WWU MÜNSTER

ERCIS

OS10: Processes

Based on Chapter 7 and Section 8.3 of [Hai19]

(Usage hints for this presentation)

Computer Structures and Operating Systems 2023

Dr. Jens Lechtenbörger (License Information)

Data Science: Machine Learning and Data Engineering (Prof. Gieseke)
Dept. of Information Systems
WWU Münster, Germany

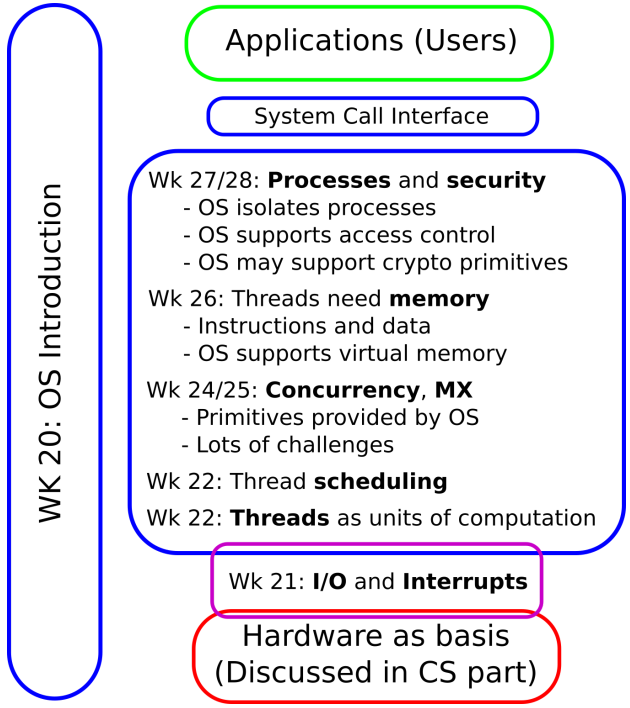
Speaker notes

- To toggle these notes, press `v`
  - If a slide contains audio, notes might show transcript
- Press `?` for key bindings (in particular, `a`, `o`, `n`, `p`, `Ctrl-Shift-f`)
- Presentations support two different PDF formats, see usage notes
  - Both hyperlinked on index page
    - Concise PDF format (replace `.html` and whatever follows in address bar with `.pdf`)
    - Print browser view to PDF (add `?print-pdf` after `.html`, then print to PDF; suggested settings)
- If you find the amount of outgoing links to be distracting, see usage notes
  - Add `?hidelinks` (maybe with a number) after `.html`
- See usage notes for other non-obvious features

1. Introduction

1.1. OS Plan

- OS Overview (Wk 20)
- OS Introduction (Wk 21)
- Interrupts and I/O (Wk 21)
- Threads (Wk 23)
- Thread Scheduling (Wk 24)
- Mutual Exclusion (MX) (Wk 25)
- MX in Java (Wk 25)
- MX Challenges (Wk 25)
- Virtual Memory I (Wk 26)
- Virtual Memory II (Wk 26)
- **Processes (Wk 27)**
- Security (Wk 28)



