Processes *

Jens Lechtenbörger

IT Systems, Summer Term 2024

This presentation is the final one on OS topics. It explains the process concept as central abstraction for resource management of running programs, where the OS isolates different processes from each other. Importantly, the process concepts combines major topics discussed so far: In particular, processes contain threads as units of OS scheduling, which cooperate and share resources, e.g., virtual memory and files.

1 Introduction

Let us look at essential questions and terminology of our topic.

1.1 Core Questions

- What is a process?
- How are files represented by the OS, and how are they used for inter-process communication?

(Based on Chapter 7 and Section 8.3 of (Hailperin 2019))
This presentation addresses the following questions:
What is a process?
How are files represented by the OS, and how are they used for inter-process communication?

1.2 Learning Objectives

- Explain process and thread concept
- Perform simple tasks in Bash (continued)
  - View directories and files, build pipelines, redirect in- or output, list processes with `ps`
- Explain access control, access matrix, and ACLs
  - Use `chmod` to modify file permissions

Take some time to think about the learning objectives specified here.

1.3 Retrieval practice

Let us see what prior knowledge is involved subsequently.

*This PDF document is an inferior version of an OER HTML page; free/libre Org mode source repository.

1.3.1 Previously on OS ...

- What are processes and threads?
- What is a thread control block?
- What are system calls?
- How to execute shell commands as part of The Command Line Murders?

Before you continue, answer the questions listed here, ideally, without outside help.

1.3.2 Quiz 1

Take this quiz.

1.3.3 Quiz 2

Take this quiz.

1.3.4 Quiz 3

Take this quiz.

Agenda

The agenda for the remainder of this presentation is as follows.

We revisit the process concept in OSs and provide more details. Then we revisit file I/O commands that you know from the command line murders and explain underlying mechanisms in terms of file descriptors. In addition, we discuss some aspects of access rights in OSs, with a focus on file permissions.

2 Processes

Warning! External figure **not** included: "/proc" © 2018 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission. (See HTML presentation instead.)

The Linux kernel exports various pieces of management information in the special directory `/proc`, which is a great place to explore what is happening behind the scenes. This drawing illustrates some pieces of information offered about processes in that directory, which is revisited several times later on.

2.1 Processes

- First approximation: Process ≈ **program in execution**
  - However
    - Single program can create multiple processes
      - E.g., web browser with process per tab model
    - What looks like a separate program may not live inside its own process
      - E.g., separate GNU Emacs window showing PDF file via PDF Tools

            - (Window contents might be produced with help of different process, though)

Processes are central management units of OSs. As you already know, you can think of a process as a program in execution. E.g., if you open an app on your phone, this app is usually managed as a process by the OS. Also, if you use a command line (as in The Command Line Murders), the command line itself is one process (whose instructions are executed in the context of a virtual address space), while commands such as `grep` lead to the creation of new processes (with their own instructions and address spaces).

However, as you have seen already, the picture is more complicated as some "apps" may really be managed with multiple processes by the OS, while also a single process may provide functionality that looks like multiple "apps".

- Reality: Process = Whatever your OS defines as such
  - Unit of **management** and **protection**
    - One or more threads of execution
    - Address space in virtual memory, shared by threads within process
    - Management information in process control blocks
      - Access rights
      - Resource allocation
      - Miscellaneous context

Ultimately, a process is whatever your OS defines to be a process. In particular, each process is associated with one or more threads to execute instructions and a single virtual address space that is shared by its threads and isolated from the address spaces of other processes (and their threads).

Similarly to the use of thread control blocks to record management information for threads, the OS uses a process control block for each process, where it next to other details keeps tracks of resources used by the process (and its threads). We will in particular look at the management of files with file descriptors and access rights, and we will do so via examples of GNU/Linux.

2.2 Process Creation

- OS starts
  - Check your OS's tool of choice to inspect processes after boot

Your operating system starts lots of processes during boot. It starts more when you log in.

Check out your system: Which processes are present? Which ones seem to be legitimate?
On Linux (and Mac OS), ps is a standard tool, to be revisited subsequently.
On Windows, process explorer is a powerful alternative to the task manager.

