the executions of the threads whose code is given below, for the following scenarios.

1.
```
// Thread1              // Thread2

while(1){}              while(1){}
```
2.
```
// Thread1              // Thread2

usleep(300000); // 300 ms
while(1) {}             while(1) {}
```

5.2.33

3 Conclusions

What we talked about

- threads
  - describe multiple, independent executions in the same process
  - share all resources of the process they belong to
- multi-threading applications
- scheduling
  - needed when there are more competitors for a limited set of resources (e.g. CPUs)
  - first-come first-served, shortest job first, round-robin, priority-based

5.2.34

Lessons Learned

Multiple threads in a process are **useful** and **effective** only in particular cases:

1. **logical application parallelism** exists
2. needed **physical resources available**

5.2.35

Chapter 6.1

Process and Thread Synchronization

Introduction and Locks

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

April 14, 2021

Purpose and Contents

The purpose of this chapter

- Define and illustrate the context of synchronization
- Present different synchronization mechanisms: locks

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, p. 100 – 132
- A. Downey, *The Little Book of Semaphores*, 2nd Edition, 2008, p. 1 – 106

Contents

1 Synchronization's Problems Overview

1.1 Identify the Problem

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level

  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance ⇒ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

6.1.5

Problem

- uncontrolled
- concurrent executions
- competing
- on shared resources
- could lead to **unpredictable, inconsistent results**
  - i.e. state of the shared resources
- ⇒ **race conditions**

6.1.6

Real-Life Example: "The Milk Buying" Problem (preview)

```
    Time

3.00
3.05
3.10
3.15
3.20
3.25
3.30


    Person A

Look in fridge. No milk.
Go to the store.
Arrive at store.
Buy milk.
Arrive at home with milk.
-
-


    Person B

-
-
Look in fridge. No milk.
Go to the store.
Arrive at store.
Buy milk.
Arrive at home with milk.
```

- no cooperation ⇒ race conditions
- ⇒ **too much milk**
- **buying no milk** could also happen
  - though, not on the given algorithm
  - but in real life with real persons :)

6.1.7

Technical Example: Global Counter

```
// C instruction
count = count + 1;
```

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;

// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0)
    count = count + 1;

// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Machine instructions
load Reg, count
add Reg, 1
store count, Reg
```

```
// Thread 1
1. load Reg1, count
2.
3.
4.
5. add Reg1, 1
6. store count, Reg1

// Thread 2
1.
2. load Reg2, count
3. add Reg2, 1
4. store count, Reg2
5.
6.
```

6.1.8

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
  - **avoiding race conditions**
  - **by cooperation**
- **synchronization** of concurrent threads / processes
  - **wait for the other threads** to reach some point in their execution
  - **establish rules** for accesses to shared resources
    - e.g. **mutual exclusion**: only one at one moment
- **parallelism**
  - synchronization imposed only on **critical regions**
    - i.e. code regions where race conditions appear
  - let the other code run in parallel (if it could) to get a **good performance**

6.1.9

General Synchronization Rules

1. no more than the safe number of threads may be simultaneous inside their critical regions
   - e.g. just one thread for mutual exclusion
2. no thread running outside its critical regions may block other threads
   - e.g. do not forget to let the "key" for the others
3. no thread should have to wait forever to enter its critical region
   - e.g. give chance to everybody
4. no assumptions may be made about speed and number of CPUs and about the system load or scheduling policy
   - e.g. do not use `sleep`-like functions

6.1.10

1.2 Playing Synchronization of ... Persons

"The Milk Buying" Problem

```
    Time

3.00
3.05
3.10
3.15
3.20
3.25
3.30


    Person A

Look in fridge. No milk.
Go to the store.
Arrive at store.
Buy milk.
Arrive at home with milk.
-
-


    Person B

-
-
Look in fridge. No milk.
Go to the store.
Arrive at store.
Buy milk.
Arrive at home with milk.
```

- no cooperation ⇒ **too much milk**
- correctness requirements
  1. never more than one person buys milk
  2. someone buys, if needed

6.1.11

Solution 1: "Lock Variable"

- **idea: let a note *when* leaving buying milk**
  - acts as a lock variable
  - "no note" (lock = 0) ⇒ OK to act (go for it)
  - "there is a note" (lock = 1) ⇒ wait (let the other does his job)
- **generality** (symmetry): every person (thread) executes the same (similar) function (algorithm)

```
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

6.1.12

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

6.1.13

Solution 2: "Pre-Locking"

- **idea: announce the intention (let the note) *before* checking the context**
- **particularity** (asymmetry): each person executes its own function
  - though could be generalized

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}

person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

6.1.14

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}

person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

6.1.15

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, . . . , Person N, Person 1, Person 2, . . . , Person N, . . .
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id)  // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```


Solution 3: "Strict Alternation" (cont.)

- **solution works**
  - mutual exclusion provided (`turn` cannot have but only one value at one moment)
  - no starvation: each thread gets its chance (turn)
- **limitations**: not appropriate for practical situations
  - impose a strict order on the execution
  - the execution speed of all threads is slowed down to the speed of the slowest one
    - in extremis: if one thread dies, all the others are blocked forever
  - generally, **busy waiting is not efficient**

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - $\Rightarrow$ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - $\Rightarrow$ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Practice (1)

- Identify on the code below the critical regions.

```
1  int n = 0;
2
3  th1()                          th2()
4  {                              {
5      int c = 0;                     int c = 0;
6
7      n++;                           c++;
8      c++;                           n++;
9
10     if (c)                         if (c)
11       n--;                           c--;
12
13     c = n;                         c = n;
14     c++;                           c++;
15     n = c;                         n = c;
16 }                              }
```

6.1.19

2 Synchronization Mechanisms

2.1 General Aspects

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

6.1.20

Synchronization Mechanisms' Functionality

- acts as a **checkpoint** for both critical region's
  - entrances
  - exits
- **provided services**
  - **ask for permission to enter** critical region
    - a **thread could be blocked** for a while
  - **announce exit** from critical region
    - a **thread is never blocked** when exiting
- provided by OS, i.e. implemented in OS
  - system calls provided for both types of services

6.1.21

Synchronization Mechanisms' Problem

- becomes itself a concurrently accessed resource
  - **race conditions could occur inside the mediator itself**
- **synchronization mechanisms' functions need atomicity**
  - executed by a thread without interleaving other thread's execution
  - logically, executed like a single, indivisible instruction
  - named **primitive functions (primitives)**
- **OS** must control non-deterministic hardware events
  - **needs hardware support to get atomicity** on normally non-atomic operations

6.1.22

2.2 Hardware Mechanisms That Provide Atomicity

Disable/Enable Interrupts

- the mechanism
  - **disable interrupts** when entering the synchronization mechanisms' functions
  - **(re)enable interrupts** before returning from the synchronization mechanisms' functions
- result: **functions' atomicity**
  - code between interrupt disable and interrupt disable is executed with no interruption
  - i.e. atomically
- **limitations**
  - it works only on uni-processor systems
  - not accessible from user space

6.1.23

Atomic "Read-Modify-Write"/"Test-And-Set-Lock" Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register's value and overwrite it with another register's value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the involved memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

6.1.24

Intel Atomic Instructions

6.1.25

LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Description
F0	LOCK	Asserts LOCK# signal for duration of the accompanying instruction

Description

This instruction causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal insures that the processor has exclusive use of any shared memory while the signal is asserted.

Note that in later Intel Architecture processors (such as the Pentium® Pro processor), locking may occur without the LOCK# signal being asserted. Refer to Intel Architecture Compatibility below.

The LOCK prefix can be prepended only to the following instructions and to those forms of the instructions that use a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. An undefined opcode exception will be generated if the LOCK prefix is used with any other instruction. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

Intel Architecture Compatibility

Beginning with the Pentium® Pro processor, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism insures that the operation is carried out atomically with regards to memory. Refer to Section 7.1.4., *Effects of a LOCK Operation on Internal Processor Caches* in Chapter 7, *Multiple-Processor Management* of the *Intel Architecture Software Developer's Manual, Volume 3*, the for more information on locking of caches.

Operation

```
AssertLOCK#(DurationOfAccompaningInstruction)
```

Flags Affected

None.

Taken from Intel Instructions Manual

2.3 Locks

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock** (**acquire**)
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock** (**release**)
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

6.1.26

Possible Implementations (uni-processor systems)

```
mutex_lock()
{
    disable_interrupts();
    while (value != FREE) {
        insert(crt_thread, w_queue);
        block_current_thread();;
    }
    value = BUSY;
    enable_interrupts();
}

mutex_unlock()
{
    disable_interrupts();
    value = FREE;
    if (! empty(waiting_queue)) {
        th = remove_first(w_queue);
        insert(ready_queue, th);
    }
    enable_interrupts();
}
```

- **interrupts are disabled**, to provide **code atomicity**
  - after critical code is executed, interrupts are enabled back
- "**sleep / wake-up technique**" is used instead of the "*busy waiting*"
  - "**sending to sleep**" (**blocking**) a thread
    - append it to a waiting queue and
    - take the CPU from it
  - "**waking up**" (**unblocking**) a thread
    - remove it from waiting queue
    - append it to the ready queue
    - eventually being given the CPU again

6.1.27

Possible Implementations (multi-processor systems)

```
mutex_lock:
    mov eax, 1              ; prepare the value for acquired lock
    xchg eax, [lock_value]  ; atomic instruction
    cmp eax, 0              ; check if lock was free
    jz lock_taken           ; if it was 0, now it belongs to the calling thread
    jmp mutex_lock          ; otherwise, was taken by other thread and must wait
lock_taken:
    ret

mutex_unlock:
    mov [lock_value], 0
    ret
```

- the atomic "**xchg**" instruction is used to provide **lock's test and set atomicity**
  - disabling interrupts does not work on multi-processor systems
- **busy-waiting** is used in this example
  - it correspond to "spin locks", useful in multi-processor systems
  - sleep / wakeup technique is more difficult to implement and requires the use of a spin lock

6.1.28

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;

// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}

// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

6.1.29

Practice (2)

- Protect the critical regions below with locks.

```
1  int n = 0, m = 0;
2
3  th1()                          th2()
4  {                              {
5      int c = 0;                     int c = 0;
6
7      n++;                           c++;
8      c++;                           n++;
9
10     if (c)                         if (c)
11       m--;                           m--;
12
13     c = n;                         c = n;
14     c++;                           c++;
15     n = c;                         n = c;
16 }                              }
```

6.1.30

3 Conclusions

What we talked about

- race conditions
  - could lead to unpredictable, bad results
- need for synchronization
  - yet do not affect too much the possible parallelism
  - as an OS support
- locks
  - provide mutual exclusion
  - "`acquire()`" and "`release()`" primitives

6.1.31

Lessons Learned

1. race conditions could lead to problems (bugs) difficult to find and investigate
2. synchronization is difficult
3. we need specialized OS provided synchronization mechanisms
4. locks could be too restrictive
5. synchronization could reduce the possible parallelism, though both are needed

6.1.32

Chapter 6.2

Process and Thread Synchronization

Semaphores and Condition Variables

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

April 21, 2021

Purpose and Contents

The purpose of this chapter

- Present different synchronization mechanisms: semaphores and condition variables
- Present some classical synchronization patterns (problems): producer / consumer

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, p. 100 – 132
- A. Downey, *The Little Book of Semaphores*, 2nd Edition, 2008, p. 1 – 106

Contents

1 Semaphores

Characteristics

- **generalization of locks**
  - more threads could be given permission
  - to enter simultaneously in the critical regions protected by the semaphore
- consists of
  - **a value**
    - number of available permissions to pass the checkpoint
    - only positive values are allowed
  - **two primitives**

        - **P** (**sema_down**): blocks the current process if the value is 0, otherwise decrements the value
        - **V** (**sema_up**): increments the value and wakes up a sleeping process
- main difference from locks
    - any thread is allowed to increase a semaphore value (no of permissions), not only those that previously got one
    - $\Rightarrow$ could be used both as a (generalized) lock and an event counter

6.2.5

Possible Implementation (uni-processor systems)

```
P() // sema_down()
{
    disable_interrupts();
    while (value == 0) {
        insert(crt_thread, w_queue);
        block_current_thread();;
    }
    value = value - 1;
    enable_interrupts();
}


V()  // sema_up()
{
    disable_interrupts();
    value = value + 1;
    if (! empty(waiting_queue)) {
        th = remove_first(w_queue);
        insert(ready_queue, th);
    }
    enable_interrupts();
}
```

- **interrupts are disabled**, to provide **code atomicity**
    - after critical code is executed, interrupts are enabled back
- "**sleep / wake-up technique**" is used instead of the "*busy waiting*"
    - "**sending to sleep**" (**blocking**) a thread
        - append it to a waiting queue and
        - take the CPU from it
    - "**waking up**" (**unblocking**) a thread
        - remove it from waiting queue
        - append it to the ready queue
        - eventually being given the CPU again

6.2.6

Usage Example

```
#include <pthread.h>

// restrict the number of "persons" in a room
// i.e number of threas in their critical region
sem_t sem;

void three_in_a_room();

main()
{
    // initialize the semaphore!
    // do that before having threads working with it!
    sem_init(&sem, 1, 3);

    // create the competing threads
    for (int i=0; i < 100; i++)
        pthread_create(&t[i], NULL, three_in_a_room, NULL);

    // wait for all threads' termination
    for (int i=0; i < 100; i++)
        pthread_join(t[i], NULL);

    // remove the semaphore
    sem_destroy(&sem);
}



void three_in_a_room()
{
    // ask for a permission, i.e try decrementing the semaphore
    // the thread will be blocked, if no permission is available,
    // i.e. semaphore's value is 0
    sem_wait(&sem);

        // stay in room (critical region) for a while

    // release the obteained permission
    // i.e. increment the semaphore's value
    // this operation NEVER blocks the thread
    sem_post(&sem);
}
```

6.2.7

Practice (1)

- Using semaphores, write the (pseudo)code for the functions in the code below such that to not allow more than 22 players enter simultaneously on the football field. A player is represented by a thread executing the *football_player()* function.

```
1  void* football_player(void* arg)
2  {
3      int id = (int)arg;
4
5      enter_football_field();
6
7      // play football for a while
8      while (!match_end & !player_changed[id]);
9
10     exit_the_footbal_field();
11 }
12
13 main()
14 {
15     // create the competing threads
16     for (int i=0; i < 32; i++)
17         pthread_create(&t[i], NULL, football_player, (void*)i));
18
19     // wait for all threads' termination
20     for (int i=0; i < 32; i++)
21         pthread_join(t[i], NULL);
22 }
```

6.2.8

Practice (2)

- Using semaphores, write the (pseudo)code that limits to 5 the number of threads executing simultaneously the critical section in the function below.

```
1  #define FREE 1
2  #define BUSY 0
3  #define MAX_TH 5
4
5  int available_unit[MAX_TH] = {FREE, FREE, FREE, FREE, FREE};
6  int unit[MAX_TH] = {0, 0, 0, 0, 0};
7
8  void limited_area()
9  {
10     int pos = -1;
11
12     for (pos = 0; pos < MAX_TH; pos++)
13         if (available_unit[pos] == FREE) {
14             available_unit[pos] = BUSY;
15             break;
16         }
17
18     unit[pos]++;
19
20     available_unit[pos] = FREE;
21 }
```

6.2.9

2 The "Producer/Consumer" Problem

Problem Description

- two types of processes
  - *producers*: produce messages
  - *consumers*: consume messages
- use a bounded buffer
- waiting conditions
  - for producer: "buffer is full"
  - for consumer: "buffer is empty"
- wakeup conditions
  - for producer: "there is space in buffer"
  - for consumer: "there are messages in buffer"

