System Security

The Boring Introduction

Mohamed Sabt

Univ Rennes, CNRS, IRISA

2024 / 2025

Part I

Finding Inside the Shell

Reverse-i Search

- Press `Ctrl+R`. This will start the reverse-i search.
- Type the string you want to search for.
- If the command shown is not the one you want, press `Ctrl+R` again to move to the next matching command.
- Once you've found the command you want, press `Enter` to execute it.

Shell Globbing 1/2

- The shell substitutes arguments containing globs to filenames.
  - The substitution is called expansion because the shell substitutes all matching filenames for a simplified expression.

- Examples:
  - at* expands to all filenames that start with at.
  - *at expands to all filenames that end with at.
  - *at* expands to all filenames that contain at.

- Another shell glob character, the question mark ?, instructs the shell to match exactly one arbitrary character.
  - Example: b?at matches brat and boat.

Shell Globbing 2/2

- If you don’t want globbing, use quotes.

- If no file matches a glob, the bash shell performs no expansion, and the command runs with literal characters such as *.

- It is important to remember that the shell performs expansion before running commands, and only then.
  - Therefore, if a * makes it to a command without expanding, the shell won’t do anything more with it.

Find 1/3

- Goal: search for files.
- General syntax:
  - find (starting directory) (matching criteria and actions)

- Important matching criteria:
  - -name nam (nam with globbing syntax)
  - -regex nam (nam with regex syntax)
  - -path pth (pth is the path)
  - -type c (c is for file type: f, d)
  - -user usr (file's owner is usr)
  - -group grp (file's group owner is grp)
  - -perm p (the files access mode is exactly p) ; variants -p or /p
  - -size n (file is n blocks big; a block is 512 bytes)

Find 2/3

- Chaining matching criteria
  - ! for negating
  - (expr) force precedence
  - -o for or
  - , for both (not and)

- Important actions:
  - -print (self-explaining)
  - -delete (self-explaining)
  - -ls (self-explaining)
  - -prune (if the file is a directory, do not descend into it)
  - -exec cmd ({} \; vs {} +)
  - -ok cmd (ask the user first)
  - -execdir cmd (run from the subdirectory containing the matched file)

Find 3/3

- Example 1
  - find /home -path /home/sabt -prune -o -name ‘*.c’ -print
- Example 2
  - sudo find /sbin /usr/sbin -executable \! -readable –print
- Example 3
  - find . -perm -444 -perm /222 \! -perm /111

Grep 1/2

- Goal: search for patterns inside files.
- General syntax:
  - grep [OPTIONS] PATTERN [FILE...]
  - PATTERN is a regex pattern defining what we want to find in the content of the files specified by the FILE argument.

- Important (Basic) Patterns:
  - []: any single character inside the list.
  - [^]: any character not in the list.
  - ^: beginning of line
  - $: end of line.
  - .: any single character
  - *: zero or many occurrences of the same previous character.

Grep 2/2

- Important (Extended) Patterns:
  - ?: zero or one occurrence of the same previous character.
  - +: one or many occurrences of the same previous character.
  - {}: intervals
  - |: or operator
  - (): grouping

- Example 1
  - grep ‘the’ file.txt

- Example 2
  - grep ‘[wv]ine’ file.txt

- Example 3
  - ls | grep ‘\.c$’

Sed 1/2

- Goal: stream editor.

- General syntax:
  - sed [OPTIONS] SCRIPT FILE
  - the SCRIPT argument is the sed script that will be executed on every line for the files that are specified by the FILE argument.

- Script structure: [addr] X
  - addr is the condition applied to the lines of the text file.
  - The X character represents the sed command to execute.

Sed 2/2

- Addr:
  - Fixed number or an interval: first,last
  - Regex pattern: /regex/
  - first~step
  - first,+N: matches first and the N lines following first (N + 1 lines).