- User starts program
  - Touch, click, type
- Processes start other processes
  - POSIX standard with Process management API; see (Hailperin 2019)

Users start further processes, depending on the device and OS by touching, clicking, or typing. In fact, with their interactions, users really ask some process, which implements a user interface, to start further processes. That process in turn uses system calls for process creation.

Also without user interaction, processes may start other processes, e.g., a browser with a "process per tab" model. While details of underlying APIs are not important for us, note that POSIX is a standard aiming for compatibility between OSs.

This standard specifies an API for process management, and it also specifies details about file access to be addressed subsequently.

2.2.1 Bash as Command Line

- Recall: Command line as interface to OS to execute processes
  - Unix command line historically called "shell"
    - Command line itself is a process
    - Lots of shell variants; Bash from The Command Line Murders used here

A command line, or shell, runs as process and provides a textual user interface, with which more processes can be executed. We only consider the Bash in this course.

  - Command line can (1) execute builtin commands and (2) create processes for other commands
    1. Builtin commands are executed internally
       - Type `help` to execute one and see all of them

Importantly, not every typed command leads to the creation of a new process. Instead, every shell offers a set of builtin commands, which are executed internally, as part of the shell's process. Execute the builtin command `help` to learn more.

    1. Programs are executed as new child processes (requires system calls)
       - E.g., `cat`, `grep`, `less`, `man`, `ps`
       - By default, while child process for program runs, process of bash waits (not on CPU but blocked) for return value of child

In addition, every operating system comes with programs, and those programs can also be executed from the shell, to be managed as processes by the OS. In GNU/Linux, we expect a standard set of preinstalled utility programs, e.g., the ones listed here, but also text editors, web browsers, games, etc.

If you execute a program, then by default the process of the command line is blocked, waiting for a return value from a new child process that runs the program.

2.3 Process Control Block

- Similarly to thread control blocks the OS manages **process control blocks** for processes
  - Numerical IDs (e.g., own and parent)
  - Address space
  - Resources (shared by threads)
    - E.g., file descriptors discussed next

  - Security information
    * Related to access rights
  - And more, beyond course, e.g.:
    * Interprocess communication with signals

Similarly to thread control blocks for threads, the OS manages process control blocks for processes.

Typical pieces of information are listed here, including numerical identifiers for the process itself and its parent process; management information for virtual memory, for resources such as files, to manage access rights, and signals for communication between processes.

Note that the process control block collects information that is shared by all threads of the process, while thread control blocks are about information that is specific to each thread.

Of course, implementations across OSs differ considerably. Beyond class goals, the Linux kernel, for example, uses the same structure for process control blocks and thread control blocks, where process information shown on this slide is only stored once in memory. Such memory areas are then referenced with pointers from all thread control blocks that share the same information.

2.3.1 Seeing Processes and Threads on Linux

- Linux kernel offers `/proc` (drawing) (man page offers details)
  - Pseudo-filesystem as interface to Linux kernel data structures
    On Linux systems, the directory `/proc` is a so-called pseudo-filesystem. It looks and feels like any other directory, but it does not contain "real" files. Instead, it exports kernel management data through the interface of a filesystem.
    On other systems, this directory may be missing or be incomplete.
    * Subdirectories per process ID (e.g., `/proc/42`) with details of process control block for process with ID 42
      For example, every process and every thread has a numeric identifier, ID for short, and you find a subdirectory under `/proc` for these numbers. In turn, files and directories in that subdirectory show management information including details of process and thread control blocks.
      On a side note, the ID assigned to the first thread of a process is equal to the process ID.
  - Process listing command `ps` inspects `/proc`
    * (Use `man ps` for implementation-specific details, following options are for GNU/Linux)
    * `ps -e` shows some details on all processes (IDs, time, etc.)
      · Option `-L` adds thread information, option `-f` for "full format", e.g.: `ps -eLf`
      · ("L" for "light weight process", a synonym for thread; column `LWP` shows thread IDs)
      On GNU/Linux, the command `ps` lists running processes, and it does so by inspecting the directory `/proc`. As usual, the manual page contains all details about the command. In particular, it uses "light weight process" as synonym for "thread".
      By default, threads are not shown in `ps` output, but you need an extra option as suggested on the slide.
    * `ps -C <name>` shows some details on all processes with the given name
      Depending on the implementation of `ps`, you may also be able to display only information related to processes with a known command name as shown here.