1.2. Today’s Core Questions

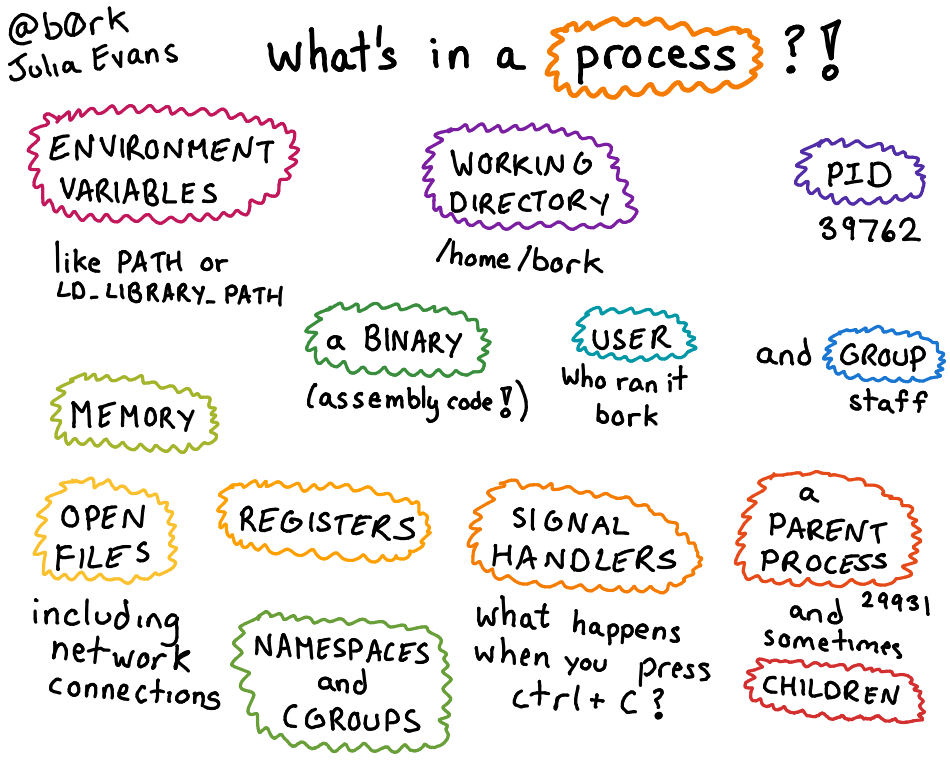
- What is a process?
- How are files represented by the OS and how are they used for inter-process communication?

1.3. Learning Objectives

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

1.4. Retrieval Practice

1.4.1. Recall: Processes



What's in a process?

1.4.2. Previously on OS …

- What are processes and threads?
- What is a thread control block?
- What are kernel and user mode?
- How do threads enter kernel mode?
- How to execute shell commands as part of The Command Line Murders?

1.4.3. Quiz 1

What's running?

1. Select correct statements about computations.
   - [ ] Running programs are managed as processes by the OS.
   - [ ] To create a process, system calls are necessary.
   - [ ] Not every process needs to contain a thread.
   - [ ] The OS parallelizes computations with additional threads to keep all CPU cores busy.
   - [ ] Hack supports multitasking.

2. Select correct statements about system calls
   - [ ] System calls protect resources and hardware access.
   - [ ] The OS kernel creates a new thread for each system call.
   - [ ] Retrieving the currently pressed key requires a system call.
   - [ ] Printing text in some high-level language invokes system calls in the background to draw pixels on screen.
   - [ ] In Hack, every program runs in kernel mode.

1.4.4. Quiz 2

Threads, concurrency, and preemption

1. Select correct statements about thread terminology
   - [ ] Running programs are managed as threads by the OS.
   - [ ] Processes and threads are OS management units.
   - [ ] Each thread may contain one or more threads.
   - [ ] Concurrency can only arise on multi-core systems.
   - [ ] Scheduling may lead to interleaved executions of multiple threads.

2. Select correct statements about preemption
   - [ ] Concurrency can cause preemption.
   - [ ] Preempted threads are garbage-collected by the OS.
   - [ ] In response to an interrupt, a thread may be preempted.
   - [ ] Preemption hinders effective caching.
   - [ ] Management information on stacks can be used to resume preempted threads later on.

1.4.5. Quiz 3

Thread Control Blocks

1. Select correct statements about TCBs
   - [ ] The OS manages a TCB for each thread.
   - [ ] TCBs contain state information to continue preempted threads later on.
   - [ ] The TCB points to a page table.
   - [ ] The TCB includes information for scheduling and about blocking events.

Table of Contents

2. Processes

Speaker notes

These notes summarize core process concepts for which subsequent slides offer more details.