- Commands:
  - p: print
  - d: delete
  - s/pattern/replacement

- Examples:
  - sed ‘1,+5 d’ file.txt
  - sed ‘s/hello world/bonjour le monde’ file.txt

Awk 1/2

- Goal: full-fledged programming language that is comparable to Perl.

- General syntax:
  - awk [options] script file
  - Script structure: '(pattern){action}+’

- Pattern:
  - The pattern is a regex pattern that will be tested against every input line.
  - If a line matches the pattern, awk will then execute the script defined in action on that line.
  - If the pattern condition is absent, the action will be executed on every line.
  - Special patterns: BEGIN, END

Awk 2/2

- Actions:
  - print
  - Special variables: $0, $1, $2, etc.

- Example 1:
  - awk ‘/hello/{print $0}’

- Example 2:
  - ls -l | awk '/\.c$/{count++} END {print count}'

Part II

Introduction to OS

Diversity of Operating Systems

Computer System Components

- Hardware – provides basic computing resources
  - CPU, memory, I/O devices.

- Operating System
  - Controls and coordinates use of hardware among various applications and users.

- Application programs – define the ways in which the system resources are used to solve users computing problems.
  - Word processors, compilers, web browsers, video games.

- Users
  - People, machines, other computers.

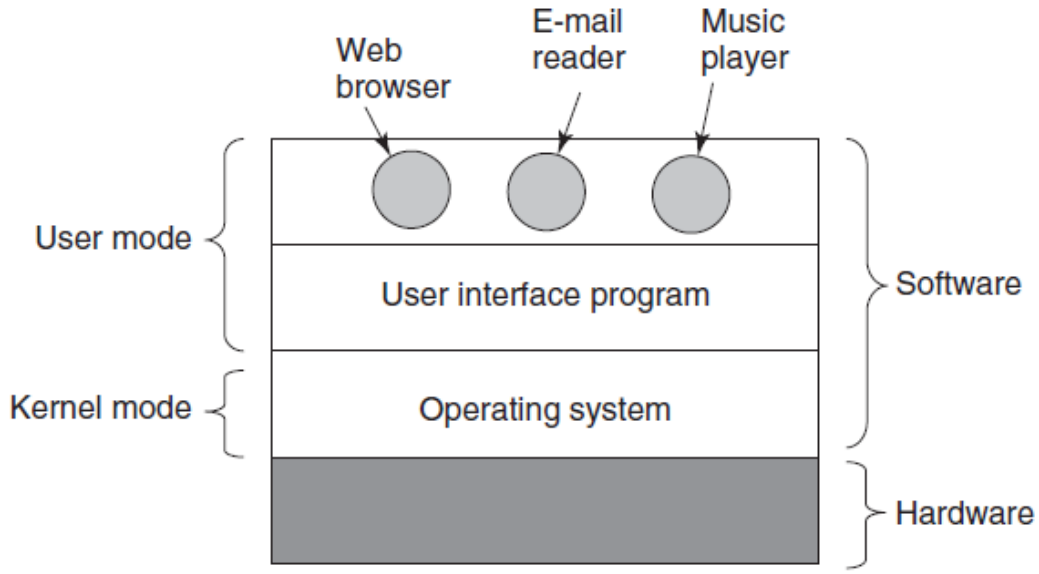
What is an Operating System?

- Operating System
  - is a system software which acts as an intermediary between user and hardware.
  - keeps the complicated details of the hardware hidden from the user and provides user with easy and simple interface.
  - performs functions which involves allocation of resources efficiently between user program, file system, Input Output (IO) device.

- Operating system goals:
  - Execute user programs.
  - Make the computer convenient to use.
  - Use the computer hardware in an efficient manner.

Abstract View of Operating System

- The hardware components lie at the bottom of the diagram. It is considered as The most crucial part of computer system



- Operating system runs in the kernel mode of the system wherein the OS gets an access to all hardware and can execute all machine instructions. Other part of the system runs in user mode.