If that option is not available in your implementation, you can pipe the output of `ps` to `grep`. In the `grep` output, column headers are filtered out. Thus, check without `grep` first, and take note of column headers. (E.g., for Cygwin or Mac OS.)

- Other OSs come with their own tools

Of course, other OSs come with their own tools.

3 File Descriptors

- Recall The Command Line Murders

1. `cd clmystery/mystery`
2. `head crimescene | grep Alice`
   - `crimescene` ⇝ `head` ⇝ `grep` ⇝ `console output`
3. `head crimescene > first10lines`
   `grep Alice < first10lines`
   - `crimescene` ⇝ `head` ⇝ `first10lines` ⇝ `grep` ⇝ `console output`

- (Files are covered in Section 8.3 of (Hailperin 2019))

Files are a common OS abstraction to organize data as named streams of bytes that are stored persistently. Persistence means that files should keep their data beyond crashes and power failures.

(As a side note, we can also create file systems in RAM, whose contents are forgotten, when power is turned off...)

As you experienced in the context of The Command Line Murders, file systems provide a hierarchically organized name space, where files are located in directories. (In fact, directories are just files with special properties, but we do not go into details here.)

In the context of The Command Line Murders, you saw that files can serve as inputs and outputs of processes and that one process can communicate its output with the pipe symbol as input to another process. Thus, you are able to explain how the commands shown under items (2) and (3) here produce the same final output. The bullet points underneath the commands are supposed to visualize the flow of data with squiggly arrows, where files are shown in blue, commands in black.

(Please think about differences and commonalities: Both create one process for `head` and one for `grep`. One requires more typing than the other. One requires the creation of an additional file, which is left around and may or may not be relevant afterwards...)

3.1 File Descriptors

- OS represents open files via **integer numbers** called **file descriptors**

  Processes invoke system calls to open files and to work on files' contents. The OS represents each file opened by a process with a file descriptor, which is just an integer number that is returned from the system call that opened the file. Afterwards, processes invoke further file operations with system calls on such file descriptors, e.g., read and write.

  Importantly, file descriptors are local to processes. Thus, the same number, say 5, may refer to file `instructions` for one process but to another file for a different process; in fact, for some processes, 5 may not be a valid file descriptor at all.

  - Files are abstracted as named **streams of bytes**

Each file has a name, and is just used as stream of bytes, regardless of its format and contents. E.g., a text file is a stream of characters, while a movie file contains binary data, both of which are accessed in the same way.

- File abstraction includes "real" files, directories, devices, network access, and more
  - Typical operations: `open`, `close`, `read`, `write`

  Importantly, the file abstraction offers a uniform interface to all kinds of data sources, including directories and network connections.

  Typical operations on files are `open`, `close`, `read`, and `write`, offered by system calls. Here, `open` returns a new file descriptor for some file. That file descriptor can then be used subsequently to `read` or `write` from the file.
- POSIX standard describes three descriptors (numbered 0, 1, 2) for every process

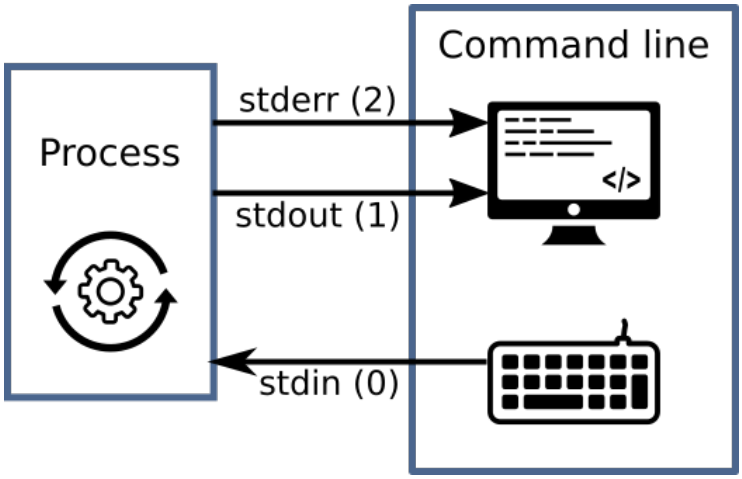

Figure 1: "Standard file descriptors" by Jens Lechtenbörger under CC BY-SA 4.0; using UXWing icons: keyboard, monitor, operations; from GitLab

0. Standard input, `stdin` (e.g., keyboard input)
1. Standard output, `stdout` (e.g., print to screen/terminal)
2. Standard error, `stderr` (e.g., print error message to terminal)

The POSIX standard specifies three file descriptors for every process, which are numbered 0, 1, and 2.