6.2.10

Basic Solution With Race Conditions

```
// Global variables
const N = 100;         // number of slots in buffer
int count = 0;         // number of messages in the buffer


void producer(int item)
{
    // wait for available space
    while (count == N) {} // full buffer

    insert_item(item);

    // wakeup a consumer
    count = count + 1;
}


int consumer(int *item)
{
    // wait for available message
    while (count == 0) {}  // empty buffer

    *item = remove_item();

    // wakeup a producer
    count = count - 1;
}
```

- **race conditions**
  - uncontrolled concurrent access to variable "`count`"
  - same message could be consumed more times
  - messages could be overwritten
- **busy waiting**
  - suggests us that **synchronization is needed**
  - **should be avoided** in practice
  - **replace it** with a sleep / wakeup **synchronization mechanism**

6.2.11

Producers/Consumers Problem's Implementation With Semaphores

```
// Global variables and synchronization mechanims
const N = 100;                  // number of slots in buffer
Semaphore mutex = 1;             // provides mutual exclusion to buffer
Semaphore can_produce_msg = N;   // controls producers' access to buffer
Semaphore can_consume_msg = 0;   // controls consumers' access to buffer


void producer(int item)
{
    // check if buffer is full
    can_produce_msg.P();

    // gets mutual exclusion to buffer
    mutex.P();

    insert_item(item);

    // releases the buffer
    mutex.V();

    // new permission for consumers
    can_consume_msg.V();
}


int consumer(int *item)
{
    // check if buffer is empty
    can_consume_msg.P();

    // gets mutual exclusion to buffer
    mutex.P();

    *item = remove_item();

    // releases the buffer
    mutex.V();

    // new permission for producers
    can_produce_msg.V();
}
```

6.2.12

Remarks On Producers/Consumers Implementation With Semaphores

- there are two ways of using semaphores
  - as a (generalized) lock to provide limited access (mutual exclusion in the example with `mutex` semaphore)
  - as a condition checking and event counter (semaphores `can_consume_msg` and `can_produce_msg`)
- the order of getting the semaphores is important and it can lead to deadlock
  - see the solution with `mutex` semaphore get first

6.2.13

Deadlock In The Producers/Consumers Problem's Implementation With Semaphores

```
// Global variables and synchronization mechanims
const N = 100;                  // number of slots in buffer
Semaphore mutex(1);             // provides mutual exclusion to buffer
Semaphore can_produce_msg(N);   // controls producers' access to buffer
Semaphore can_consume_msg(0);   // controls consumers' access to buffer


void producer(int item)
{
    // gets mutual exclusion to buffer
    mutex.P();

    // check if buffer is full
    can_produce_msg.P();

    insert_item(item);

    // new permission for consumers
    can_consume_msg.V();

    // releases the buffer
    mutex.V();
}


int consumer(int *item)
{
    // gets mutual exclusion to buffer
    mutex.P();

    // check if buffer is empty
    can_consume_msg.P();

    *item = insert_item();

    // new permission for producers
    can_produce_msg.V();

    // releases the buffer
    mutex.V();
}
```


3 Condition Variables

Condition Variables: Context and Problem

- **used in a mutual exclusion context**
- **a thread** having a lock **must wait for a specific condition** to be fulfilled before going further
  - waited condition reflect the state of the shared (competing) resource
  - lock must be taken before testing / changing the condition
- **should not wait (sleep) keeping the lock**
  - could block the other threads that could change condition to true
- $\Rightarrow$ waiting thread should **release the lock while waiting** (sleeping)
- $\Rightarrow$ another thread can take the lock in the meantime and change condition
- that thread **wakes up the sleeping (waiting) thread**

Condition Variables: Possible Deadlock Scenario

```
// Thread waiting
// for a condition
// to be fulfilled
thread_1()
{
  mutex.lock();

  while (cond == FALSE) {
    // e.g. (no_msg == 0)
    // for a consumer
    sleep(1);
  }

  mutex.unlock();
}

// Thread changing
// the waited condition
// to TRUE
thread_2()
{
```

```
  mutex.lock();

  cond = TRUE;
  // e.g. no_msg++
  // for a producer



  mutex.unlock();
}
```

6.2.16

Condition Variables: Functionality

- what is it?
  - **a specialized waiting mechanism**
  - transparently release the lock while blocking the lock holder
  - at wakeup transparently reacquire the lock before resuming the waiting thread
- consists of
  - a waiting queue
  - two (three) primitives
    - **wait**: the calling process is inserted in the waiting queue and suspended
    - **signal**: one sleeping process is removed from the waiting queue and woken up
    - **broadcast**: all sleeping processes are woken up

6.2.17

Condition Variables. Basic Pattern Usage

```
// Thread waiting for
// an event (condition)
// to occur
thread_1()
{
  mutex.lock();

  while (! cond)
    c.wait(mutex);

  mutex.unlock();
}

// Thread generating
// the event
//
thread_2()
{
  mutex.lock();

  cond = TRUE;
  c.signal();

  mutex.unlock();
}
```

6.2.18

Condition Variables: Implementation

```
wait(Lock mutex)
{
  disable_interrupts();

  mutex.unlock();

  insert(crt_th, w_queue);
  block_current_thread();

  mutex.lock();

  enable_interrupts();
}
```

```
signal()
{
  disable_interrupts();


  if (! empty(waiting_queue)) {
    th = remove_first(w_queue);
    insert(ready_queue, th);
  }


  enable_interrupts();
}
```

6.2.19

Condition Variables. General Pattern Usage (Var 1)

```
// Synchronization mechanims: one condition variable
Lock mutex;
Condition c;


// Thread 1's Function
thread_1()
{
    // critical region entrace
    mutex.lock();
    while (! cond_1)
        c.wait(mutex);
    mutex.unlock();

    // thread T1 in its critical region
    ....

    // critical region exit
    mutex.lock();
    cond_2 = TRUE;
    c.broadcast();
    mutex.unlock();
}


// Thread 2's Function
thread_2()
{
    // critical region entrace
    mutex.lock();
    while (! cond_2)
        c.wait(mutex);
    mutex.unlock();

    // thread T2 in its critical region
    ...

    // critical region exit
    mutex.lock();
    cond_1 = TRUE;
    c.broadcast();
    mutex.unlock();
}
```

6.2.20

Condition Variables. General Pattern Usage (Var 2)

```
// Synchronization mechanims: two condition variables
Lock mutex;
Condition c1, c2;


// Thread 1's Function
thread_1()
{
    // critical region entrace
    mutex.lock();
    while (! cond_1)
        c1.wait(mutex);
    mutex.unlock();

    // thread T1 in its critical region
    ....

    // critical region exit
    mutex.lock();
    cond_2 = TRUE;
    c2.signal(); // c2.broadcast();
    mutex.unlock();
}


// Thread 2's Function
thread_2()
{
    // critical region entrace
    mutex.lock();
    while (! cond_2)
        c2.wait(mutex);
    mutex.unlock();

    // thread T2 in its critical region
    ...

    // critical region exit
    mutex.lock();
    cond_1 = TRUE;
    c1.signal();  // c1.broadcast();
    mutex.unlock();
}
```

6.2.21

Condition Variables. Multiple Conditions (Var 1)

```
// Synchronization mechanims: multiple conditions & one condition variable
Lock mutex;
Condition c;


// Thread 1's Function
thread_1()
{
    // critical region entrace
    mutex.lock();
    while (! cond_1 || ! cond_2)
        c.wait(mutex);
    mutex.unlock();

    // T1's critical region
    ....

    // critical region exit
    mutex.lock();
    .....
    mutex.unlock();
}


// Thread 2's Function
thread_2()
{
    ....
    ....
    // critical region exit
    mutex.lock();
    switch (x) {
      case 1:
        cond_1 = TRUE;
        c.broadcast(); // c.signal() ?
        break;
      case 2:
        cond_2 = TRUE;
        c.broadcast(); // c.signal() ?
        break;
    }
    mutex.unlock();
}
```

6.2.22

Condition Variables. Multiple Conditions (Var 2 - BAD)

```
// Synchronization mechanims: multiple conditions & multiple condition variables & one lock
Lock mutex;
Condition c1, c2;


// Thread 1's Function
thread_1()
{
    // critical region entrace
    mutex.lock();
    while (! cond_1)
        c1.wait(mutex);

    while (! cond_2)
        c2.wait(mutex);
    mutex.unlock();

    // T1's critical region
    ....

    // critical region exit
    mutex.lock();
    .....
    mutex.unlock();
}


// Thread 2's Function
thread_2()
{
    ....
    ....
    // critical region exit
    mutex.lock();
    switch (x) {
      case 1:
        cond_1 = TRUE;
        c1.signal(); // c1.broadcast();
        break;
      case 2:
        cond_2 = TRUE;
        c2.signal(); // c2.broadcast();
        break;
    }
    mutex.unlock();
}
```

6.2.23

Condition Variables. Multiple Conditions (Var 2 - OK)

```
// Synchronization mechanims: multiple conditions & multiple condition variables & multiple locks
Lock mutex1, mutex2;
Condition c1, c2;


// Thread 1's Function
thread_1()
{
    // critical region entrace
    mutex1.lock();
    while (! cond_1)
        c1.wait(mutex1);

    mutex2.lock();
    while (! cond_2)
```

```
        c2.wait(mutex2);
    mutex2.unlock();
    mutex1.unlock();

    // T1's critical region
    ....

    // critical region exit
    .....
}


// Thread 2's Function
thread_2()
{
    ....
    // critical region exit

    switch (x) {
    case 1:
        mutex1.lock();
        cond_1 = TRUE;
        c1.signal(); // c1.broadcast();
        mutex1.unlock();
        break;
    case 2:
        mutex2.lock();
        cond_2 = TRUE;
        c2.signal(); // c2.broadcast();
        mutex2.unlock();
        break;
    }
}
```

6.2.24

Condition Variables: Comparison With Semaphores

- *wait* always suspends the calling process
  - while *P* does this only in case the semaphore's value is zero
- a `signal` can be lost (i.e. not seen)
  - while a *V* is not, because it increments the semaphore's value
- safe to use inside a mutual exclusion area
  - while semaphore could lead to deadlock

6.2.25

Rules to Remember About Using Condition Variables

1. **ALWAYS use them** (`wait` and `signal`) **inside a mutual exclusion area** (protected by a **lock**)!
2. the lock does (normally) not protect the shared resource
   - is would simply provide mutual exclusion, but maybe not needed
   - does not support deadlock-free waiting
3. the lock protects the entrance in and exit from critical region
   - where some condition checks or changes are done
   - in a mutual exclusion manner
4. it **could be needed to recheck the condition** after returning from `wait`
   - use *while* instead of *if*
5. **NEVER use `wait` without checking first a condition**!
   - it is not sure the thread must wait (we do not control thread scheduling)
   - `signal` has no history, so can be lost

6.2.26

Rules to Remember About Using Condition Variables (review)

1. **eyes** for seeing
2. **ears** for hearing
3. **mind** for thinking (understanding, learning, remembering)
4. see https://www.youtube.com/watch?v=RZxAc3Grkck
5. review previous slide and make use of the "elements" mentioned above

6.2.27

Producers/Consumers Problem's Implementation With Locks and Condition Variables

```
// Global variables and synchronization mechanisms
const N = 100;        // number of slots in buffer
int count = 0;        // number of messages in buffer
Lock mutex;           // provides mutual exclusion to buffer
Condition producers;  // controls producers' access to buffer
Condition consumers;  // controls consumers' access to buffer


void producer(int item)
{
    // gets mutual exclusion to buffer
    mutex.lock();

    // check if buffer is full
    while (count == N)
      producers.wait(mutex);

    insert_item(item);
    count = count + 1;

    // wakes up a consumer
    consumers.signal();

    // releases the buffer
    mutex.unlock();
}


int consumer(int *item)
{
    // gets mutual exclusion to buffer
    mutex.lock();

    // check if buffer is empty
    while (count == 0)
      consumers.wait(mutex);

    *item = remove_item();
    count = count - 1;

    // wakes up a producer
    producers.signal();

    // releases the buffer
    mutex.unlock();
}
```


Practice (3)

- Using locks and condition variables, write the (pseudo)code for the functions in the code below such that to not allow more than 22 players enter simultaneously on the football field. A player is represented by a thread executing the *football_player()* function.

```
void* football_player(void* arg)
{
    int id = (int)arg;

    enter_football_field();

    // play football for a while
    while (!match_end & !player_changed[id]);

    exit_the_footbal_field();
}

main()
{
    // create the competing threads
    for (int i=0; i < 32; i++)
        pthread_create(&t[i], NULL, football_player, (void*)i));

    // wait for all threads' termination
    for (int i=0; i < 32; i++)
        pthread_join(t[i], NULL);
}
```


4 Conclusions

What we talked about

- semaphores
  - a generalization of locks ⇒ could allow more threads pass the barrier
  - important to initialize the semaphore
  - "`P()`" and "`V()`" primitives
  - could also be used as an event counter, usually initialized with 0
- condition variables
  - `wait`: provide a specialized way to wait for a condition to be fulfilled
  - `signal`: provide a way to signal that a condition is fulfilled
- producer-consumer synchronization pattern

6.2.30

Lessons Learned

1. locks could be too restrictive
2. semaphores are more flexible, though must be used with care to avoid deadlock
3. condition variables must be used with a lock, i.e. inside a mutual exclusion area

6.2.31

Chapter 6.3

Process and Thread Synchronization

Classical Synchronization Patterns

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Colesa

April 28, 2021

6.3.1

Purpose and Contents

The purpose of this chapter

- Present some classical synchronization patterns (problems): readers / writers, barrier, philosophers etc.

6.3.2

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, p. 100 – 132
- A. Downey, *The Little Book of Semaphores*, 2nd Edition, 2016, p. 1 – 115

6.3.3

Contents

6.3.4

1 Implementing Synchronization Mechanisms With Other Synchronization Mechanisms

Semaphores Using Locks And Condition Variables

```
// Internal Variables
int value = initValue;
Lock mutex;
Condition permission;


// decrement the semaphore by 1
P()
{
    mutex.lock();

    while (value == 0)
        permission.wait(mutex);

    value--;

    mutex.unlock();
}
```

```
// increment the semaphore by 1
V()
{
    mutex.lock();

    value++;

    permission.signal();

    mutex.unlock();
}


// decrement the semaphore by N
P(int n)
{
    mutex.lock();

    while (value < n)
        permission.wait(mutex);

    value -= n;

    mutex.unlock();
}


// increment the semaphore by N
V(int n)
{
    mutex.lock();

    value += n;

    permission.broadcast();

    mutex.unlock();
}
```

6.3.5

Locks Using Semaphores

```
// internal variables
Semaphore s(1);
int lockHolder = -1;


// Acquire the lock
lock()
{
    s.P();
    lockHolder = gettid();

}


// Release the lock
unlock()
{
    if (lockHolder = gettid()) {
        lockHolder = -1;
        s.V();
    }
}
```

6.3.6

Condition Variables Using Semaphores

- implementation

```
// internal variables
List<Semaphore> semList;


// wait until signaled
// releasing the lock
wait(Lock *mutex)
{
    // creates a new 0 sem
    Semaphore s(0);

    // add teh sem to waiting list
    semList.add(s);

    // release the lock
    mutex->unlock();

    // go to sleep
    s.P();

    // re-acquire the lock
    mutex->lock();
}


// wake up a waiting thread
// supposes to be called when
//   the lock is held!!!
signal()
{
    Semaphore s;

    if (! semList.isEmpty()) {
        s = semList.removeFirst();
        s.V();
    }
}
```

.