The OS as an Extended Machine

- The structure of computers system at the machine-language level is complicated to program, especially for IO.
  - Programmers don’t deal with hardware, so a level of abstraction is supposed to be maintained.
  - Abstraction allows a program to create, write, and read files, without having to deal with the messy details of how the hardware actually works

- Good abstractions turn a nearly impossible task into two manageable ones.
  - The first is defining and implementing the abstractions.
  - The second is using these abstractions to solve the problem at hand.

The OS as a Resource Manager

- The operating system provides for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs.
- Resource management includes multiplexing (sharing) resources in two different ways: in time and in space.
- In time multiplexed, different programs take turns using CPU. First one of them gets to use the resource, then the another, and so on.
- In space multiplexing, instead of programs taking turns, each one gets part of the resource.

Design Goals 1/2

- Abstraction
  - Make the system convenient to use.

- Performance
  - Minimize the overheads: both time and space.

- Protection
  - Isolating processes from one another.

Design Goals 2/2

- Reliability
  - Run without crash.
- Security
  - An extension of protection.
  - Against malicious applications.
- Energy-efficiency.
- Portability.

Jargon: Mechanisms and Policies

- In many operating systems, a common design paradigm is to separate high-level intelligence from their low-level machinery.

- We call the low-level machinery mechanisms;
  - **Mechanisms** are low-level methods or protocols that implement a needed piece of functionality.

- On top of these mechanisms resides some of the intelligence in the OS, in the form of policies.
  - **Policies** are algorithms for making some kind of decision within the OS.

Separate Policy and Mechanism

- You can think of the mechanism as providing the answer to a **how** question about a system;
  - for example, how does an operating system perform time multiplexing?

- The policy provides the answer to a **which** question;
  - for example, which process should the operating system run right now?

- Separating the two allows one easily to change policies without having to rethink the mechanism,
  - and is thus a form of **modularity**, a general system design principle.

Part III

Brief History

Early Operating Systems

- In the beginning, the operating system was just a set of libraries of commonly-used functions, and thus made life easier for developers.
  - for example, instead of having each programmer of the system write low-level I/O handling code, the “OS” would provide such APIs.

- Computers, as of that point, were not used in an interactive manner.
  - **batch processing**, as a number of programs (or jobs) were set up and then run in groups, called a “batch”, by a human operator.

- Computers were known as mainframes and were kept in separate rooms.

The Era of Multiprogramming

- Multiprogramming became commonplace due to the desire to make better use of machine resources.

- Instead of just running one job at a time, the OS would load a number of jobs into memory and switch rapidly between them, thus improving CPU utilization.
  - This switching was particularly important because I/O devices were slow; having a program wait on the CPU while its I/O was being serviced was a waste of CPU time.

- **Memory protection** became important; we wouldn't want one program to be able to access the memory of another program.

The Advent of UNIX

- One of the major practical advances of the time was the introduction of the UNIX operating system.
  - UNIX took many good ideas from different operating systems (particularly from Multics from MIT), and made them simple.

- The UNIX environment was friendly for programmers and developers alike, also providing a compiler for the new C programming language.

- The spread of UNIX was slowed a bit as companies tried to assert ownership and profit from it.
  - Many companies had their own variants: **SunOS** from SunMicrosystems, **AIX** from IBM, etc.
  - The legal wrangling cast a dark cloud over UNIX, while Windows was introduced and took over much of the PC market.

The ACM Turing Award (UNIX)

- ACM is the association for Computing Machinery.
  - World's largest educational and scientific computer society.
  - You can become a student member too ***www.acm.org***.

- The ACM awards the Turing Award every year.

It is the "Nobel Prize" of computing.

- Names after British mathematician

Alain M. Turing (1912 -- 1954).

The Modern Era

- Beyond the minicomputer came a new type of machines: cheaper, faster, and for the masses:
  - the personal computer, or **PC** as we call it today.