File descriptor 0 is called standard input, and reading from it can be used for keyboard input on the command line.

File descriptor 1 is called standard output, and writing to it can be used to print output on the command line.

File descriptor 2 is called standard error, and writing to it can be used to print error messages. So, both, standard output and standard error can print to the command line.

As a programmer, you need to decide what to print where (if at all). You should choose standard output for ordinary output of your program, and standard error for error messages. Those two types of messages can then be distinguished semantically and, e.g., be sent to different files via redirection or to different consumers in general.

3.2 Drawing on File Descriptors

Warning! External figure **not** included: "File descriptors" © 2018 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
(See HTML presentation instead.)

This drawing illustrates various aspects of file descriptors, most of which were mentioned already. Maybe note the command in the second picture to list open files.

3.3 Files/Streams for IPC

- IPC = Inter-process communication
  - Communication between processes
  - Files and streams enable communication
    - (Next to other mechanisms, e.g., shared memory, signals, networking)

Files can be used for interprocess communication, where different processes communicate by reading and writing on the same file or set of files.

As side note, other mechanisms for communication between processes exist as well. E.g., with shared memory, some frames may be embedded into virtual address spaces of several processes, giving up address space isolation for specific purposes, e.g., to share code or data structures between processes.

Also, processes can send signals to each other (via the OS). You may know this from the `kill` command, which allows users to terminate running processes. Actually, this command can send different signals, which is beyond the scope of our course.

With networking functionality, even processes on different computers can communicate with each other, e.g., web browsers and web servers.

- Files provide persistent storage
  - Written/created by one process
  - Potentially accessed by other processes, **communication**, e.g.:
    - Files with source code
      - Write source code in editors (over time, different processes)
      - Perform quality checks on source files with specialized tools (other processes, maybe passing results to editor)
      - Compile source code to executable code
    - Collect log messages from several processes in one file; analysis of file contents with alerting by separate process

As mentioned already, files are usually stored persistently on secondary storage devices.

Thus, files enable decoupled, asynchronous communication, examples of which are listed in bullet points on the slide.

3.4 Redirection of Streams

- Streams of bytes can be **redirected**
  - E.g., send output to file instead of terminal
    - `head names.txt > first10names.txt`

        - (Recall: This command occurs in cheatsheet of Recall The Command Line Murders)
    - Code for `head` invokes system calls
        - `open` file `names.txt`, results in newly allocated file descriptor
        - `read` from file descriptor for `names.txt`
        - `write` to `stdout` (opened automatically by default)
    - Operator `>` **redirects** `stdout` of process to file `first10names.txt`

Here you see some details about a command which you know from The Command Line Murders.

The program `head` accesses a file and prints some output, by default on the commend line. In terms of file descriptors, it asks the OS to open the input file, which results in a newly allocated file descriptor, i.e., some number larger than those of the standard file descriptors, which in turn can then be used to read contents of the file. The program really writes its output to `stdout`, which the command line by default prints in the terminal. With the output redirection shown here, `stdout` is redirected to a file.

- Also, lots of commands can access data on `stdin`
    - `head < names.txt`
        - Operator `<` **redirects** file to `stdin` of process; here, access of `names.txt` via `stdin`

Programs may also be able to access input on `stdin`, and `head` is an example for such programs: When no file name is passed as argument, `head` reads from `stdin`, i.e., from file descriptor 0. With the input redirection shown here, `head` will see the contents of the file on `stdin`.

3.5 Streams for IPC

- Processes can communicate with **pipelines**/**pipes**
    - One process connects stream as **writer** into pipeline
    - Second process connects stream as **reader** from pipeline
    - Pipelines (and files) are passive objects (used by processes)
- E.g., send `stdout` of one process to `stdin` of another
    - `head names.txt | grep "Steve"`
        - (Recall: This pipeline occurs in cheatsheet of The Command Line Murders)
        - Here, process for `head` sends its `stdout` via **pipe operator** (`|`) to `stdin` of process for `grep`
            - In contrast to files, pipes do **not** store data persistently

Here you see some details about a command which you know from The Command Line Murders.