- usage

```
Lock mutex;
Condition c;
int ok = 0;


// Thread T1
mutex.lock();
while (!ok)
    c.wait(&mutex);
mutex.unlock();


// Thread T2
mutex.lock();
ok = 1;
c.signal();
mutex.unlock();
```

6.3.7

Practice (1)

You are given the three functions below, executed by three different threads. You are required to use

1. semaphores
2. locks and condition variables

to make sure that the string displayed on the screen is always "**1 + 2 + 3 + 4 = 10**", no matter how the threads are scheduled.

```
thread_function_1()
{
  printf("1 + ");
  printf("3 + ");
}
```

```
thread_function_2()
{
  printf("2 + ");
  printf("4 =");
}
```

```
thread_function_3()
{
  printf("10\n");

}
```

6.3.8

2 Classical Synchronization Patterns

Strict Alternation Using Semaphores

```
// Semaphores initialization
Semaphores s[N];

// consider initial turn is for thread with ID = 0
s[0] = 1;

// it is not the turn of other threads
for (i=1; i<N; i++)
    s[i] = 0;


// Threads' function
thread_function(int th_id)
{
    s[th_id].P();

    printf("It is my turn now! Nobody can take it from me.\n");

    // ... until I give it voluntarily to next
    s[(th_id+1)%N].V();
}
```

6.3.9

Strict Alternation Using Locks and Condition Variables

```
// Global variables
Lock mutex;
Condition c[N];
int turn = 0;


// Threads' function
thread_function(int th_id)
{
    // ENTRY in critical region
    mutex.lock();

    while (turn != th_id)
        c[thId].wait(mutex);

    mutex.unlock();

    // INSIDE the critical region
    printf("It is my turn now! Nobody can take it from me.\n");

    // EXIT from critical region
    mutex.lock();

    // ... until I give it voluntarily to next
    turn = (turn + 1) % N;
    c[turn].signal();

    mutex.unlock();
}
```

6.3.10

"Unfair" Rendezvous

```
// Global variables
Semaphore is_friend_1 = 0;
Semaphore is_friend_2 = 0;;


// function of friend_1 threads
friend_1()
{
    // announce its own presence
    is_friend_1.V();

    // check for its partner's presence
    is_friend_2.P();
}


// function of friend_2 threads
friend_2()
{
    // announce its own presence
    is_friend_2.V();

    // check for its partner's presence
    is_friend_1.P();
}
```

6.3.11

"Fair" Rendezvous

```
// Global variables
Semaphore is_friend_1 = 0;
Semaphore access_to_meeting_point_1 = 1;

Semaphore is_friend_2 = 0;
Semaphore access_to_meeting_point_2 = 1;


// function of friend_1 threads
friend_1()
{
    // get exclusive access to the meeting
    access_to_meeting_point_1.P();

    // announce its own presence
    is_friend_1.V();

    // check for its partner's presence
    is_friend_2.P();

    // let another of the same type enter
    access_to_meeting_point_1.V();
}


// function of friend_2 threads
friend_2()
{
    // get exclusive access to the meeting
    access_to_meeting_point_2.P();

    // announce its own presence
    is_friend_2.V();

    // check for its partner's presence
    is_friend_1.P();

    // let another of the same type enter
    access_to_meeting_point_2.V();
}
```

6.3.12

"Deadlocked" Rendezvous

```
// Global variables
Semaphore is_friend_1 = 0;
Semaphore is_friend_2 = 0;


// function of friend_1 threads
friend_1()
{
    // check for its partner's presence
    is_friend_2.P();

    // announce its own presence
    is_friend_1.V();
}


// function of friend_2 threads
friend_2()
{
    // check for its partner's presence
    is_friend_1.P();

    // announce its own presence
    is_friend_2.V();
}
```

6.3.13

One-Time Usage Barrier (Meeting of N Processes)

```
// Global variables
Semaphore mutex = 1;
Semaphore barrier = 0;
int count = 0;      // count how many threads arrived at meeting point
const int N = 10;   // barrier initialization (app's specific)
```

```
// function called by threads to wait for meeting
meeting_point()
{
    // get exclusive access
    // when updating and checking count
    mutex.P();

    count++;

    if (count == N)     // if the last one
        barrier.V();    // open the barrier

    // let the others enter the checking point
    mutex.V();

    barrier.P();        // here is the barrier
    barrier.V();        // let it open for the next
}
```

6.3.14

Reusable Barrier (By The Same Set of N Processes)

```
// Global variables
Semaphore mutex = 1;
Semaphore barrier1 = 0; // for stoping thread at entrence of meeting point
Semaphore barrier2 = 1; // for stopping threads at exit from meeting point
int count = 0;          // count how many threads
                        // arrived and inside the meeting point
const int N = 10;       // barrier initialization (app's specific)
                        // see POSIX pthread_barrier_init() function


// function called by threads to wait for meeting
meeting_point() // see POSIX pthread_barrier_wait() function
{
  // get exclusive access
  // at entrace checkpoint
  mutex.P();

  count++;

  if (count == N) { // if last one entering
      barrier2.P(); // close exiting barrier
      barrier1.V(); // open entering barrier
  }

  // let the others enter the IN checkpoint
  mutex.V();

  barrier1.P();  // the entering barrier
  barrier1.V();  // let it open for the next

  .....


  ......

  // get exclusive access
  // at exit checkpoint
  mutex.P();
  count--;

  if (count == 0) { // if last one exiting
      barrier1.P(); // close entering barrier
      barrier2.V(); // open exiting barrier
  }

  // let the others enter the OUT checkpoint
  mutex.V();

  barrier2.P();  // the exiting barrier
  barrier2.V();  // let it open for the next
}
```

6.3.15

Readers/Writers Problem. Description

- models accesses to a shared database (DB)
- there are two types of threads
  - *readers*: just read, do not modify the shared resource
  - *writers*: modify the shared resource
- synchronization (access) rules are
  - multiple readers allowed simultaneously, but not in the same time with a writer
  - when a writer accesses the shared resource, no other process can accesses it

6.3.16

Readers/Writers Problem. Implementation With Locks and Condition Variables

```
// Global variables
int WR = 0; // waiting readers
int AR = 0; // active readers on the DB; AR >= 0
int WW = 0; // waiting writers
int AW = 0; // active writers on the DB, 0 <= WR <=1

Lock mutex;
Condition okToRead, okToWrite;
```

```
// Reader's function
Reader()
{
    mutex.Acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.WAIT(&mutex);
        WR--;
    }
    AR++;
    mutex.Release();

    //  -----------> read DB

    mutex.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.SIGNAL();
    mutex.Release();



}


// Writer's function
// writers get preference over readers
Writer()
{
    mutex.Acquire();
    while (AR + AW > 0) {
        WW++;
        okToWrite.WAIT(&mutex);
        WW--;
    }
    AW++;
    mutex.Release();

    //  -----------> write DB

    mutex.Acquire();
    AW--;
    if (WW > 0) // favor writers
      okToWrite.SIGNAL();
    else
      if (WR > 0)
        okToRead.BROADCAST();
    mutex.Release();
}
```


Readers/Writers Problem. Implementation With Semaphores (1)

```
Semaphore permissions = MAX_READERS;


// Reader's function
// readers get preference over writers
Reader()
{
    permissions.P(1);

    //  -----------> read DB

    permissions.V(1);
}


// Writer's function
Writer()
{
    permissions.P(MAX_READERS);

    //  -----------> write DB

    permissions.V(MAX_READERS);
}
```


Readers/Writers Problem. Implementation With Semaphores (2)

```
Semaphore permissions = 1;
Semaphore mutex = 1;
int readers = 0; // numer of readers in critical region


// Readres' function
// readers get preference over writers
Reader()
{
    mutex.P();
    readers++;

    if (readers == 1)
      permissions.P();
    mutex.V();

    //  -----------> read DB

    mutex.P();
    readers--;

    if (readers == 0)
      permissions.V();
    mutex.V();
}


// Writers' function
Writer()
{

    permissions.P();
```

```
    //  ----------> write DB

    permissions.V();

}
```

6.3.19

Readers/Writers Problem. A Particular Case: Single Favored Reader

```
Semaphore permissions(1);
Semaphore writerBarrier(1);


// Single reader's function
Reader()
{


    permissions.P();

    //  ----------> read DB

    permissions.V();


    }


// Writers' function
Writer()
{
    writersBarrier();

    permissions.P();

    //  ----------> write DB

    permissions.V();

    writersBarrier.V();
}
```

6.3.20

Dining Philosophers. Description

- proposed by Dijkstra in 1965
- five philosophers are seated around a circular table
- each philosopher has a plate with spaghetti
- between each pair of plates is one fork
- a philosopher needs two forks to eat
- a philosopher eats and thinks
- only one philosopher can hold a fork at a time
- **deadlock** and **starvation** should be avoided

6.3.21

Dining Philosophers. Implementation with Deadlock

```
// Global variables
const int N = 5;

// Utility functions
int right(id) { return id; }
int left(id) { return (id+1) % N; }

// Synchronization mechanisms
Semaphores forks[N];
for (i=0; i<N; i++)
    forks[i] = 1;


void philosopher(int id)
{
    while (TRUE) {
        think();
        take_forks(id);
        eat();
        put_forks(id);
    }
}


void take_forks(int id)
{
    forks[right(id)].P();
    forks[left(id)].P();
}

void put_forks(int id)
{
    forks[right(id)].V();
    forks[left(id)].V();
}
```

6.3.22

Dining Philosophers. Solution 1

```
// Global variables
const int N = 5;

// Utility functions
int right(id) { return id; }
int left(id) { return (id+1) % N; }

// Synchronization mechanisms
Semaphore limit = 4;
Semaphores forks[N];
for (i=0; i<N; i++)
    forks[i] = 1;


void philosopher(int id)
{
    while (TRUE) {
        think();
        take_forks(id);
        eat();
        put_forks(id);
    }
}


void take_forks(int id)
{
    limit.P();
    forks[right(id)].P();
    forks[left(id)].P();
}

void put_forks(int id)
{
    forks[right(id)].V();
    forks[left(id)].V();
    limit.V();
}
```

6.3.23

Dining Philosophers. Solution 2

```
// Global variables
const int N = 5;
typedef enum {THINKIG, HUNGRY, EATING} STATE;
STATE state[N];

// Utility functions
int right(id) { return id; }
int left(id) { return (id+1) % N; }

Semaphore mutex = 1;
Semaphores permission[N];
for (i=0; i<N; i++)
    permission[i] = 0;


void philosopher(int id)
{
    while (TRUE) {
        think();
        take_forks(id);
        eat();
        put_forks(id);
    }
}

void take_forks(int id)
{
    mutex.P();
    state[id] = HUNGRY;
    test(id);
    mutex.V();
    permission[id].P();
}


void put_forks(int id)
{
    mutex.P();
    state[id] = THINKING;
    test(left(id));
    test(right(id));
    mutex.V();
}

void test(int id)
{
    if (state[id] == HUNGRY &&
        state[left(id)] != EATING &&
        state[right(id)] != EATING) {
        state[id] = EATING;
        permission[id].V();
}
```

6.3.24

3 Conclusions

What we talked about

- implementing some synchronization mechanisms with other synchronization mechanisms
  - semaphores with locks and condition variables
  - locks with semaphores

  - condition variables with semaphores
- some common synchronization patterns
  - rendezvous
  - barrier
  - readers / writers
  - dining philosophers

6.3.25

Lessons Learned

1. synchronization problems are complex
2. readers / writers synchronization pattern is a sort of “relaxed lock”

6.3.26

Chapter 7

Inter-Process Communication (IPC) Mechanisms

Pipes, Message Queues, Shared Memory

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

May 12, 2021

7.1

Purpose and Contents

The purpose of this chapter

- Presents the main mechanisms for communication between processes
- Presents different examples of using them

7.2

Bibliography

- A. Colesa, I. Ignat, Z. Somodi, *Sisteme de Operare. Chestiuni Teoretice si Practice*, 2007, Chapters 8 and 11, p. 105 – 117, p. 149 – 169 (Romanian only — see the pdf file on moodle page). For the english version see the html pages also on moodle page at Lecture resources.

7.3

Contents

7.4

IPC Mechanisms

- indirect communication
  - synchronization mechanisms, e.g. semaphores
- direct communication
  - wait/exit system calls
  - pipes
  - message queues
  - shared memory
  - sockets

7.5

1 Pipe (FIFO Files)

1.1 Characteristics and Communication Principles

Characteristics of a pipe

- provided **as a file**
  - accessed as a regular file (open, read, write, close)
- though, it is a **special file**
  - controlled and managed by the OS
  - exposed in a special format (FIFO)
  - imposed special rules to work with it
- it is **an IPC mechanism**
  - more processes (any number) can communicate through it
  - in a synchronized way

7.6

Communication Principles

- **based on producer/consumer** synchronization pattern
- **data access particularities** (differences by regular files)
  - **FIFO** principle, i.e. **sequential access** ⇒ there is **no seek** in the pipe
  - once read, data from pipe is no more available
  - circular fix-sized buffer, i.e. like a ring ⇒ **no end of file**
- **synchronization rules**
  1. **reading from an empty pipe blocks** the calling processes (consumer)
     - though, trying to read more bytes than available will not block
     - `read` returns the number of read bytes
  2. **writing into a full pipe blocks** the calling processes (producer)
- **synchronization exceptions**
  1. **reading** from an **empty pipe** with **no writer connected** returns immediately **as if end of file** was detected
  2. **writing** into a pipe with **no reader connected** triggers a kind of **exception** (Linux signals)

7.7

Uni- and Bi-directional Pipes

- normally **a user convention**, but sometime an OS support
- consider the **simple case of one producer and one consumer**
  - communication patterns, i.e. "**pipe usage types**"
    - **uni-directional**: used for a one-way communication
    - **bi-directional**: used for both directions
- in practice, though, there could be **any number of consumers and producers**
  - $\Rightarrow$ a sort of **N-directional pipes**
  - though, the OS could not manage making distinction between "directions", i.e. communication between two particular processes

7.8

Types of Pipes

- **nameless** (**anonymous**) pipes
- pipes **with name** (FIFO files)

7.9

Nameless (Anonymous) Pipes

- created with a special system call
- **have no name**
  - $\Rightarrow$ not visible as an element (file) in the file system
  - cannot be opened (like other visible files)
- not accessible, but to their creator process
  - just an in-memory OS data structure (and memory buffer)
  - invisible file automatically opened by the OS for the creator process
  - accessible through file handles (i.e. descriptors)
- used normally for **communication between processes in parent-child relationship**
  - based on the inheritance property
  - i.e. child inherits from its parent pipe's handles (file descriptors)
- $\Rightarrow$ **anonymous pipes must be created before child creation**
- once created and inherited, could be **used through normal file-related syscalls**
  - write and read on the returned file descriptors

7.10

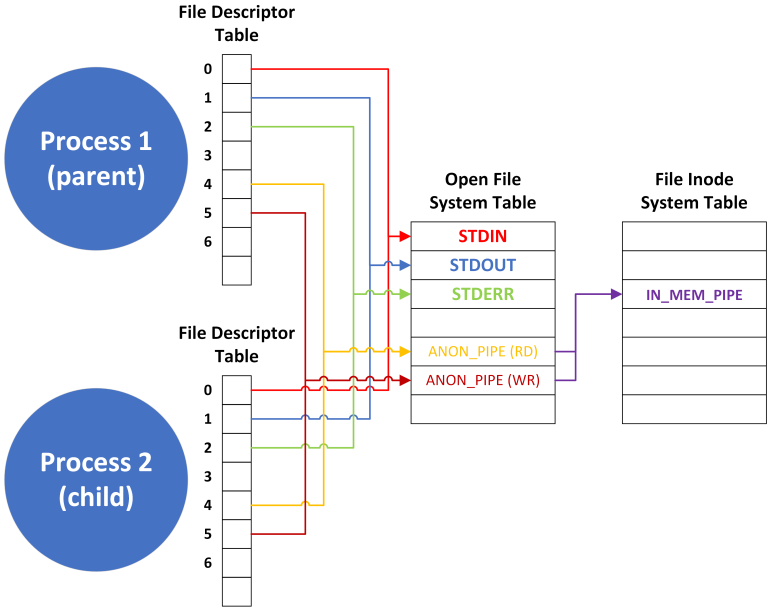
Anonymous Pipe Management in File System Tables