Processes are management units of our 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”. 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 (a) shared by its threads and (b) 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. There, as you have seen earlier, the Linux kernel exports various pieces of management information in the directory /proc, which is a great place to explore what is happening behind the scenes.

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)
- 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
      - Access rights
      - Resource allocation
      - Miscellaneous context

2.1.1. Aside: Single Address Space Systems

- We only consider the case where each process has its own address space
  - OS acts as **multiple address space system**
  - OS mainstream
- [Hai19] contains some details on **single address space systems** (beyond scope of class)
  - E.g., AS/400

2.2. Process Creation

- OS starts
  - Check your OS's tool of choice to inspect processes after boot
- User starts program
  - Touch, click, type
- Processes start other processes
  - POSIX 🚀 Process Management API in [Hai19]
  - Command line (e.g., `bash`) is a process
    - Commands often lead to creation of child processes

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

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, executing user)
  - Address space information
  - Privileges
  - Resources (shared by threads)
    - E.g., file descriptors discussed next
  - Interprocess communication
    - Flags, signals, messages

2.3.1. Seeing Processes

- Recall: `/proc` is a pseudo-filesystem which acts as interface to Linux kernel data structures
  - Subdirectories per process ID (e.g., `/proc/42`) allow to see details of process control blocks
- 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.)
  - `ps -C <name>` shows some details on all processes with the given name
    - Note that some processes, e.g., for `cat` may be too short-lived to be seen with `ps`
- Other OSs come with their own tools

2.3.2. Counters for Context Switches

- `/proc/<pid>/status`
  - File with status information of process
    - View with, e.g.: `cat /proc/42/status`
- Selected information
  - Process ID (also of parent process)
  - Information concerning memory usage
  - `voluntary_ctxt_switches`
    - Thread gave up CPU (yield) or did system call
  - `nonvoluntary_ctxt_switches`
    - Thread removed from CPU (preempted) by OS

2.3.3. Sample Bash Loops

- Bash allows scripting, e.g., while loops with the builtin command `while`:

  ```
  while <condition>; do <commands>; done
  ```

- Consider two infinite loops and take the quiz on the next slide:
  1. `while true; do true; done`
     - Here, `true` is a builtin bash command that immediately returns a true value.
  2. `while true; do sleep 1; done`
     - Here, `sleep` is not builtin, but creates a single-threaded process whose thread sleeps for the indicated number of seconds before the process exits.

2.3.4. Quiz

Who switches when?

1. Select correct statements about the infinite loops.
   - [ ] voluntary_ctxt_switches increases dominantly for loop 1.
   - [ ] voluntary_ctxt_switches increases dominantly for loop 2.
   - [ ] nonvoluntary_ctxt_switches increases dominantly for loop 1.
   - [ ] nonvoluntary_ctxt_switches increases dominantly for loop 2.
   - [ ] Loop 1 is an example for a CPU bound execution.
   - [ ] Loop 2 is an example for a CPU bound execution.

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 a common OS abstraction to organize data as named streams of bytes that are stored *persistently*. (This is covered in Section 8.3 of [Hai19].) Persistence means that files should retain their data beyond crashes and power failures.

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.)