In this case, `head` sends its output into a pipeline, or pipe for short, from which `grep` reads its input. In terms of file descriptors, the process, or more precisely a thread, for `head` sends its `stdout` via a pipe to the `stdin` of the process for `grep`, where a thread reads this data as input.

Importantly, pipe and files are passive objects, to be used by threads in processes as active subjects. In particular, the use of files and pipe does not create additional threads.

3.5.1 Drawing on Pipes

Warning! External figure **not** included: "Pipes" © 2016 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
(See HTML presentation instead.)

The upper row of this drawing visualizes aspects of pipes discussed so far.

The lower row starts with aspects beyond class topics such as buffering, which may limit or block the writing thread, or signals as notification if the target terminates.

Finally, as already mentioned in the cheatsheet of The Command Line Murders, we can build pipelines with an arbitrary number of stages.

3.6 File Descriptors under `/proc`

- For process with ID `<pid>`, subdirectory `/proc/<pid>/fd` indicates its file descriptors
  - Entries are symbolic links pointing to real destination

    In a dedicated subdirectory for each process under `/proc`, file descriptors in use by the process are shown as so-called symbolic links. You can think of a symbolic link as an additional name, which points to some file or device.

    On Mac OS, listing file descriptors in this fashion does not work. As alternative, you can use a command to list open files shown in an earlier drawing.

  - Use `ls -l` to see numbers and their destinations, e.g.:

    ```
    lrwx------ 1 jens jens 64 Jun 26 15:34 0 -> /dev/pts/3
    lrwx------ 1 jens jens 64 Jun 26 15:34 1 -> /dev/pts/3
    lrwx------ 1 jens jens 64 Jun 26 15:34 2 -> /dev/pts/3
    lr-x------ 1 jens jens 64 Jun 26 15:34 3 -> /dev/tty
    lr-x------ 1 jens jens 64 Jun 26 15:34 4 -> /etc/passwd
    ```

    * Use of `/dev/pts/3` (a so-called pseudo-terminal, which represents user interaction with the command line) for `stdin`, `stdout`, and `stderr`
    * Access of file `/etc/passwd` via file descriptor 4
    * (If you are curious: `/dev/tty` is mostly the same as `/dev/pts/3` here)

  E.g., here we see five file descriptors used by some process, numbered from 0 to 4, where the symbolic link with the name `4` points to a file `passwd`, which contains user information under GNU/Linux.

  We also see that the standard file descriptors all point to the same device, which represents user interactions with the command line; details are not important here.

3.6.1 A Quiz

Take this quiz.

4 Access Rights

We now look at access rights on files and other objects.

4.1 Fundamentals of Access Rights

- Who is allowed to do what?

  Access rights determine who is allowed to do what.

- System controls access to **objects** by **subjects**
  - Object = whatever needs protection: e.g., region of memory, file, service
    - With different operations depending on type of object
  - Subject = active entity using objects: process
    - Threads of process **share** same access rights
    - Subject may also be object, e.g., terminate thread or process

    With that goal, the OS controls accesses to objects by subjects. Here, objects can be of varied types, such as memory, files, or services, while subjects are processes. The threads of a process then share the access rights of a process.

  - Subject acts on behalf of **principal**
    - Principal = User or organizational unit
    - Different principals and subjects have different **access rights** on different objects
      - Permissible operations

    The process is created by some user, maybe as part of an organizational unit, who is called principal.

    Importantly, different principals and subjects have different access rights on different objects, where the access rights specify what operations are permitted on a given object for a given subject.

    As simple example, consider a document with tasks for the next written exam, created by a team of instructors on a multi-user computer. Clearly, that team should be allowed to read and write the exam document, while students must not have any permissions on that document at all. In contrast, students may be allowed to read documents of selected previous exams, but not to modify them.

4.1.1 Typical Access Right Operations

- In general, dependent on object type, e.g.:
  - Files
    - Create, remove
    - Read, write, append
    - Execute
    - Change ownership

    For different types of objects, different operations are permitted or restricted by access rights. For example, on files, operations such as create, remove, read, write, append, execute, or change of ownership may be applicable.