7.11

Pipes With Name (FIFO Files)

- created by a special system call
- **have a name**
  - $\Rightarrow$ visible in the file system
  - can be opened like any other files
- **any process** that wants to use them should **open them** before usage
- once created and opened, could be **used through normal file-related syscalls**
  - write and read on the returned file descriptor

7.12



1.2 Examples

Linux Anonymous Pipes

- system call

```
int pipe(int fd[2]);
```

- usage (uni-directional between two processes)

```
int fd[2];
int nr1 = 10, nr2;
pipe(fd);    // ex. fd[0] = 3 (for read)
             // ex. fd[1] = 4 (for write)
if (fork() == 0) { // child
     close(fd[0]);  // close access to pipe for read
     write(fd[1], &nr1, sizeof(int));
} else {            // parent
     close(fd[1]);  // close access to pipe for write
     read(fd[0], &nr2, sizeof(int));
}
```

7.13

Windows Anonymous Pipes

- Windows Win32 Function

```
BOOL WINAPI CreatePipe(
__out     PHANDLE hReadPipe,
__out     PHANDLE hWritePipe,
__in_opt  LPSECURITY_ATTRIBUTES lpPipeAttributes,
__in      DWORD nSize
);
```

- usage (unrealistically, in a loop, by the same process)

```
HANDLE hReadPipe = NULL;
HANDLE hWritePipe = NULL;
DWORD dwRead, dwWritten;
CHAR chBuf[BUFSIZE];
SECURITY_ATTRIBUTES saAttr;
saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
saAttr.bInheritHandle = TRUE;
saAttr.lpSecurityDescriptor = NULL;

CreatePipe(&hReadPipe, &hWritePipe, &saAttr, 0);

WriteFile(hWritePipe, chBuf, 10, &dwWritten, NULL);
ReadFile(hReadPipe, chBuf, 10, &dwRead, NULL);
```

7.14

Linux FIFO Files

- system call

```
int mkfifo(char *name, mode_t permissions);
```

- usage (unrealistically, in a loop, by the same process)

```
int fdPipe;
int nr1 = 10, nr2;

mkfifo("FIFO", 0644);
fdPipe = open("FIFO", O_RDWR);

write(fdPipe, &nr1, sizeof(int));
read(fdPipe, &nr2, sizeof(int));
```


Windows FIFO Files. Server Process

```
LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");

hPipe = CreateNamedPipe(
            lpszPipename,             // pipe name
            PIPE_ACCESS_DUPLEX,       // read/write access
            PIPE_TYPE_MESSAGE |       // message type pipe
            PIPE_READMODE_MESSAGE |   // message-read mode
            PIPE_WAIT,                // blocking mode
            PIPE_UNLIMITED_INSTANCES, // max. instances
            BUFSIZE,                  // output buffer size
            BUFSIZE,                  // input buffer size
            0,                        // client time-out
            NULL);                    // default security attribute

fConnected = ConnectNamedPipe(hPipe, NULL);

ReadFile(
    hPipe,        // handle to pipe
    pchRequest,    // buffer to receive data
    BUFSIZE*sizeof(TCHAR), // size of buffer
    &cbBytesRead, // number of bytes read
    NULL);        // not overlapped I/O
```


Windows FIFO Files. Client Process

```
LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");

hPipe = CreateFile(
            lpszPipename,   // pipe name
            GENERIC_READ |  // read and write access
            GENERIC_WRITE,
            0,              // no sharing
            NULL,           // default security attributes
            OPEN_EXISTING,  // opens existing pipe
            0,              // default attributes
            NULL);

WriteFile(
    hPipe,                  // pipe handle
    lpvMessage,             // message
    (lstrlen(lpvMessage)+1)*sizeof(TCHAR), // message length
    &cbWritten,             // bytes written
    NULL);                  // not overlapped
```


Linux Uni-directional Pipe

```
// First Process
main()
{
    // create the pipe
    // must be done before any process try opening the pipe
    mkfifo("FIFO", 0600);

    // open the pipe for WRITE only
    // the process will block until the second process open the pipe for READ
    int fdW = open("FIFO", O_WRONLY);

    // write into pipe
    // unblock second process
    write(fdW, "1", 1);
}

// Second Process
main()
{
    // open the pipe for READ only
    // unblock the first process
    int fdR = open("FIFO", O_RDONLY);

    // read from pipe
    // block until first process succeeds writing into
    read(fdR, &c, 1);
}
```

Linux Bi-directional Pipe. Intended Scenario

```
// First Process
main()
{
    mkfifo("FIFO", 0600);
    int fdW = open("FIFO", O_WRONLY);
    int fdR = open("FIFO", O_RDONLY);

    write(fdW, "1", 1);

    read(fdR, &c, 1);
}

// Second Process
main()
{
    int fdR = open("FIFO", O_RDONLY);
    int fdW = open("FIFO", O_WRONLY);

    read(fdR, &c, 1);
    // do something with the read data
    write(fdW, "2", 1);
}
```

7.19

Linux Bi-directional Pipe. Wrong Scenario

```
// First Process
main()
{
    mkfifo("FIFO", 0600);
    int fdW = open("FIFO", O_WRONLY);
    int fdR = open("FIFO", O_RDONLY);

    write(fdW, "1", 1);

    read(fdR, &c, 1);  // problem: could read what it has just written
}

// Second Process
main()
{
    int fdR = open("FIFO", O_RDONLY);
    int fdW = open("FIFO", O_WRONLY);

    read(fdR, &c, 1);  // problem: could be blocked forever
    // do something with the read data
    write(fdW, "2", 1);
}
```

7.20

Linux Bi-directional Communication. Functional Solution

```
// First Process
main()
{
    mkfifo("FIFO1", 0600);
    mkfifo("FIFO2", 0600);
    int fdW = open("FIFO1", O_WRONLY);
    int fdR = open("FIFO2", O_RDONLY);

    write(fdW, "1", 1); // writes on FIFO1

    read(fdR, &c, 1); // blocks until data becomes available on FIFO2
}

// Second Process
main()
{
    int fdR = open("FIFO1", O_RDONLY);
    int fdW = open("FIFO2", O_WRONLY);

    read(fdR, &c, 1); // blocks until data becomes available on FIFO1
    // do something with read data
    write(fdW, "2", 1); // writes on FIFO2
}
```

7.21

Command Line Pipe (2 commands)

```
// Command interpreter's handling of command lines like
// cmd_0 | cmd_1
// e.g. "cat file.txt | wc -l"

// prepare for handling an anonymous pipe
```

```
// used between cmd_0 and cmd_1
int fd[2];

// command paths (names) and arguments
char cmd[2][256];
char argv[2][256][256];

prepare_cmds_and_args(cmd, argv);

// pipe creation must be done
// before creating processes that use that pipe
pipe(fd);
```

7.22

Command Line Pipe (2 commands) (cont.)

```
// first child process creation
if (fork() == 0) {
    // child executing cmd_0
    close(fd[0]);    // not reading from pipe
    dup2(fd[1], 1);  // redirect 1 (STDOUT) to pipe
    close(fd[1]);    // not using anymore the pipe explicitly
                     // critical to be done
    // load command code
    execvp(cmd[0], argv[0]);   // cmd[0] == argv[0][0]
    exit(1);         // executed only when execvp fails
}
```

7.23

Command Line Pipe (2 commands) (cont.)

```
// second child process creation
if (fork() == 0) {
    // child executing cmd_1
    close(fd[1]);    // not writing into pipe
    dup2(fd[0], 0);  // redirect 0 (STDIN) to pipe
    close(fd[0]);    // not using anymore the pipe explicitly

    // load command code
    execvp(cmd[1], argv[1]);    // cmd[1] == argv[1][0]
    exit(1);         // executed only when execvp fails
}

// parent
// not using the pipe, so close it
close(fd[0]);
close(fd[1]); // critical to be done

wait(NULL);
wait(NULL);
```

7.24

Two Commands Linked by an Anonymous Pipe

7.25

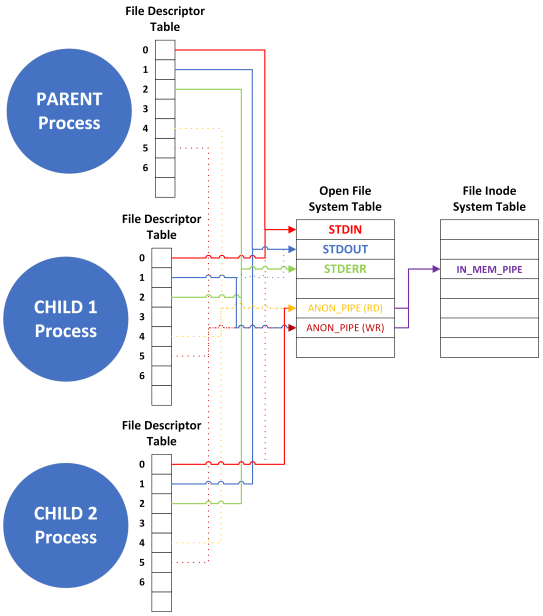
Command Line Pipe (n commands)

- command line syntax

  ```
  cmd_0 | cmd_1 | ... | cmd_n-1
  ```

- command interpreter (shell) code

  ```
  // prepare for handling n-1 pipes
  // pipe_0 between cmd_0 and cmd_1
  //  fd[0][0] for reading from pipe_0
  //  fd[0][1] for writing into pipe_0
  // ....
  int fd[n][2];

  // command paths (names) and arguments
  char cmd[n][256];
  char argv[n][256][256];

  prepare_cmds_and_args(cmd, argv);
  ```




```
for (i=0; i<n; i++) {

    // pipe creation must be done
    // before creating processes that use that pipe
    // e.g. pipe_0 before cmd_0 and cmd_1
    if (i < n-1)
        pipe(fd[i]);

    // child process creation
    if (fork() == 0) {
        // if not cmd_0
        if (i > 0) {
            close(fd[i-1][1]);
            dup2(fd[i-1][0], 0);
            close(fd[i-1][0]);
        }


        // if not cmd_n-1
        if (i < n-1) {
        close(fd[i][0]);
        dup2(fd[i][1], 1);
        close(fd[i][1]);
        }

        // load command code
        execvp(cmd[i], argv[i]);
    }
    else {
        close(fd[i][0]);
        close(fd[i][1]);
    }
}
```

7.26

Practice (1)

1. There are three processes running into the system, whose code is given below. Change and complete the code of the three processes (writing all the code in just one C file and adding the code to create the processes), such that the communication to be possible and the contents of the buffer to be

   (a) "ab" or

   (b) "ba"

   at the end of the execution of the three processes.

```c
// Process P1
int fd[2];

pipe(fd);
...
write(fd[1], "a", 1);
```

```c
// Process P2

...
...
write(fd[1], "b", 1);
...
```

```c
// Process P3
char buf[2];
...
read(fd[0],&buf[0],1);
read(fd[0],&buf[1],1);
...
```

7.27

Practice (2)

2. Write the C code/pseudo-code to find out the size of
   (a) an anonymous pipe;
   (b) a named pipe.

7.28

Practice (3)

3. Implement the semaphore's primitives `P()` and `V()` using pipes.

7.29

2 Message Queues

2.1 Characteristics and Communication Principles

Characteristics

- a more specialized pipe
- group bytes in messages, making distinction between messages
- different types (labels) of messages
- processes can get messages of specified type from queue

7.30

Communication Principles

- same as pipe, though sometimes FIFO order can be broken, when a message of a specified type is requested

7.31

2.2 Examples

Two-Way Communication. System V Msg Queues

- first process

```c
struct msg {
    long type;
    int data;
} msg1, msg2;

int msgId = msgget(10000, IPC_CREAT | 0600);
msg1.type = 1;
msg1.data = getpid();

// send a message of type 1
msgsnd(msgId, &msg1, sizeof(msg1) - sizeof(long), 0);

// gets a message of type 2
msgrcv(msgID, &msg2, sizeof(msg2) - sizeof(long), 2, 0);
```

- second process

```c
struct msg {
    long type;
    int data;
} msg1, msg2;

int msgId = msgget(10000, 0);
msg2.type = 2;
msg2.data = getpid();

// gets a message of type 1
msgrcv(msgID, &msg1, sizeof(msg1) - sizeof(long), 1, 0);

// send a message of type 2
msgsnd(msgId, &msg2, sizeof(msg2) - sizeof(long), 0);
```

7.32

Two-Way Communication. POSIX Msg Queues

- first process

```c
struct msg {
    int data;
} msg1, msg2;

mqd_t msg_id = mq_open("/my_msg_queue", O_CREAT | O_RDWR, NULL);

msg1.data = getpid();

// send a message
mq_send(msg_id, &msg1, sizeof(msg1), 0);

// gets a message (could be the one just sent)
mq_receive(msg_id, &msg2, sizeof(msg2), NULL);
```

- second process

```c
struct msg {
    int data;
} msg1, msg2;

mqd_t msg_id = mq_open("/my_msg_queue", O_RDWR, NULL);

msg2.data = getpid();

// gets a message (could not found one)
mq_receive(msg_id, &msg1, sizeof(msg1), NULL);

// send a message
mq_send(msg_id, &msg2, sizeof(msg2), 0);
```

7.33

3 Shared Memory

3.1 Characteristics and Communication Principles

Characteristics

- a common (shared) memory area belonging to more process address spaces
- created with a special system call
- managed by the OS
- the fastest IPC mechanism

7.34

Principles

- no need for special system calls for communication (only for creation)
- used as any process memory area referenced by a pointer
- sequence of bytes, whose structure is known by the collaborating processes
- the concurrent accesses to it is NOT synchronized by the OS ⇒ the processes SHOULD make this

7.35

3.2 Examples

System V Shared Memory

- first process

```
int shm_id = shmget(10000, sizeof(int), IPC_CREAT | 0600);
int *i = shmat(shm_id, 0, 0);
*i = 10; // write shared memory -> "send" information
```

- second process

```
int shmId = shmget(10000, 0, 0);
int *j = shmat(shm_id, 0, 0);
printf("Received info is %d\n", *j);  // read the shared memory -> "receive" information
```

7.36

POSIX Shared Memory

- first process

```
int main (int argc, char **argv)
{
        int shm_id, *pInt = NULL;

        shm_unlink("/my_shm");
        shm_id = shm_open("/my_shm", O_CREAT | O_EXCL | O_RDWR, 0666);
        if (shm_id < 0) {
                perror("Cannot create the shared memory");
                exit(1);
        }

        ftruncate(shm_id, sizeof(int));

        pInt = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_id, 0);
        if (pInt == NULL) {
                perror("Cannot map the shared memory");
                exit(2);
        }

        close(shm_id);

        *pInt = 100;

        munmap(pInt, sizeof(int));
}
```

- second process

```
int main (int argc, char **argv)
{
        int shm_id;
        int *pInt = NULL;

        shm_id = shm_open("/my_shm", O_RDWR, 0);
        if (shm_id < 0) {
                perror("Cannot open the shared memory");
                exit(1);
        }

        pInt = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_id, 0);
        if (pInt == NULL) {
                perror("Cannot map the shared memory");
                exit(2);
        }

        close(shm_id);

        printf("shm = %d\n", *pInt);

        munmap(pInt, sizeof(int));
}
```

7.37

4 Conclusions

What we talked about

- inter-process communication mechanisms (IPC)
- pipes
  - data handled based on FIFO principle
  - synchronized based on the producer-consumer pattern
  - with name vs anonymous (i.e. nameless)
  - unidirectional vs bidirectional
- message queues
  - similar to pipes, yet more specialized
  - make distinction between messages
  - messages could be labeled
- shared memory
  - explicitly share memory between processes
  - accessed using pointers, like dynamically allocated memory
  - communication: write and read from the shared memory

7.38

Lessons Learned

1. by default processes are isolated, i.e. share nothing
2. yet, they could communicate using explicitly designed mechanisms, i.e. IPC mechanisms
3. pipes and message queues are synchronized IPC mechanisms
4. shared memory is the fastest IPC mechanism, yet provides no synchronization

7.39

Chapter 8
Memory Management
Basic Management Techniques

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

May 19/26, 2021

8.1

Purpose and Contents

The purpose of this chapter

1. presents basic role and functionality of a memory manager
2. describes different memory management techniques
    - contiguous allocation, swapping, paging, segmentation

8.2

Bibliography

- Andrew Tannenbaum, *Modern Operating Systems*, second edition, Prentice Hall, 2001, pg. 190 – 263.