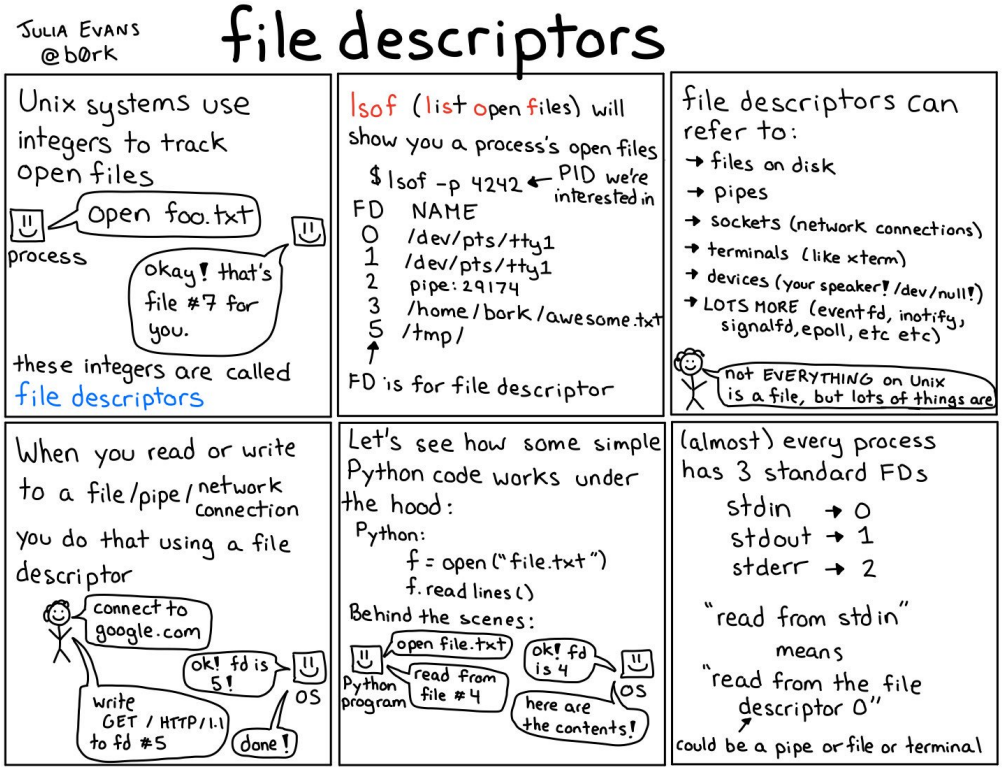
In the context of The Command Line Murders, you also saw that files can serve as inputs and outputs of processes and that one process can communicate its output with the pipe symbol `|` as input to another process. Thus, you are able to explain how the commands shown under items (2) and (3) here produce 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 yourself, e.g.: 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…)

We revisit such uses of files from the OS perspective next. Processes invoke system calls to *open* files and to work on files' contents. You will see that 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 (e.g., read and write) with system calls on such file descriptors.

Importantly, file descriptors are *local* to processes. Thus, the same number, say 5, may refer to file `crimescene` 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.

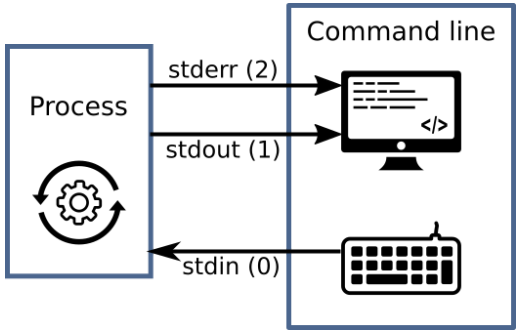
3.1. Drawing on File Descriptors



File descriptors

3.2. File Descriptors

- OS represents open files via **integer numbers** called **file descriptors**
  - Files are abstracted as **streams of bytes**
  - File abstraction includes “real” files, directories, devices, network access, and more
    - Typical operations: Open, close, read, write
  - POSIX 🚀 standard describes three descriptors (numbered 0, 1, 2) for every process
    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)



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, networking, signals)
- Files provide **persistent** storage
  - Written/created by one process
  - Potentially accessed by other processes
    - Then, communication from one process to others
    - E.g., files with source code
      - Write source code in editor (one process)
      - Perform quality checks on source files with specialized tools (other processes, maybe passing results to editor)
      - Compile source code to executable code