  - Access rights
    - Copy/grant

    Access rights themselves may also be considered as objects under access control. Thus, some subjects may be allowed to copy access rights or to grant access rights to other subjects.

4.2 Representation of Access Rights

- Conceptual: Access (control) matrix
- Slices of access matrix
  - Capabilities
  - Access control lists

Access rights can be represented in different ways, and we look at the examples listed here.

4.2.1 Access (Control) Matrix

- Matrix
  - Principals and subjects as rows
  - Objects as columns
  - List of permitted operations in cell

An access matrix, or access control matrix, is a matrix of rows and columns, where the principals and subjects are listed in separate rows, the objects in columns, and each cell contains a list of permitted operations.

4.2.2 Access Matrix: Transfer of Rights

- Transfer of rights from principal JDoe to process P_1
  - Figure 7.12 (a) of (Hailperin 2019): copy rights

	F_1	F_2	JDoe	P_1	...
JDoe	read	write			
P_1	read	write			
⋮					

This small excerpt of an access matrix demonstrates the representation of access rights in general. John Doe and process 1 are listed as principal and subject in separate rows, while objects are listed in columns. More specifically, the columns list two files and again principal John Doe and process 1.

Note that principal and process occur in column headers as well as row headers, indicating that they serve dual roles as objects and subjects. Access right of process 1 (as subject) are indicated in the row for process 1. You see that the process is allowed to read file 1 and write file 2. You also see that John Doe and process 1 share the same access rights.

Processes obtain their access rights from principals on whose behalf they are operating. For example, if you and me have got user accounts on my machine and if both of us start the same text editor, then the two processes for these text editors will have different access rights, which are derived from our (users') access rights: Typically, you will be able to read and write your own files, while you should be unable to access my files, and vice versa.

In this example, process 1 is working on behalf of principal John Doe, and the rights of John Doe were simply copied to process 1, when that process was created by John Doe.

  - Figure 7.12 (b) of (Hailperin 2019): special right for transfer of rights

	F_1	F_2	JDoe	P_1	...
JDoe	read	write			
P_1			use rights of		
⋮					

A second variant for the transfer of access rights might be used, which avoids copying lots of access rights. Towards that end, a special operation may be used in the access matrix, which treats principals as objects. Here, you see that process 1 has the right to "use rights of" John Doe. Consequently, when that process tries to access some object, the OS will check John Doe's rights.

As the access matrix can be large, it can be split by rows or by columns for simplified management. We look at these variants next.

4.2.3 Capabilities

- **Capability** ≈ reference to object with access rights
- Conceptually, capabilities arise by slicing the access matrix row-wise
  - Principals have lists with capabilities (access rights) for objects
  - Challenge: Tampering, theft, revocation
    - Capabilities may contain cryptographic authentication codes

A capability enables to perform some operation on some object. It can be thought of as reference to the object with permitted access rights.

Conceptually, capabilities arise by slicing the access matrix row-wise. Then, principals have lists with capabilities, or access rights, for objects. When a principal wants to perform an operation on some object, the principal has to provide a matching capability, which is checked by the OS.

The system needs to be secured against tempering or theft of capabilities, and it might also offer revocation mechanisms for capabilities. Towards those goals, capabilities are usually secured by cryptographic mechanism. Details are beyond the scope of our course.

4.2.4 Access Control Lists

- **Access Control List** (**ACL**) = List of access rights for subjects/principals attached to object
- Conceptually, ACLs arise by slicing the access matrix column-wise
  - E.g., file access rights in GNU/Linux and Windows (see Sec. 7.4.3 in (Hailperin 2019))

The access matrix can also be sliced column-wise to obtain a list of rights for subjects or principals on a given object. Such a slice is also called access control list. Such lists are often used on files to specify who is allowed to perform what operations, for which we look at an example next.