8.3

Contents

8.4

1 Context and Definitions

Context

- program's execution phases
    - user writes the program
    - compiler (assembler) and linker generates the executable
    - OS loads the executable in memory ⇒ process
    - process gets executed by the CPU

  - on-chip MMU (memory management unit) accesses the memory
- main memory (RAM) and registers are
  - the only storage directly accessible by CPU (i.e. MMU)
  - though, some levels of cache also, but we do not talk about this
- if some needed data (e.g. instructions, operands) is not in memory
  - it must be moved there before the CPU can operate on it

8.5

Memory Needs

- ideally, we would like the memory to be
  - very large
  - very fast
  - non-volatile
- in reality, there is a hierarchy of different types of memory (speed vs. capacity vs. cost)
  - register access is faster than memory access
    - while register access lasts one CPU cycle, a memory access may need more CPU cycles
  - faster memory, named *cache*, is used between registers and main memory
  - slower memory (storage area) is used for higher capacity and non-volatility needs
- memory protection must be provided
  - OS and processes must be protected from one another

8.6

Memory Manager

- an OS component
- keeps track of which parts of memory are in use and which are not
- allocates and releases areas of main memory to processes
- influences how each process sees the system's memory
  - based on hardware support
- coordinates how the different types of memory are used
  - moves processes' contents between different memory layers (e.g. cache, main memory, disk), to save space and improve process performance
  - invalidates cache entries for security reasons
- provides memory protection
  - based on hardware support

8.7

Basic Memory Configuration

- only one user program (process) in memory at one moment
  - usually at a known, established location
- the OS also resides in memory
  - so, actually we have two processes in memory
- disadvantages
  - a limited configuration
  - supporting multiprogramming is difficult

8.8

Basic Memory Configuration. Illustration

8.9

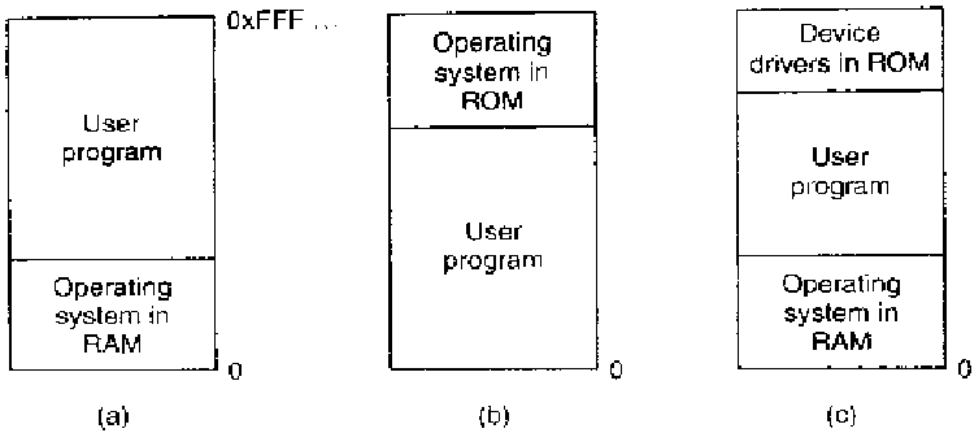


Figure 1: Tanenbaum, "Modern OS", Figure 4.1

A More Realistic Configuration

- **multiprogramming** systems
  - more processes loaded simultaneously in memory
- improves CPU (and other resources) utilization
- ⇒ **different processes loaded in different places in memory**, i.e. at different memory addresses

8.10

A More Realistic Configuration. Illustration

8.11

A More Realistic Configuration. Challenges

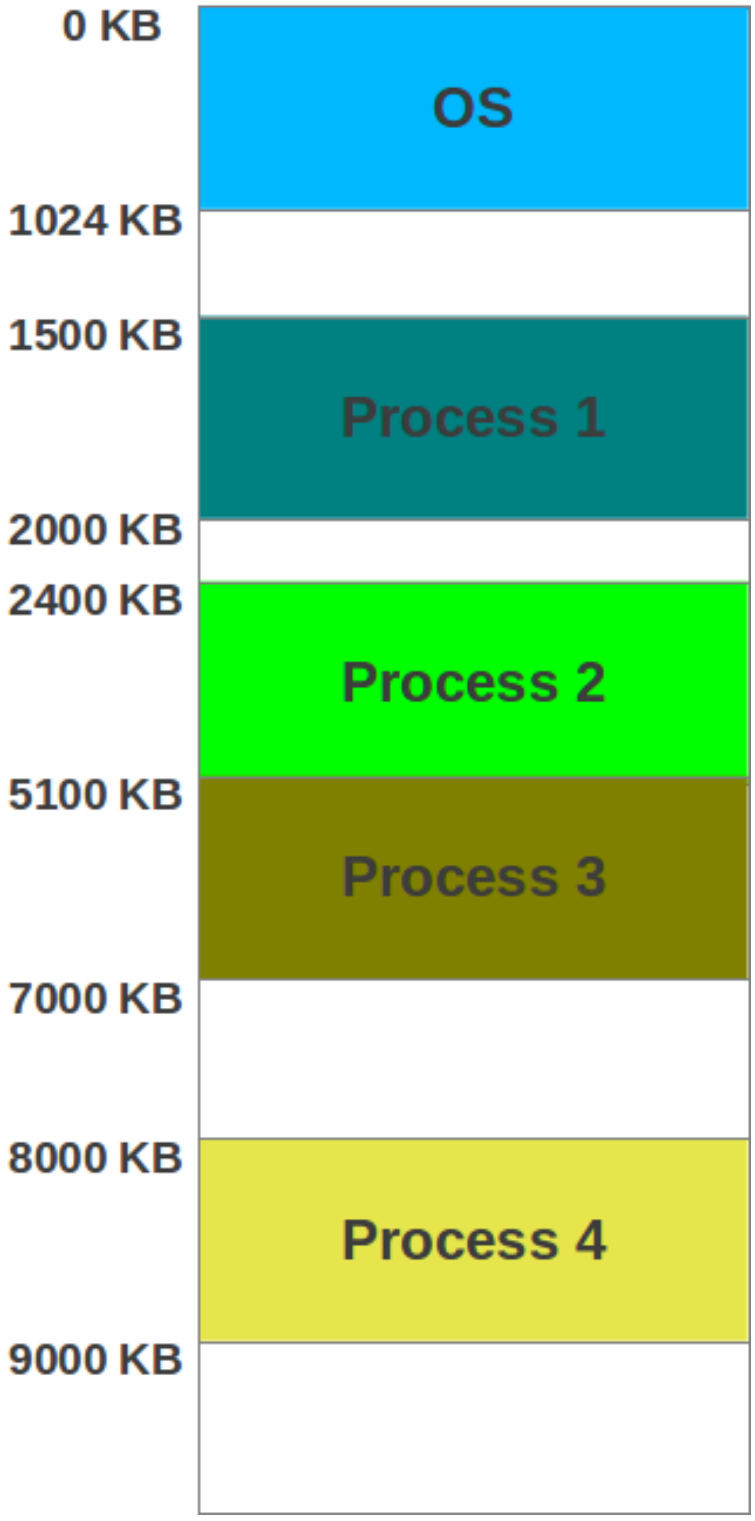
- memory addresses one program / process uses cannot be the same like those of other processes
- when one program's executable is generated, the compiler
  - mono-programming case
    - knows where that program would be loaded in memory
    - knows which are the memory addresses for that program
  - multi-programming case
    - does not know where that program will be loaded in memory
    - does not know which are the memory addresses for that program

8.12

C Source Code Example (`memory-layout.c`)

```
int x;

int inc(int v)
{
  int aux;

  aux = v;
  aux++;

  return aux;
}


int main(int argc, char **argv)
{
  x = 10;

  x = inc(x);

  return x;
}
```

8.13


0 KB
OS
1024 KB
1500 KB
Process 1
2000 KB
2400 KB
Process 2
5100 KB
Process 3
7000 KB
8000 KB
Process 4
9000 KB

Compilation and Executable Displaying Commands

```
# Compile and generate executable
#    "-m32" for x86 (32-bit) architecture
#    "-g"   compile for debugging,
#           including symbol and source code references
#    "-fno-stack-protector"
#           do not include code for stack protection
gcc -Wall -Werror -g -m32 -fno-stack-protector \
    memory-layout.c -o memory-layout

# Display executable's contents
#    "-d"   disassemble code
#    "-S"   show code source intermixed with disassembly
#    "-j .text"
#           for the code section
#    "--disassembler-options=intel"
#           usig Intel syntax
objdump --disassembler-options=intel -d -S \
        -j .text memory-layout
```

8.14

The Resulted Executable (`memory-layout`)

```
080483db <inc>:
#include <stdio.h>

int x;

int inc(int v)
{
 80483db:   55                      push   ebp
 80483dc:   89 e5                   mov    ebp,esp
 80483de:   83 ec 10                sub    esp,0x10
  int aux;

  aux = v;
 80483e1:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 80483e4:   89 45 fc                mov    DWORD PTR [ebp-0x4],eax
  aux++;
 80483e7:   83 45 fc 01             add    DWORD PTR [ebp-0x4],0x1

  return aux;
 80483eb:   8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
}
 80483ee:   c9                      leave
 80483ef:   c3                      ret


080483f0 <main>:


int main(int argc, char **argv)
{
 80483f0:   55                      push   ebp
 80483f1:   89 e5                   mov    ebp,esp
  x = 10;
 80483f3:   c7 05 1c a0 04 08 0a    mov    DWORD PTR ds:0x804a01c,0xa
 80483fa:   00 00 00

  x = inc(x);
 80483fd:   a1 1c a0 04 08          mov    eax,ds:0x804a01c
 8048402:   50                      push   eax
 8048403:   e8 d3 ff ff ff          call   80483db <inc>
 8048408:   83 c4 04                add    esp,0x4
 804840b:   a3 1c a0 04 08          mov    ds:0x804a01c,eax

  return x;
 8048410:   a1 1c a0 04 08          mov    eax,ds:0x804a01c
}
 8048415:   c9                      leave
 8048416:   c3                      ret
```

8.15

Questions

- **Question 1**: Who is responsible for establishing / specifying memory location a process will be loaded at, i.e. memory addresses that process will use?
- due to uniformity and transparency reasons, we could have
  - the compiler / process presumes (sees) one kind of memory layout
    - e.g. supposes the process will be loaded at the beginning of the memory, i.e. memory address 0 (zero)
  - the operating system and the MMU sees another kind of memory layout
- $\Rightarrow$ there could be
  - logical (virtual) memory layout $\Rightarrow$ **logical (virtual) memory addresses**
  - real (physical) memory layout $\Rightarrow$ **real (physical) memory addresses**
- **Question 2**: Who is responsible for translating logical addresses to physical addresses?

8.16

Address Binding

- binding = the way different types of addresses are related
- compile time
  - if starting memory address the program will be loaded at execution is known, than *absolute code* can be generated
  - if not (common case), *relocatable code* must be generated
- load time
  - binding is done at load time
  - if starting address changes, the program must be reloaded
- execution time (runtime)
  - if process can be moved during its execution, binding must be delayed until runtime
  - hardware support is needed for such a scheme to work

8.17

Practice (1)

1. Having the following partial contents of an executable file (i.e. an instruction accessing a memory location), illustrate, for the following two cases, the corresponding contents (i.e. how the instruction looks like) of the physical memory allocated to the process executing the given code (instruction), considering that process to be loaded in memory starting from address `0x1000000`:
   (a) load-time binding;
   (b) runtime binding.

```
mov     DWORD PTR ds:0x804a018,0x0
```

8.18

Logical (Virtual) vs. Physical Address Space

- **address space** = the set of all memory addresses accessible by a process
  - **virtual address space (VAS)**
  - **physical address space (PAS)**
- compiler generates virtual (logical) addresses (VAs) $\rightarrow$ VAS
- CPU, at runtime, tries accessing VAs $\rightarrow$ VAS
- MMU accesses physical addresses (PAs) $\rightarrow$ PAS

1. the compile-time binding method generates identical VAS and PAS
   - $\Rightarrow$ no need for VA to PA translation
2. the load-time binding method generates different VAS and PAS
   - $\Rightarrow$ VA to PA translation done by OS loader
3. the execution-time binding method generates different VAS and PAS
   - $\Rightarrow$ VA to PA translation done by MMU