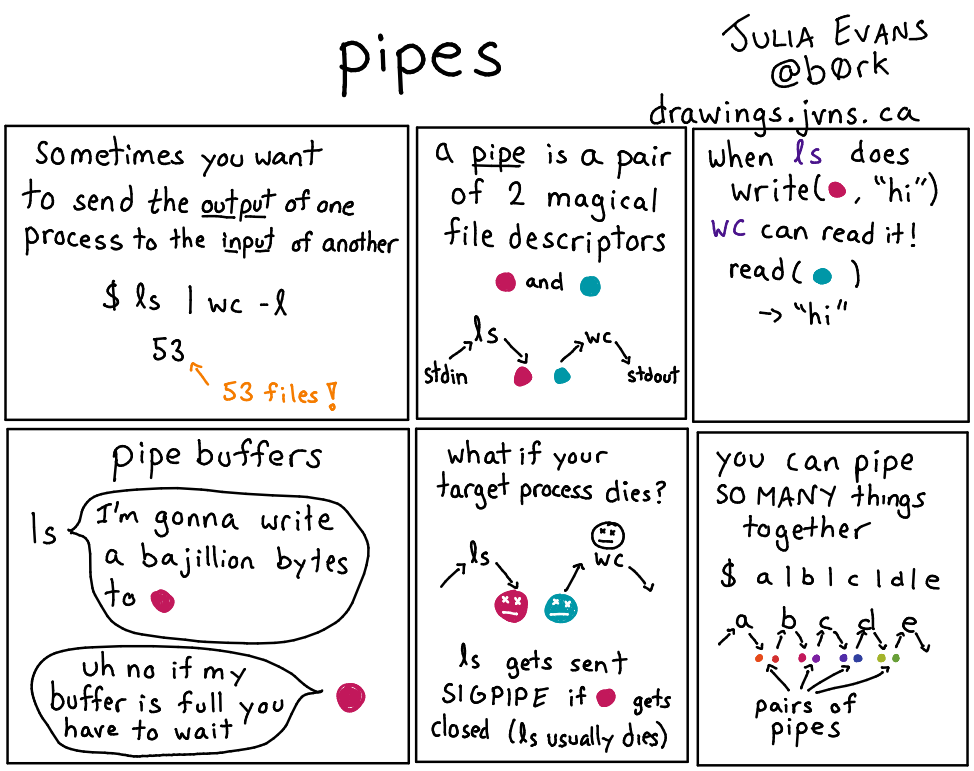
3.4. Redirection of Streams

- 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 The Command Line Murders)
      - Process for `head` outputs first lines of file `names.txt`
      - Code for `head` invokes system calls to open and read the file, which happens via a newly allocated file descriptor
      - The `>` operator **redirects** `stdout` of process to file `first10names.txt`
      - File overwritten if existing, else newly created
  - Also, lots of commands can access data on `stdin`
    - `head < names.txt`
      - The `<` operator **redirects** file to `stdin` of process; here, access of `names.txt` via `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

3.5.1. Drawing on Pipes



Pipes

3.6. File Descriptors under `/proc`

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

3.6.1. Hints for Own Experiments

- Different OSs come with different tools to inspect processes and open files
  - On GNU/Linux or Cygwin, you can inspect file descriptors of long-lives processes under `/proc/<pid>/fd`.
  - Start a process (on the command line or otherwise)
  - Use `ps` to identify process ID for given `name`
    - One line per process; one column is process ID
    - On GNU/Linux maybe: `ps -o pid,lstart -C <name>`
    - For `ps` implementations without option `-C`, use `grep`: `ps | grep <name>`
      - (E.g., Cygwin or MacOS)
      - In this case, you do not see column headers; first column should be process ID
  - ◂ As shown earlier, use `ls -l /proc/<pid>/fd` (with process ID identified in previous step)
  - (Suggestions for Mac users)

3.6.2. A Quiz

Pipelines starting with `cat file | ...`, e.g., `cat vehicles | grep "L337.*9"`, are ugly and should not be used.