4.2.5 File ACLs

- `ls` lists files and directories
  - With option `-l` in "long" form
    - Shortened ACLs
      - Permissions not for individual users; instead, separately for owner, group, other

    - Owner: Initially, the creator; ownership can be transferred
    - Group: Users can be grouped, e.g., to share files for a joint project
    - Other: Everybody else
  * File type, followed by 3 triples with permissions
    - File (`-`), directory (`d`), symbolic link (`l`), ...
    - Read (`r`), write (`w`), execute (`x`) (for directories, "execute" means "traverse")
    - Set user/group ID (`s`), sticky bit (`t`)
  * `ls -l /etc/shadow /usr/bin/passwd`
    - `- rw- r-- --- 1 root shadow 2206 Jan 11 2024 /etc/shadow`
    - `- rws r-x r-x 1 root root 68208 Feb 6 13:49 /usr/bin/passwd*`
  * `ls -ld /tmp`
    - `d rwx rwx rwt 14 root root 20480 Jun 8 13:20 /tmp`

The command `ls` can display access control lists on files in GNU/Linux. The long listings produced with option `-l` show permissions in the form of three triples. The first triple specifies what the "Owner" is allowed to do, the second one is for a "Group", the third one for "Others".

In general, the owner is the user creating a file, but ownership can also be transferred.

Users can also be grouped, e.g., to share files for a joint project. Groups are created by the administrator, with a many-to-many relationship between users and groups. Each file is assigned to one group, and files' groups can be changed by their owners.

If a user is neither the owner nor a member of the group for a file, then "other" permissions apply.

In two `ls` outputs here, we see permissions for two files, namely `shadow` and `passwd`, and for the standard temporary directory `/tmp`.

Both files are owned by `root` (in red), who is the default administrator on GNU/Linux. The file `shadow` contains hashed user passwords, and `passwd` is the command with which users can change their passwords. Clearly, users must not be able to change passwords of other users (except for `root` who can do whatever she likes).

Let us look at the triples of permissions to see how this goal is achieved. The triples may contain letters to indicate read (`r`), write (`w`), and execute (`x`) permissions, with hyphens indicating missing permissions.

First, the permissions for file `shadow` can be interpreted as follows:

The owner triple (in red) specifies that `root` is allowed to read and write but not to execute the file. (As this file just contains data, execution does not make sense.)

Next, group members (in blue) are allowed to read but neither to write nor to execute. Here, the group is `shadow`, which is not important for us. (See this hyperlink if you are interested.)

Finally, others (in green) do not have any permission.

Thus, only `root` can write to `shadow` (`w` is only present in red for the owner, while blue and green parts do not contain that letter). So how can users change their own passwords, which requires updates of the file `shadow`?

We see that everyone is allowed to read and execute `passwd`, which is the command that allows users to change their passwords. Usually, when a user executes a command, the resulting process runs with the permissions of the executing user. Here, however, we see an `s` for "set user ID" in red. With this permission, the OS will run the process for `passwd` with permissions of the file's owner, that is `root`. Thus, the process for `passwd` has write permissions of `root` on `shadow`. (Of course, `passwd` needs to make sure that users only change their own passwords.)

In the directory `/tmp`, everybody is allowed to read and write. With the green so-called sticky bit `t`, users are only allowed to delete their own files, not those of other users.

4.2.6 Drawing on File ACLs

Warning! External figure **not** included: "Unix permissions" © 2018 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal

permission.
(See HTML presentation instead.)

This drawing visualizes the three triples of access control lists explained on the previous slide. In particular, it shows how numeric values can be computed from individual bits for each permission. Such numeric values can then be used in commands to change permissions. Alternatively, a symbolic variant is also available, as explained on the next slide.

4.2.7 File ACL Management

- Management of ACLs with `chmod`
  - Read its manual page: `man chmod`
  - (Default permissions for new files are configurable)
    - (Beyond class topics, see `help umask` in bash)
- Permissions with bit pattern or symbolically
  - Previous drawing illustrates bit patterns for `r`, `w`, `x`
  - Symbolic specifications contain
    - one of (among others) `u`, `g`, `o` for user, group, others, resp.,
    - followed by `+` or `-` to add or remove a permission,
    - followed by one of `r`, `w`, `x`, `s`, `t` (and more)
  - E.g., `chmod g+w file.txt` adds write permissions for group members on `file.txt`

This slide instructs you to read the manual page for `chmod`, a command to "change" the "mode", i.e., permission bits of a file. Thus, with `chmod`, users can manage the access control lists of files.

As illustrated in the previous drawing, that command can set the permission bits for user, group, and others based on numeric values.