8.19

General Layout of a Process' VAS

8.20

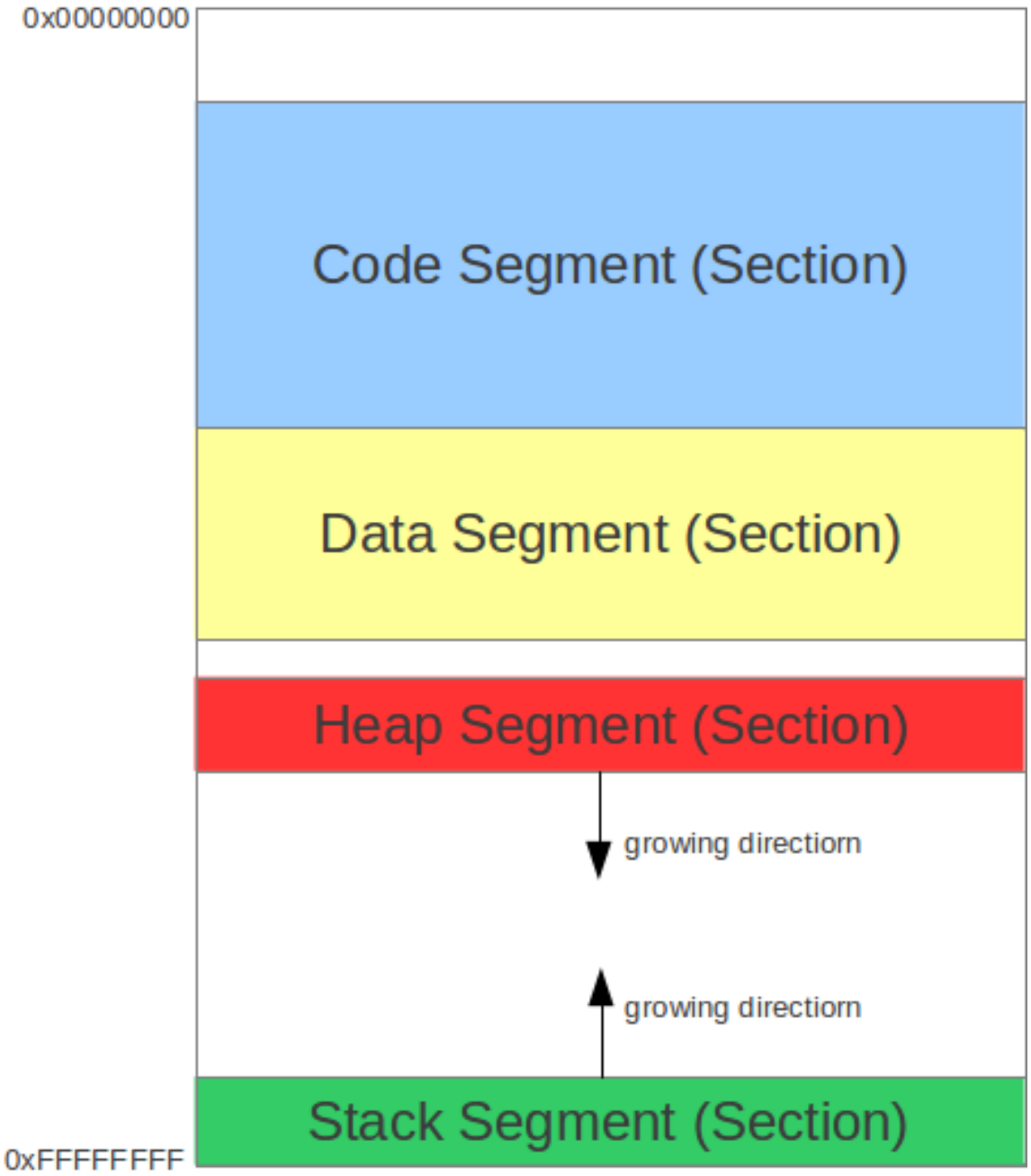
2 Basic Memory Management Techniques

2.1 Contiguous Allocation

General Rules

- each process is allocated a single contiguous area
- OS and processes must be allocated different areas
- usually the OS is mapped in low memory and is resident

8.21


0x00000000
Code Segment (Section)
Data Segment (Section)
Heap Segment (Section)
growing directiorn
growing directiorn
Stack Segment (Section)
0xFFFFFFFF

Memory Mapping (Binding) and Protection

- could be done dynamically based on the base and limit values
  - it is a hardware support
  - there are two registers: base and limit
  - each memory address is compared to the limit and added the base register
  - each process is associated different base and limit values, depending on where the process is allocated memory and its size
  - the two registers could be set only by the OS
- supports moving of processes in memory due to OS or processes variation
- problem: does not support mapping of virtual address space with holes

8.22

Allocation Strategy

- multiple fixed-size (not necessarily equal) partitions
  - two allocation strategies
    - a different waiting queue for each partition
    - one global queue for all partitions
  - the number of concurrent processes depends on the number of partitions
  - too rigid
- multiple variable-size partitions
  - each process is allocated only the space it needs
  - the system keeps track of available partitions
    - they could be ordered in some way to help finding the good one
    - they might be merged, when released

8.23

Allocation Strategy (cont.)

- which free partition to allocate?
  - *first-fit*, *next-fit*, *best-fit*, *worst-fit*
- what happens when not enough memory available?
  - another process from input queue is allocated memory
  - all memory allocations are delayed until enough memory available for the next process in input queue

8.24

Fragmentation

- external
  - free space is divided in small areas, which at some moment cannot be used, due to the contiguous allocation strategy
    - i.e. no large enough contiguous free area to cover the requested amount of memory
  - solution: compaction of allocated areas at one end of memory
- internal
  - in practice it is too expensive to keep track of small areas (few bytes) $\Rightarrow$ allocate memory in terms of blocks

8.25

Practice (2)

2. Supposing an OS uses contiguous allocation strategy for memory management and processes `P1`, `P2`, and `P3` was allocated contiguous memory areas starting at `0x1000000`, `0x3000000`, `0x6000000`, respectively, illustrate the evolution of the base register's value when processes are scheduled in the following order: `P2`, `P1`, `P2`, `P3`.

8.26

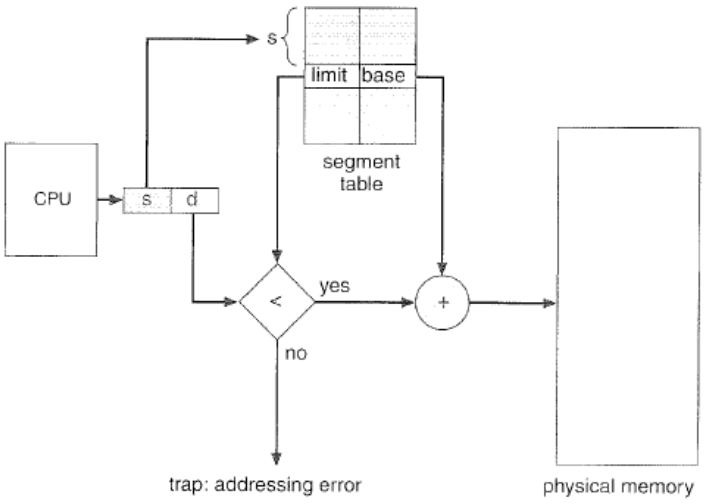


Figure 2: Silberschatz, p. 304

2.2 Segmentation

Method's Principles

- different user view of a process (program)
  - not a single contiguous area of bytes, but
  - more different areas (segments) of variable sizes
  - separated by unused areas, i.e. not a contiguous view of the address space
  - data are referred to using their offset in the corresponding segment
- logical address space is a collection of segments
  - each segment has a name (identifier) and a length
  - a logical address consists of a segment identifier and an offset
  - each segment could contain a particular (different) types of data
  - examples: code, global variables, heap, stack, standard C library, other libraries
- physical memory allocated in contiguous areas
  - i.e. each segment one contiguous area

8.27

Address Binding (Translation)

- segment table
  - a generalization of the base and limit registers used for (single area) contiguous allocation
- each segment is allocated an entry in the segment table
  - the entry contains the base and limit values for that segment
- segment table usually in memory
  - not so many CPU registers
  - CPU contains only the address and size of the segment table

8.28

Hardware Illustration

8.29

2.3 Paging

Definition

- divide memory (and swap area) in fixed-size blocks
- allocation is done in terms of those fixed-size blocks
  - physical memory is divided in *frames*
  - logical memory is divided in *pages*
- page size is defined by hardware
  - typically a power of 2

8.30

Method's Principles

- allows allocation in non-contiguous areas
  - all pages / frames of the same size
  - $\Rightarrow$ any available page (free area) could be allocated
- avoids external fragmentation
- though suffers by internal fragmentation
  - not all allocated space (in a page) is used
  - the unused ("free"), yet allocated, space is lost
  - reduce it: decrease page size $\Rightarrow$ overhead for page table and I/O operations
- common page sizes: $4KB - 8KB$

8.31

Address Binding (Translation)

- paging is "similar" to segmentation $\rightarrow$ each page a segment
- $\Rightarrow$ page table = a collection of base and limit registers
  - one pair (entry) for each page $\Rightarrow$ page table (PT)
  - though the limit is not needed as all pages have the same size
- difference by segmentation
  - there is an entry in the PT for each virtual page in the process
  - even for pages in unused areas
  - though, such pages are marked as unmapped (invalid)
  - paging is orthogonal to segmentation

8.32

Mapping One Virtual Address Space With Paging

8.33

Mapping Two Virtual Address Spaces With Paging

8.34

Paging and Page Tables Illustration

8.35

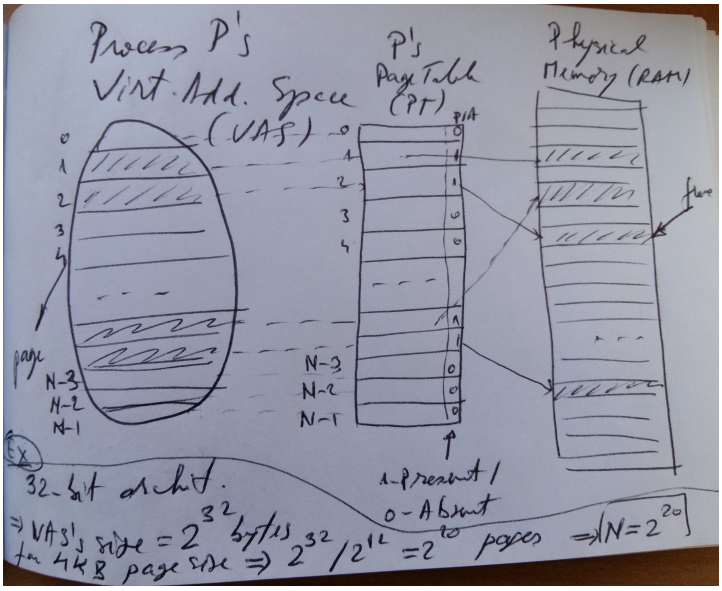
Address Translation During Runtime

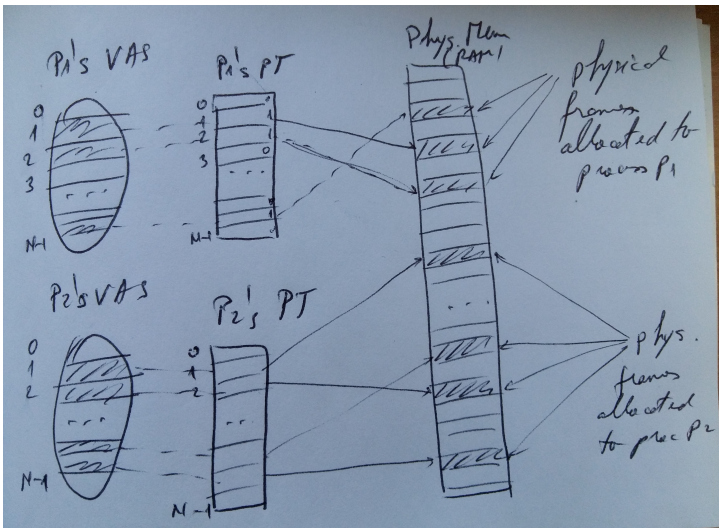
8.36

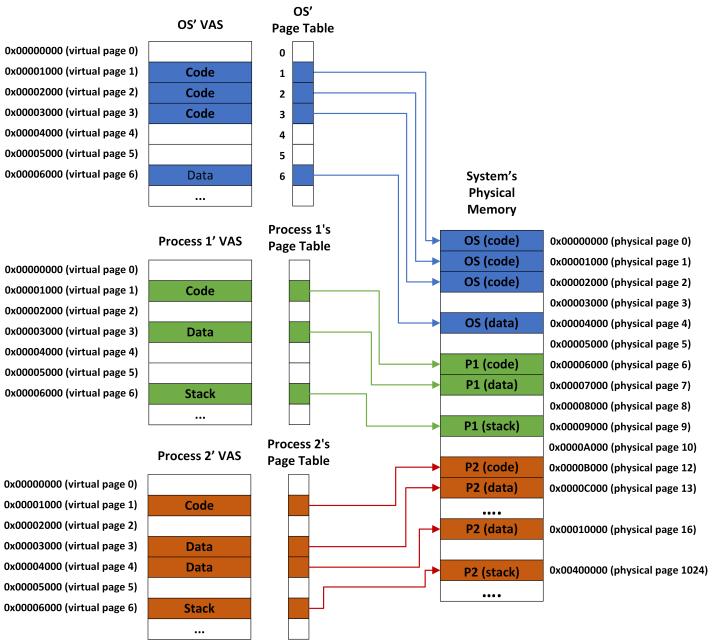
Practice (3)

3. Which is the size in bytes of a process' virtual address space, in case of a system using 32 bits for virtual memory addresses? What about a 64-bit system?
4. Which is the number of entries in the page table for both cases mentioned above?

8.37


Process P's
Virt. Add. Space
(VAS)
P's
Page Table
(PT)
Physical
Memory (RAM)
page
N-3
N-2
N-1
P/A
1-Present /
0-Absent
32-bit archit.
⇒ VAS's size = 2^32 bytes
for 4KB page size ⇒ 2^32 / 2^12 = 2^20 pages ⇒ N = 2^20


P1's VAS
P1's PT
Phys. Mem (RAM)
physical frames allocated to process P1
P2's VAS
P2's PT
phys. frames allocated to proc P2
N-1


OS' VAS
OS' Page Table
0x00000000 (virtual page 0)
0x00001000 (virtual page 1)
Code
0x00002000 (virtual page 2)
Code
0x00003000 (virtual page 3)
Code
0x00004000 (virtual page 4)
0x00005000 (virtual page 5)
0x00006000 (virtual page 6)
Data
...
0
1
2
3
4
5
6
System's Physical Memory
OS (code)
0x00000000 (physical page 0)
OS (code)
0x00001000 (physical page 1)
OS (code)
0x00002000 (physical page 2)
0x00003000 (physical page 3)
OS (data)
0x00004000 (physical page 4)
0x00005000 (physical page 5)
P1 (code)
0x00006000 (physical page 6)
P1 (data)
0x00007000 (physical page 7)
0x00008000 (physical page 8)
P1 (stack)
0x00009000 (physical page 9)
0x0000A000 (physical page 10)
P2 (code)
0x0000B000 (physical page 12)
P2 (data)
0x0000C000 (physical page 13)
....
P2 (data)
0x00010000 (physical page 16)
P2 (stack)
0x00400000 (physical page 1024)
....
Process 1' VAS
Process 1's Page Table
0x00000000 (virtual page 0)
0x00001000 (virtual page 1)
Code
0x00002000 (virtual page 2)
0x00003000 (virtual page 3)
Data
0x00004000 (virtual page 4)
0x00005000 (virtual page 5)
0x00006000 (virtual page 6)
Stack
...
Process 2' VAS
Process 2's Page Table
0x00000000 (virtual page 0)
0x00001000 (virtual page 1)
Code
0x00002000 (virtual page 2)
0x00003000 (virtual page 3)
Data
0x00004000 (virtual page 4)
Data
0x00005000 (virtual page 5)
0x00006000 (virtual page 6)
Stack
...

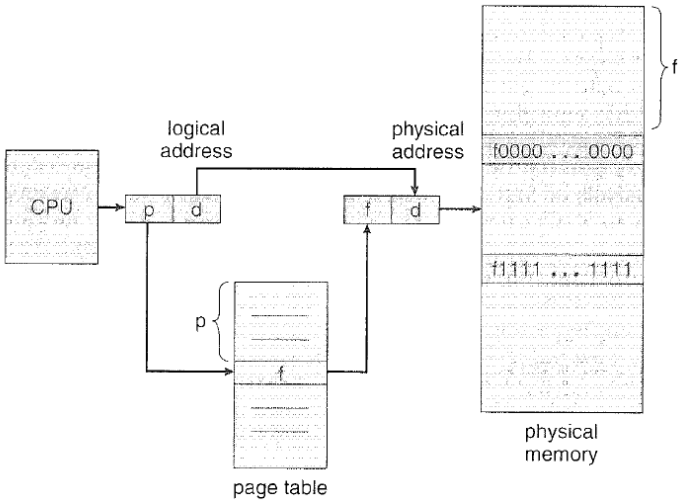


Figure 3: Silberschatz, p. 288

Hardware Support for Paging

- registers to store page table entries
  - advantage: fast
  - disadvantage: appropriate for small tables
- page-table base register (PTBR)
  - page table kept in memory
  - advantage: appropriate for large page tables
  - disadvantage: slower, due to additional memory accesses

8.38

Hardware Support for Paging (cont.)