1. Select correct statements about cats and pipes.
   - [ ] Command `cat` outputs the `file` to stdout (of the process for `cat`).
   - [ ] The pipeline creates two processes.
   - [ ] The pipeline creates three processes.
   - [ ] With such pipelines, commands such as `grep` work (via their stdin) on contents of the input file of `cat`.
   - [ ] Access of file contents can be performed with redirection of stdin (<).
   - [ ] Access of file contents on stdin does not require additional processes.
   - [ ] Such pipelines exemplify an anti-pattern.

4. Access Rights

4.1. Fundamentals of Access Rights

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

4.1.1. Typical Access Right Operations

- In general, dependent on object type, e.g.:
  - Files
    - Create, destroy
    - Read, write, append
    - Execute
    - Ownership
  - Access rights
    - Copy/grant

4.2. Representation of Access Rights

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

4.2.1. Access (Control) Matrix

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

4.2.2. Access Matrix: Transfer of Rights

- Transfer of rights from principal JDoe to process P_1
  - Figure 7.12 (a) of [Hai19]: copy rights

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

  - Figure 7.12 (b) of [Hai19]: special right for transfer of rights

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

This small excerpt of an access matrix demonstrates (1) the representation of access rights in general as well as (2) the transfer of access rights under the variants (a) by copying and (b) with a special operation.

1. Representation of access rights. In the columns, different objects are shown, namely two files called F_1 and F_2, principal JDoe, and process P_1. Note that JDoe and P_1 occur in column headers as well as row headers, indicating that they serve dual roles as objects and subjects. Access right of process P_1 (as subject) are indicated in the row for P_1. You see that P_1 is allowed to read file F_1 and write file F_2. You also see that subjects JDoe and P_1 share the same access rights.
2. Transfer of access rights. Processes obtain their access rights from principals (users) 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 (say, the final exam for this course), and vice versa.

   In this example, P_1 is a process working on behalf of principal (user) JDoe.

2.a In this first variant of the access matrix, the rights of JDoe were simply copied to P_1 when P_1 was created by JDoe.

2.b 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 P_1 has the right to "use rights of" JDoe. Consequently, when P_1 tries to access some object, the OS will check JDoe's rights.

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

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 [Hai19])

4.3. Access Control Paradigms

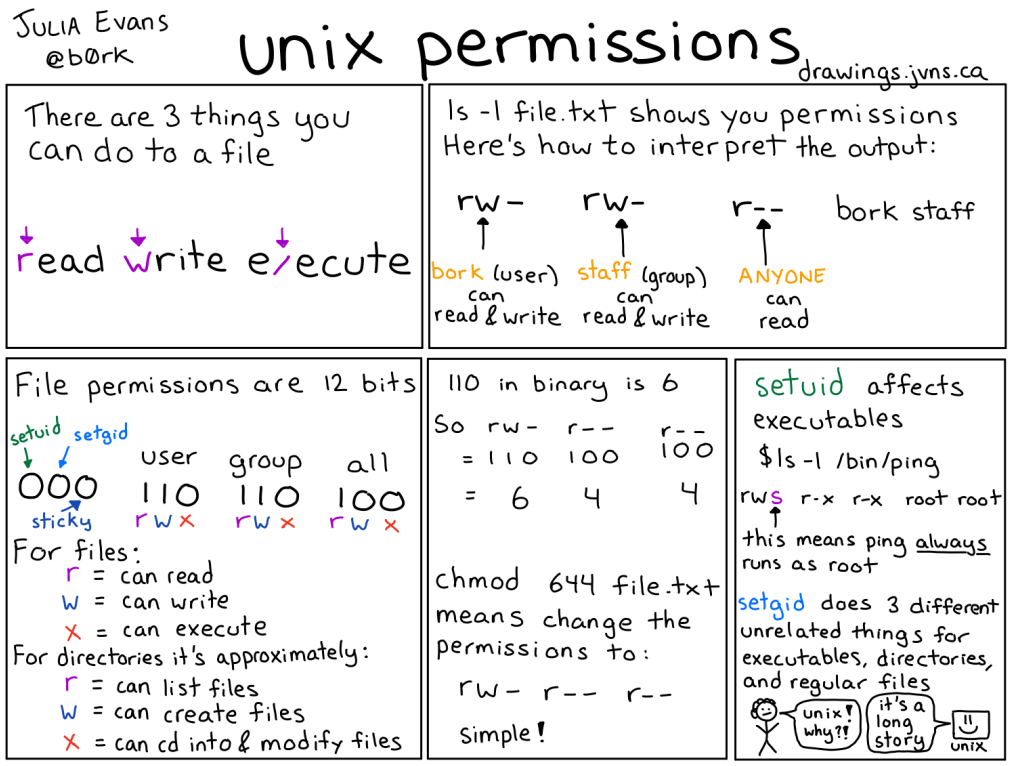
- Discretionary access control (**DAC**)
  - **Owner** grants privileges
  - E.g., file systems
- Mandatory access control (**MAC**)
  - **Rules** about properties of principals, processes, resources define permitted operations
- Role based access control (**RBAC**)
  - Permissions for tasks bound to organizational roles
    - E.g., different rights for students and teachers in Learnweb