In addition, a symbolic specification for the bits of the triples is available as shown here: We can use plus and minus signs to add and remove permissions on the different triples. A specific example shows how to add write permissions for group members on a sample file. Try this out.

5 Conclusions

Let us conclude.

5.1 Summary

- Process as unit of management and protection
  - Threads with address space and resources
    - Including file descriptors
  - Isolation of virtual address spaces as protection mechanism
- File access abstracted via numeric file descriptors as streams
  - Redirection and pipelining for inter-process communication
- Access control restricts operations of principals via subjects on objects

  - GNU/Linux file permissions as example for ACLs
  - Access control as additional OS protection mechanism

This presentation concludes the OS topics of our course.

The process is considered the fundamental unit of management and protection in operating systems. Each process has its own separate address space and resources, including file descriptors. Threads of a process share that address space and resources, and they form the units for scheduling.

The OS isolates the address spaces of different processes from each other, which prevents unauthorized access or manipulation of data belonging to one process by other processes. In addition, it also protects data structures of the OS itself.

File descriptors serve to abstract file access, allowing processes to interact with files using numerical identifiers instead of physical locations. This abstraction enables powerful features such as redirection and pipelining, which can be used for inter-process communication.

Many OSs implement access controls on objects such as files. These controls restrict certain operations by principals acting through subjects. For instance, file permissions are enforced through access control lists that determine what actions individual users and their processes may perform on specific files. With these settings, administrators and users can help prevent unwanted modifications or disclosures of sensitive information.

5.2 Perspective

- Different access control paradigms exist
  - Discretionary access control (**DAC**)
    - **Owner** grants privileges
    - E.g., file systems, seen above
  - Mandatory access control (**MAC**)
    - **Rules/policies** about properties of principals, processes, resources define permitted operations
      - E.g., SELinux, AppArmor
      - More complex to manage/use but "more secure"
  - Role based access control (**RBAC**)
    - Permissions for tasks bound to organizational roles
      - E.g., different rights for students and teachers in Learnweb

Beyond what you saw in this presentation, different access control paradigms exist. File permissions, as presented above, fall under discretionary access control. Here, the owner of a resource needs to define access restrictions or to grant privileges. As users are typically lazy and as their options are limited, this approach is error prone.

More secure systems rely on mandatory access control, MAC for short, e.g., with SELinux. Here, a set of policies defines who is allowed to do what. E.g., the root user is not allowed to do everything any longer. Also, processes started by a user do not inherit user permissions any more, but can be limited, e.g., in terms of permissions on files and directories or regarding network access. Then, if an exploitable security flaw exists in software, it will still be restricted by MAC.

Finally, access control is often tied to roles instead of individual users. E.g., in Learnweb, different roles such as guest, student, and lecturer exist, which come with different privileges.

5.3 In-class: Safety vs Security

- Selected pointers
  - **Safety**: Protection against unintended/natural/random events

    - Requires proper management, involves training, redundancy (e.g., hardware, backups), and insurances
- **Security**: Protection against deliberate attacks/threats
    - Protection of **security goals** for objects and services against **attackers**
    - Security goals and risk
        - **CIA triad** with classical goals: **Confidentiality**, **Integrity**, **Availability**
        - Many more, e.g., accountability, anonymity, authenticity, (non-) deniability
    - E.g.: Processes on "your" system?
        - Advanced persistent threats (APTs), rootkits
        - Check externally, e.g., German Desinfec't
- **Design processes** and **management**
    - E.g., BSI in Germany and ISO standards: IT-Grundschutz

This slide points to the general concepts of security and safety in IT systems, which are not covered in this course. They will be briefly addressed in class.

Every IT system needs to be protected against failures or other negative consequences that arise from unintended events, which concerns the safety of the system, and deliberated attacks, which concerns security.

In organizational settings, safety and security require design processes and proper management, for which national and international standards with recommendations and best practices exist.

You may want your own systems to be safe and secure as well. Then, probably most importantly, create backups, and make sure that you can restore them. Afterwards, check out terms such as advanced persistent threats and rootkits.

In our Bachelor's program, a separate module addresses information security in the context of distributed systems.

Bibliography

Hailperin, Max. 2019. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3.1. https://gustavus.edu/mcs/max/os-book/.

The bibliography contains references used in this presentation.

License Information