- translation look-aside buffer (TLB)
  - associative, **high-speed memory**
  - a **TLB entry** consists of a **key (VA)** and a **value (PA)**
  - contains some (most used) page table entries (PTEs)
  - advantage: a key (VA) is searched very fast (in parallel on all entries)
  - disadvantage: expensive and small (typically 64 – 1024 entries) $\Rightarrow$ **TLB miss**
  - replacement algorithms must be used
  - some entries could be *locked* (typically the OS's entries)
  - a TLB entry could also contain an address-space identifier (ASID) to be able to store page of different processes simultaneously
  - when not having ASID, the TLB must be flushed at each context (process) switch

8.39

Paging Mechanism in Hardware with TLB

8.40

Protection

- based on some bits (flags) associated to each page and supported by the hardware
  - read (R), write (W), execute (X)
  - 1 $\rightarrow$ allows the operation
  - 0 $\rightarrow$ disallow the operation
- valid / present bit
  - 1 $\rightarrow$ legal / present page
  - 0 $\rightarrow$ illegal / non-present page

8.41

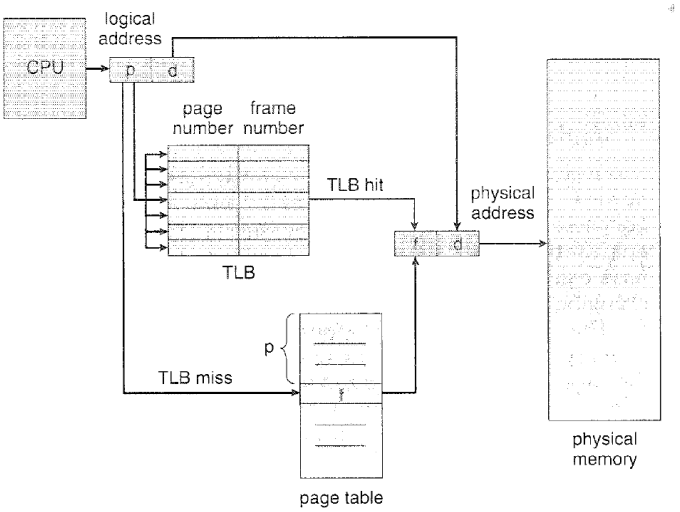
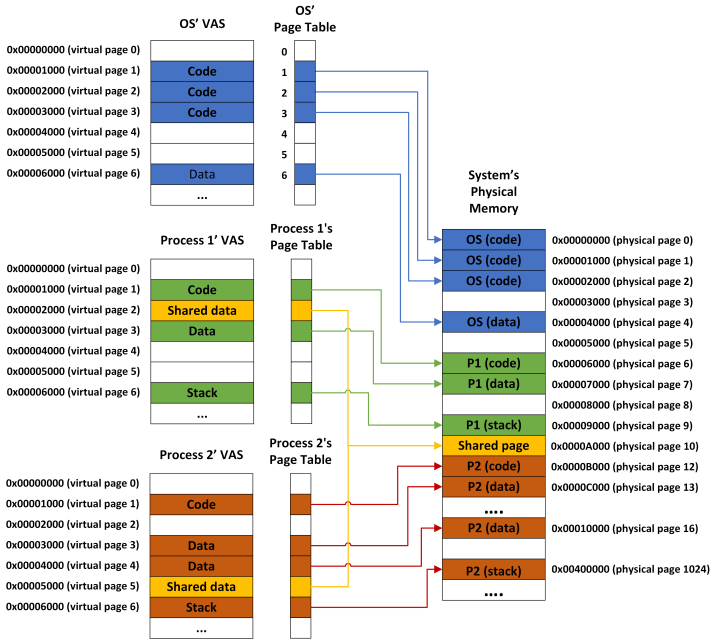


Figure 4: Silberschatz, p. 294



Shared Pages

- same physical pages (frames) shared between different processes
- the shared memory could be mapped on different areas in each process' VAS
- easy to be done using page tables
- useful especially for shared reentrant code
  - the read-only nature of shared code should not be let to the correctness of code
  - the OS should enforce this property
- used provide memory-based inter-process communication mechanisms

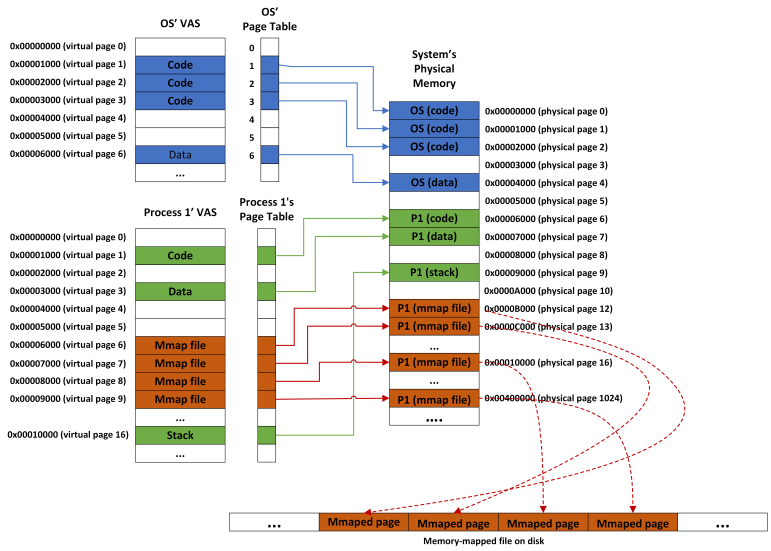
8.42

Shared Memory Illustration

8.43

Memory Mapped Files

- map parts of a file into a process' virtual address space
- provide a memory-like interface to a file contents
- advantage: transparency, performance
- when used with memory lazy-loading mechanisms



  - only accessed parts of the file are loaded in memory (transparently done by OS)
  - advantage: efficiently manage huge files
- limitations
  - file could not be extended

8.44

Memory-Mapped File Illustration

8.45

Practice (4)

4. You are given the following C code of a process running on a system that uses paging with pages of 4KB in size. You have to illustrate on the process's page table the way the allocated memory referred to by the "*p*" pointer could be mapped in the physical memory, using non-consecutive physical memory frames (pages).

```
1 char *p;
2 p = (char*) malloc (4097 * size(char));
3 printf("p = %p\n", p);    // displays p = 10240
```

5. What will be the effect of the following instructions, executed after the ones given above.

```
1 *p = 0;
2 p[4096] = 10;
3 p[4100] = 20;
4 p[2*4096+1] = 30;
```

6. How much space is lost due to internal fragmentation?
7. How much space is lost due to external fragmentation?

8.46

2.4 Swapping

Definition

- method to extend the main memory
- swap out (to some backing store) to make space for other process
- swap in (back in memory) to be available for execution
- there could be different strategies for swap out/in depending on the scheduling algorithm
  - round-robin
  - priority based
- swapping granularity
  - *hole process level*

    - it is too slow to swap out/swap in an entire process at CPU switch
  - *parts-of-process level*
    - more feasible in practice
    - besides, running a process does not require the entire process to be in memory, but only the parts needed for current instruction

8.47

Restrictions

- binding method
  - if binding is done at compile-time or load-time, swap (back) in must be done at the same position (address) from where the process was swapped out
  - if binding is done at run-time, the process can be swapped in (back) at any position
- transfer speed (time) to and from backing store (normally the dominant term in switching time)
  - it would be useful for the system to know exactly how much memory a process needs, to swap in only that part, in order to reduce the swap in (transfer) time

8.48

3 Security Issues

Overview

- common programmers' mistakes related to memory
  - some lead to **program crash**
  - some lead to **program misbehavior**
  - some turn into **vulnerabilities** that could be **exploited** by an **attacker**
    - e.g. lead to execution of unintended code, in attacker's advantage
- possible solutions or mitigation techniques
  - compiler based
  - OS based
- (some) explained in relation to paging, though independent of it
  - based on the fact that a page table does not map all pages, but only the ones really filled with the program code and data
  - the virtual pages (VP) in holes are marked as invalid in the corresponding page table entries (PTE)

8.49

Usage of a Non-Initialized Pointer

- **global** pointer
  - this will be automatically **initialized with 0, i.e. NULL**
    - as all global variables are
  - lead to **program crash**, due to invalid memory access, i.e. "*page fault*" exception
    - usually the page 0 (so VA zero) is not mapped
- **local** pointer, i.e. declared in a function
  - allocated (as any local variable) on the stack $\Rightarrow$ its **initial value is undefined**
  - lucky case: its value corresponds to an unmapped virtual page
    - generates a "*page fault*" exception
    - $\Rightarrow$ lead to **program crash**
  - unlucky case: its value corresponds to a mapped page
    - obviously not to a page desired by the user
    - accesses and possible changes go undetected until maybe later time
    - read / overwrite random data in the program
    - $\Rightarrow$ very **tricky bugs**
    - $\Rightarrow$ exploitations: **data leakage**, **execute unintended code**

8.50

Usage of a Non-Initialized Pointer. Example

```
#include <stdio.h>

#define MAX_LEN 4096

int* global_pointer;

int rec_function(int no)
{
    char* local_pointer;

    printf("global_pointer = %p, local_pointer = %p\n", global_pointer, local_pointer);

    //local_pointer[0] = 0;

    if (no > 0)
        rec_function(no-1);

    return 0;
}

int main(int argc, char **argv)
{
    rec_function(9);
    printf("------\n");
    rec_function(9);

    return 0;
}
```

```
$ gcc -Wall uninitialized-pointers.c -o uninitialized-pointers.exe
uninitialized-pointers.c: In function 'rec_function':
uninitialized-pointers.c:11:2: warning: 'local_pointer' is used uninitialized in this function [-Wuninitialized]
  printf("global_pointer = %p, local_pointer = %p\n", global_pointer, local_pointer);
  ^
$ ./uninitialized-pointers.exe
global_pointer = (nil), local_pointer = (nil)
global_pointer = (nil), local_pointer = 0x7ff4fdffa168
global_pointer = (nil), local_pointer = 0x400650
global_pointer = (nil), local_pointer = (nil)
global_pointer = (nil), local_pointer = (nil)
global_pointer = (nil), local_pointer = 0x7fffffd2
global_pointer = (nil), local_pointer = 0x7fffffc9
global_pointer = (nil), local_pointer = 0x7fffffcf
global_pointer = (nil), local_pointer = 0x7fffffd2
global_pointer = (nil), local_pointer = 0x7fffffd2
------
global_pointer = (nil), local_pointer = 0x7ffe2d8c4e30
global_pointer = (nil), local_pointer = 0x7ff4fda8381b
global_pointer = (nil), local_pointer = 0x7ff4fda83409
global_pointer = (nil), local_pointer = 0x7ff4fda81bff
global_pointer = (nil), local_pointer = 0x7ff4fdb002c0
global_pointer = (nil), local_pointer = 0x7fffffc9
global_pointer = (nil), local_pointer = 0x7fffffc9
global_pointer = (nil), local_pointer = 0x7fffffc9
global_pointer = (nil), local_pointer = 0x7fffffc9
global_pointer = (nil), local_pointer = 0x7fffffc9
```

8.51

Usage of a Non-Initialized Pointer. Coding Recommendation

- initialize with NULL a pointer, when declared
- assure a pointer is assigned a valid value (i.e. initialized) before using it

```
int* global_pointer = NULL;

int rec_function(int no)
{
    char* local_pointer = NULL;
    ...

    global_pointer = (char*) malloc(sizeof(int));
    if (global != NULL) {
        // use it
    }

    local_pointer = (char*) global_pointer;
    if (local_pointer != NULL) {
        // use it
    }
}
```

8.52

Use After Free

- context
  - **dynamically allocated memory**
  - its address returned

  - got it in a pointer variable
  - **free the allocated memory**, passing the pointer as argument
    * though the **pointer value is not affected**
    * $\Rightarrow$ the previously used memory address still available in program
    * the virtual page the (released) memory was located could be still valid (mapped)
- **vulnerability: use the "*dangling pointer*" as if still valid**
  - lucky case: the program crashes, due to invalid memory
  - unlucky case: the program overwrites other data
- **could be exploited** by attackers to
  - **leak application's secrets**
  - **execute unintended code**

8.53

Use After Free. Example

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

#define MAX_LEN 4096

int main(int argc, char **argv)
{
    char* pointer_1, *pointer_2;

    pointer_1 = malloc(MAX_LEN * sizeof(char));
    printf("pointer_1 = %p\n", pointer_1);

    for(int i=0; i<MAX_LEN; i++)
        pointer_1[i] = 'A';
    printf("pointer_1[0] = %c\n", pointer_1[0]);

    free(pointer_1);

    pointer_2 = malloc(MAX_LEN * sizeof(char));
    printf("pointer_1 = %p, pointer_2 = %p\n", pointer_1, pointer_2);

    for(int i=0; i<MAX_LEN; i++)
        pointer_2[i] = 'B';

    printf("pointer_1[0] = %c\n", pointer_1[0]);

    return 0;
}
```

```
$ gcc -Wall use-after-free.c -o use-after-free.exe

$ ./use-after-free.exe
pointer_1 = 0x8f2010
pointer_1[0] = A
pointer_1 = 0x8f2010, pointer2 = 0x8f2010
pointer_1[0] = B
```

8.54

Use After Free. Coding Recommendation

- initialize to NULL a pointer to a just released memory
- make sure to never reuse a dangling pointer (i.e. pointing to a released memory)

```
char* pointer;

pointer = malloc(...);

...

free(pointer);
pointer = NULL;
```

8.55

Return the Address of a Local Variable

- context
  - **local variables are allocated on the stack**
  - at function call (enter)
  - at function return, the stack space is released and could be reused
- **mistake**: a function **returns the address of a local variable**

- lucky case: stack is not reused when accessing the variable
  - program runs normally
- unlucky case: stack reused, i.e. other functions called
  - $\Rightarrow$ **bugs**, i.e. **program misbehavior**
  - $\Rightarrow$ **vulnerabilities**, like **data leakage**, **unintended code execution**

8.56

Return the Address of a Local Variable. Example

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

#define MAX_LEN 4096

int* compute_power(int base, int exp)
{
    int result;

    result = 1;
    for (int i=0; i<exp; i++)
        result *= base;

    printf("Inside the function: &result = %p, result = 2^10 = %d\n", &result, result);

    return &result;
}

int main(int argc, char **argv)
{
    int* power;

    power = compute_power(2, 10);

    printf("Outside the function: power = %p\n", power);
    printf("Outside the function: 2^10 = %d\n", *power);
}
```

```
gcc -Wall return-as-result-the-address-on-stack.c -o return-as-result-the-address-on-stack.exe
return-as-result-the-address-on-stack.c: In function 'compute_power':
return-as-result-the-address-on-stack.c:16:9: warning: function returns address of local variable [-Wreturn-local-addr]
  return &result;
         ^
```

```
$ ./return-as-result-the-address-on-stack.exe
Inside the function: &result = 0x7fffc7905990, result = 2^10 = 1024
Outside the function: power = (nil)
Segmentation fault (core dumped))
```

8.57

Return the Address of a Local Variable. Coding Recommendation

Never do this!