4.3.1. DAC vs MAC

- With DAC, **users** are in control
  - Users are lazy
  - If defaults are too restrictive, too permissive rights may be granted
    - “Allow all” is simpler than fine-grained control
- With MAC, a **system** of rules is in control
  - E.g., SELinux 🚀, AppArmor 🚀
  - More complex to manage/use
  - Respects more design principles for secure systems to be discussed in next presentation

4.4. DAC File ACLs in GNU/Linux

4.4.1. Drawing on File ACLs



Unix permissions

4.4.2. File ACLs

- `ls` lists files and directories
  - With option `-l` in “long” form
  - `ls -l /etc/shadow /usr/bin/passwd`
    - `- rw- r-- --- 1 root shadow 1465 Jan 21 2015 /etc/shadow`
    - `- rws r-x r-x 1 root root 47032 Jan 27 01:40 /usr/bin/passwd*`
  - `ls -ld /tmp`
    - `d rwx rwx rwt 14 root root 20480 Jul 4 13:20 /tmp`
    - File type and 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`)
    - 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

Speaker notes

The long listings produced by `ls` with option `-l` show permissions in the form of three triples, where hyphens indicate missing permissions. For file `/etc/shadow` we see the permissions `rw-` , `r--` , `---` . Therefore:

1. The owner (in red) is allowed to read and write but not to execute
2. Group members (in blue) are allowed to read but neither to write nor to execute. (As an aside, groups are created by the administrator, with a many-to-many relationship between users and groups. Each file is assigned to one group, e.g., the group for the file `/etc/shadow` is `shadow` here; files' groups can be changed by their owners.)
3. Others (in green) do not have any permission

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

We see that 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`. 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.)

We also see the directory `/tmp` in which 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.

4.4.3. File ACL Management

- Management of ACLs with `chmod`
  - Read its `man` page
  - (Default permissions for new files are configurable)
    - (Beyond class topics, see `help umask` in bash)
- Permissions can be represented 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`

5. Conclusions

5.1. Summary

- Process as unit of management and protection
  - Threads with address space and resources
    - Including file descriptors
  - Access control as one 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

Bibliography

[Hai19] Hailperin, Operating Systems and Middleware – Supporting Controlled Interaction, revised edition 1.3.1, 2019. https://gustavus.edu/mcs/max/os-book/

License Information

This document is part of an Open Educational Resource (OER) course on Operating Systems. Source code and source files are available on GitLab under free licenses.

> *No warranties are given. The license may not give you all of the permissions necessary for your intended use.*