- Led by Apple's early machines (e.g., the Apple II) and the IBM PC, this new breed of machine would soon become the dominant force in computing,
  - as their low-cost enabled one machine per desktop instead of a shared minicomputer per workgroup.

It was Better Before!

- Unfortunately, for operating systems, the PC at first represented a great leap backwards, as early systems forgot (or never knew of) the lessons learned in the era of minicomputers.

- **DOS** (the **Disk Operating System**, from **Microsoft**) didn't think memory protection was important;
  - thus, a malicious (or perhaps just a poorly-programmed) application could scribble all over memory.

- The first generations of the **Mac OS** (v9 and earlier) took a cooperative approach to job scheduling;
  - thus, a thread that accidentally got stuck in an infinite loop could take over the entire system, forcing a reboot.

And Then Came Linux

- Fortunately for UNIX, a young Finnish hacker named **Linus Torvalds** decided to write his own version of UNIX, but not from the code base, thus avoiding legal issues.
  - He took advantage of the sophisticated GNU tools that already existed

- As the internet era came into place, most companies (such as Google, Amazon, Facebook, and others) chose to run Linux.

- Steve Jobs took his UNIX-based **NeXTStep** operating environment with him to Apple, thus making UNIX popular on desktops

- As smartphones became a dominant user-facing platform, Linux found a stronghold there too (via Android).

Part IV

Use-Case
Virtualizing Memory

Memory and Programs

- Memory is just an array of bytes;
  - to **read** memory, one must specify an **address** to be able to access the data stored there;
  - to **write** (or **update**) memory, one must also specify the data to be written to the given address.

- A program keeps all of its data structures in memory, and accesses them through various instructions,
  - like loads and stores or other explicit instructions that access memory in doing their work.

- Don’t forget that each instruction of the program is in memory too; thus memory is accessed on each instruction fetch.

Simple Example 1/2

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main(int argc, char *argv[]) {
    int * p = malloc(sizeof(int)); // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n", getpid(), p); // a2
    *p = 0; // a3

    while (1) {
        sleep(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p); // a4
    }
    return 0;
}
```

Simple Example 2/2

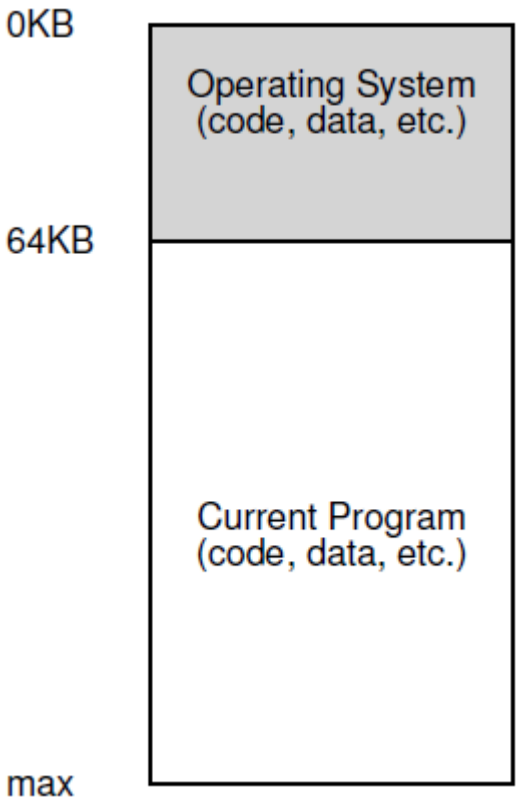
```
prompt> ./mem &   ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Example Explained

- We run multiple instances of this program to see what happens.
- We see from the example that each running program has allocated memory at the same address, and yet each seems to be updating the value independently!
  - It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs.
- Each process accesses its own private **virtual address space** (sometimes just called its **address space**), which the OS somehow maps onto the physical memory of the machine.
- A memory address within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned.
  - The reality, however, is that physical memory is a shared resource, managed by the operating system.