8.58

Integer Representation Misunderstanding

- a number (**integer**) is **represented** on some number of bytes
- if the integer **variable is local** $\Rightarrow$ its initial value is random
- when using `read()` to load value in variable
  - could read less bytes than those used for representing the entire number
- if **little-endian representation** $\Rightarrow$ first written bytes are **the less significant bytes**
- $\Rightarrow$ not the entire variable's bytes changed (initialized)
- $\Rightarrow$ **random value in variable**

8.59

Integer Representation Misunderstanding. Example

```
void function(int fd)
{
    int var;    // supposed to be represented on 4 bytes

    read(fd, &var, 1); // only the least significat byte changed

    printf("var = %d\n", var); // possibly not the correct / desired value
}
```

8.60

Integer Representation Misunderstanding. Coding Recommendation

```
#include <stdint.h>

void function(int fd)
{
    // solution 1 - works only for little endian
    int var1;    // supposed to be represented on 4 bytes
    var1 = 0;   // initialize to 0 all bytes
    read(fd, &var, 1); // only the least significat byte changed
    printf("var1 = %d\n", var1); // correct value

    // solution 2 - depends in architecture / compiler
    char var2;      // supposes char is on one byte
```

```
    read(fd, &var2, 1);
    printf("var2 = %d\n", var2); // correct value

    // solution 3 - portable (GNU C Library)
    uint8_t var3;      // know precisely it is on one byte
    read(fd, &var3, 1);
    printf("var3 = %d\n", var3); // correct value
}
```


Buffer Overflow

- **vulnerability** (maybe the **most famous**)
  - declare / **allocate a memory buffer of a certain size**
  - **write more space than allocated**
  - ⇒ overwrites other adjacent memory locations, e.g. variables, control data
- lucky case
  - touches addresses in non-mapped pages
  - ⇒ generate page fault and **program crash**
- unlucky case
  - touches addresses only in mapped (so, valid) pages
  - **overwrites other program's data**
  - could generate tricky bugs
- **could be exploited** to
  - reveal application's secrets (**data leakage**)
  - **execute unintended code**
- types
  - stack smashing
  - heap smashing

Buffer Overflow. The "*Classical*" Example

```
#define MAX 10

int main(int argc, char **argv)
{
    char dst[MAX];
    char *src;

    if (argc != 2)
        exit(1);

    src = argv[1];
    strcpy(dst, src);  // vulnerability!
}
```

- a vulnerability when `src`'s contents and length controlled by the user (i.e. attacker)
- ⇒ could copy more than allocated MAX bytes, overwriting the bytes following in memory to "`dst`"

Buffer Overflow. Non Nul-Terminated String Example

```
#define MAX 10

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

    read(fd, buf, MAX);

    int len = strlen(buf);       // vulnerability!
    printf("buf = %s, len = %d\n", buf, len); // vulnerability!
}
```

- "`read`" just reads bytes, not working with nul-terminated strings
- string functions go in memory until finding a 0
- maybe the most encountered mistake in the (first) lab assignment solutions

Buffer Overflow. Another "*Classical*" (off-by-one) Example

```
#define MAX 10

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

    int n = read(fs, buf, MAX);
    buf[n] = '\0'; // vulnerability!

    printf("buf = %s\n", buf);
}
```

- when $n = MAX \Rightarrow$ "`buf[MAX] = 0`" overwrites the next byte after "`buf`"
- due to the good intention to terminate a string with 0 (nul), to be correctly handled by string functions

Buffer Overflow. Non Nul-Terminated String (Subtle) Example

```
#define MAX 10

int main(int argc, char **argv)
{
    char dst[MAX];
    char *src;

    if (argc != 2)
        exit(1);

    src = argv[1];
    strncpy(dst, src, MAX);  // vulnerability!
    printf("dst = %s\n", dst);
}
```

- does not terminate "`dst`" with 0 when "`strlen(src) >= MAX`"
- best solution: `strcpy_s`

Buffer Overflow. Coding Recommendation

- use safe functions for string manipulation
  - e.g. `strncpy`, `strcpy_s` instead of `strcpy`
- take care of string functions, read carefully their specification
- check correctly buffer limits

```
#define MAX 10

int main(int argc, char **argv)
{
    char dst[MAX];
    char *src;

    if (argc != 2)
        exit(1);

    src = argv[1];
    int err = strcpy_s(dst, MAX, src);
    if (err != 0) {
        // error
        exit(1);
    }
    // dst always nul-terminated and not overflowed
    printf("dst = %s\n", dst);
}
```

Buffer Overflow. Solutions

- fundamental: **write correct (secure) code**
  - most effective (**remove the cause**)
  - **do not trust user-provided data**, but **validate it**
- **compiler-based** (**mitigate effects**)
  - stack canaries
  - control-flow integrity
- **OS-based** (**mitigate effects**)
  - process **isolation**
  - address space layout randomization (**ASLR**)
  - no-execution pages / data execution prevention (**DEP**)
- **hardware-based** (**mitigate effects**)
  - Intel MPX
  - needs compiler support

8.68

More Information

- vulnerability description
  - http://cwe.mitre.org
  - http://cwe.mitre.org/data/slices/2000.html
  - http://cwe.mitre.org/top25/index.html
- vulnerability databases
  - http://www.cvedetails.com/
  - http://cve.mitre.org/
  - http://www.securityfocus.com/
  - https://www.exploit-db.com/

8.69

4 Conclusions

What we talked about

- *virtual address space* (VAS) vs. *physical address space* (PAS)
  - virtual address (VA) vs physical address (PA)
- binding (translating) VAs to PAa
  - *compile time* vs *load time* vs *run time*
- *contiguous allocation*
  - base and limit
  - + simple
  - - rigid, external fragmentation, no support for sparse VAS
- *segmentation*
  - more contiguous areas, each with its base and limit
  - + supports sparse VAS
  - - external fragmentation
- *paging*
  - uniform, fixed-size allocation units, i.e. pages
  - + supports sparse VAS, fine-grained memory sharing, no external fragmentation
  - - internal fragmentation

8.70

What we talked about (cont.)

- *swapping*
- memory related-*vulnerabilities*

8.71

Lessons Learned

1. multiprogramming implies usage of virtual memory addresses (VAs)
2. low-level access to memory, like C allows, is both useful and dangerous

8.72

Chapter 10

Review and Conclusions

What have you learned? What else could you still learn?

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

June 2, 2021

10.1

Purpose and Contents

The purpose of this chapter

1. review subjects presented during the OS course (this semester)
2. draw some conclusions
3. presents future OS-related courses

10.2

Contents

10.3

1 OS Subject Review

OS Definition, Role, Architecture

- system software placed between hardware and user software (applications)
- roles
  - virtual machine provider (hide hardware and provide abstractions)
  - resource manager
- every resource access / need must be required from OS
  - by calling SO services, i.e. system calls
- OS protects itself from user applications based on different CPU execution modes
  - privileged: kernel mode
  - non-privileged: user mode
- architectures
  - monolith
  - micro-kernel

10.4

Practice: OS Definition Related Questions

- When is the OS code executed on an uni-processor system? Give examples of at least two situations.
- How is the system call mechanism implemented?
- Which software is run in kernel and user mode respectively in the following two cases?
  - monolithic OS
  - micro-kernel OS

10.5

OS Shell

- provides the user interface to the OS
- usually a simple application
- takes user requests and translated them in system calls
- two types
  - graphical
  - text, named command interpreter
- command interpreter functionality
  - get user's command line
  - split it in tokens, i.e. command line and arguments
  - create a new process to execute the specified command
  - waits for created child process' termination
- command line
  - a string of characters separated by spaces
  - first item: command name, actually an executable path
    - searched in directories from PATH
    - security: trust user-established environment (e.g. PATH)
  - other items: command line arguments
  - special characters, like STDIN/OUT redirection, pipe etc.

10.6

Practice: Shell Related Questions

Which is the effect of the following commands?

- ls > file
- read n < file
- ls -R / 1>good 2>err
- cat dict.txt | sort

10.7

File System (FS)

- file concept
  - basic unit of data allocation
  - unstructured stream of bytes
  - file contents managed by user applications, not by OS
  - components: data and meta-data
  - security: too much permissions vulnerability
- directory concept
  - used for organizing the file system space
  - impose file system hierarchy
  - paths: absolute and relative
  - security: path traversal vulnerability

- FS system calls
  - file: open, read, write, lseek, close
  - directory: opendir, readdir, stat, unlink, link
- allocation aspects
  - contiguous allocation ⇒ external fragmentation
  - any-free-block allocation ⇒ internal fragmentation
  - i-nodes, directory entries, links, files with holes

10.8

Practice: FS Related Questions

- How is usually the file provided like to the user applications?
- Which of the following extensions usually correspond to text and binary files respectively: html, pdf, c, zip?
- What is an i-node in Linux?
- What does 0640 means in terms of permission rights in Linux?
- Which is the most probable file descriptor returned (and displayed) by the following Linux program? Explain your answer.

```
main()
{
    int fd = open ("/etc/passwd", O_RDONLY);
    printf("fd = %d\n", fd);
}
```

- Write in one line the C code to read a text line from a file, whose file descriptor is given.
- Write the C code to read an integer from offset 16 from a file, whose file descriptor is given.

10.9

Process and Thread Management

- process
  - models execution: abstractizes the machine (CPU, memory)
  - describe execution and resources needed for that execution
  - isolates (separates) resources / executions
  - states: running, ready, blocked, terminated
- thread
  - models execution in a process
  - more threads = more concurrent executions in the same process
  - threads of a process share all resources of that process
  - threads useful and effective when
    - logical parallelism exist in the application
    - enough hardware resources available
- scheduling
  - decides who runs and for how long
  - preemptive vs. non-preemptive
    - preemptiveness based on timer interrupt
  - first-come first-served (FCFS), shortest job first (SJB), round-robin (RR), priority-based

10.10

Practice: Process Related Questions

- How many times is each message in the code below displayed on the screen? Explain your answer.

```
int fd[2];
pipe(fd);
printf("Step 1\n");
fork();
fork();
fork();
printf("Step 2\n");
```

- How many readers and writers, respectively, will exist in the system for the created pipe after the execution of the given code, supposing no process is terminated at that moment?
- Which is the optimum number of threads that should be created by an application to get the best performance (i.e. execution time) when running on a uni-process system and copying a file from one disk to another disk, by encrypting the file contents during the copy operation?

Synchronization Mechanisms

- problem: race conditions of concurrent threads sharing resources
- synchronization means imposing access rules
  - leads to waiting (blocking) $\Rightarrow$ reduce parallelism
- synchronization needs OS and hardware support to get atomicity
- synchronization mechanisms
  - lock $\Rightarrow$ mutual exclusion
  - semaphore
    - generalized lock
    - event counter
  - condition variable
    - specialized waiting mechanism in mutual exclusion area
- classical patterns
  - producers-consumers, readers-writers, rendez-vous (barrier)

Practice: Synchronization Related Questions

- Use semaphores to allow just 10 threads run simultaneously a given function's body.
- Rendezvous: synchronize threads executing functions `boy()` and `girl()` respectively, such that to allow them returning from that functions only in pairs of a "boy" and a "girl".
- Synchronize two concurrent threads executing the two functions below respectively, such that to make them display on the screen the message "*Life is wonderful, isn't is?*"

```
thread_1()
{
    printf("Life ");
    printf("wonderful, ");
}
```

```
thread_2()
{
    printf("is ");
    printf("isn't it?");
}
```


Memory Management

- memory addresses: physical vs. virtual
  - address space
  - virtual address space structure: code, data, heap, stack
- memory binding / translation
  - compile-time (very limited), load-time, run-time (most flexible)
  - need translation tables
- contiguous allocation
  - simple, efficient
  - leads to external fragmentation

  - base and limit registers
- segmentation
  - one contiguous area for each segment (area)
  - segment table
- paging
  - allocates memory in fixed-size chunks ⇒ internal fragmentation
  - virtual pages and physical frames
  - page tables and page table entries
  - page sharing, memory mapped files

10.14

Practice: Memory Mng Related Questions

- Which is the size in bytes of a process' virtual address space on a system using 64 bits for a memory address?
- How may page table entries must be used by such a system to map a process VAS, supposing the page size to be 4MB?
- Illustrate on such a system the part of a process' VAS and page table (entries) used for mapping the memory required be the following code:

```
unsigned int *p = malloc(4*4*1024*1024);
printf("p=%u\n", p);      // displays p = 1000*4*1024*1024
```

- Which page does the following instruction refer to? Could it be executed successfully or not?

```
p[4*1024*1024] = 10;
```

10.15

Security Aspects

- untrusted application environment
  - untrusted PATH variable
- file system
  - too much permissions
  - path traversal
- memory-related: bad / wrong memory accesses
  - NULL-pointer usage
  - use-after free
  - buffer overflow

10.16

2 Conclusions

2.1 Past

What did we talk about?

- OS's place and role (and definition), relative to hardware and other software
- OS structure
- Shell, i.e. command interpreter
- File System
- Process Management
- Memory Management
- Security Aspects

10.17

What have we learned?

- basic concepts like
  - process, thread — execution, resources, isolation, scheduling
  - file: storage, sequence of bytes (no format), meta-data (i-node), links, fragmentation, open files
  - directory: organization, file tree, collection of directory entries
  - synchronization: locks (mutex), semaphores, condition variables, producers/consumers, readers/writers, barrier
  - memory: address space, virtual/physical memory addresses, ELF, paging
  - IPC mechanisms: pipes, shared memory
  - fragmentation: external (contiguous allocation, best-fit, worst-fit), internal
  - basic security issues: buffer overflow, path traversal
- system calls to access various OS (Linux) services
- write C programs to have access to Linux system calls

10.18

What can we do?

- explain OS functionality and concepts
- understand better applications functionality, their relationship with the OS and some of their crash reasons
- write (if needed) C programs to access low-level OS services
- choose the appropriate OS for a particular purpose
- tune better an OS for particular applications/context/purposes

10.19

2.2 Future

Will I need OS knowledge?

- all the time: understand, explain, evaluate, configure

10.20

Will I use OS system calls?

- sometimes, especially when you need a particular (efficient, lower-level) functionality
- all the time, if you work at low-level (in C, asm)

10.21

Operating System Design

- this is (our) next step: the advanced course (UTCN, CS Dept.) regarding OS
- what about: internals of an OS, design/implementation alternatives of
  - scheduling algorithms and synchronization mechanisms
  - user processes and threads
  - system calls for open files, processes and threads
  - memory management, virtual memory, page replacement algorithms
  - file system
- practical aspects: design, implement (in C) and test an OS (*HAL9000* — UTCN; *Pintos* — Stanford, USA); learn how to debug an OS on a remote virtual machine
- practical aspects: work in team (3 members)

10.22

Operating System Design (cont.)

- usefulness
  - an OS is a complex management software; all the management algorithms and techniques work for many other complex management software even at higher levels
  - understanding low-level mechanisms makes you understand better higher-level ones' behavior
- it is an 4th year *optional subject*
- *students' myth*: (highly) difficult
- *reality (trust me)*: just interesting and challenging, not more difficult than other subjects/projects

10.23

Operating System Design (cont.)

- I would like you to think like this:
  - "I'd choose this subject just because I am really interested in, it sounds great and challenging, I feel I need it in my future career, I want to understand how low-level mechanisms and aspects work, it fits my aptitudes and interests, I have heard about learning useful things etc."
- I would not like to hear about you saying:
  - "I DID NOT choose that subject just because I have a job and do not have enough time for it, I've heard it takes long(er) to solve the assignments, I've heard it is (more) difficult etc."

10.24

Security Master Program (SISC)

- many OS-related courses
  - build low-level OS layers on the 64-bit architecture
  - kernel driver development
  - build virtualization security-oriented OS (hypervisor)
- many security-related aspects
  - secure coding
  - Web security
  - mobile systems (Android) security
  - big-data and security
  - penetration testing
  - risk management
  - cryptography

10.25

Finish

That's all! So long, folks!

10.26