Memory in Early Systems

- From the perspective of memory, early machines didn't provide much of an abstraction to users. Basically, the physical memory of the looked something like what you see in Figure
  - The OS was a set of routines (a library, really) that sat in memory (starting at physical address 0 in this example), and there would be one running program (a process) that currently sat in physical memory (starting at physical address 64k in this example) and used the rest of memory.



Multiprogramming and Time Sharing

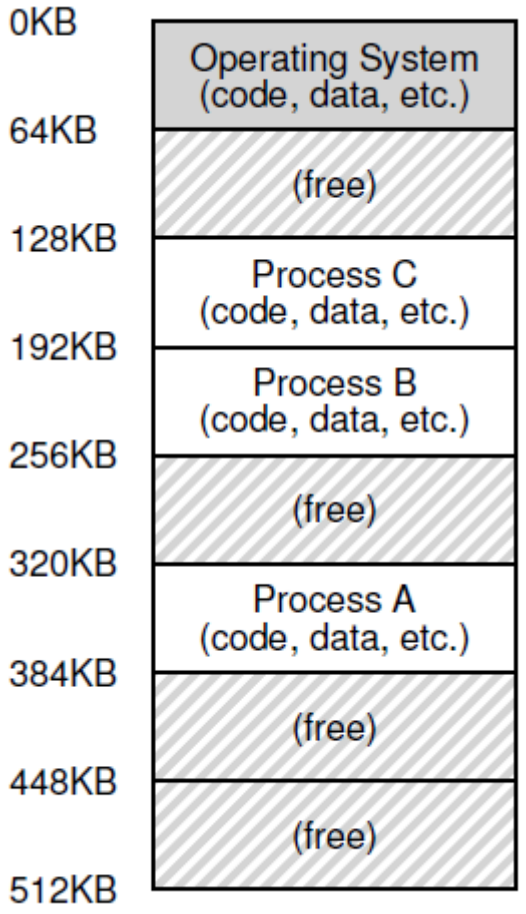
- Later, because machines were expensive, the era of **multiprogramming** was born.
  - in which multiple processes were ready to run at a given time, and the OS would switch between them.

- Soon enough, however, people began demanding more of machines, and the era of **time sharing** was born.

Sharing Memory

- In the diagram, there are three processes (A, B, and C) and each of them have a small part of the 512KB physical memory carved out for them.
  - Assuming a single CPU, the OS chooses to run one of the processes (say A), while the others (B and C) sit in the ready queue waiting to run.

- Multiple programs to reside concurrently in memory makes **protection** an important issue; you don't want a process to be able to read, or worse, write some other process's memory.



Failing Approach

- One way to implement time sharing would be to
  - run one process for a short while, giving it full access to all memory,
  - then stop it, save all of its state to some kind of disk (including all of physical memory),
  - load some other process's state, run it for a while, etc.

- Unfortunately, this approach has a big problem: it is way too slow, particularly as memory grows.

The Principle of Isolation

- Operating systems strive to isolate processes from each other and in this way prevent one from harming the other.
  - Isolation is a key principle in building reliable systems.

- By using memory isolation, the OS further ensures that running programs cannot affect the operation of the underlying OS.
  - When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of any other process or the OS itself

- Some modern OS's take isolation even further, by walling off pieces of the OS from other pieces of the OS.
  - Such **microkernels** thus may provide greater reliability than typical monolithic kernel designs.

The Address Space

- The address space of a process contains all of the memory state of the running program.
  - Code.
  - Stack: for temporary execution state.
  - Heap: dynamically-allocated, user-managed memory.

- We say the OS is **virtualizing memory**, because the running program thinks it is loaded into memory at a particular address (say 0) and has a potentially very large address space.

- When, for example, process A tries to perform a load at address 0, somehow the OS, in tandem with some hardware support, will have to make sure the load doesn't actually go to physical address 0 but rather to physical address 320KB (where A is loaded into memory).