OS10: Processes [*]

Based on Chapter 7 and Section 8.3 of [Hai19]

Jens Lechtenbörger

Computer Structures and Operating Systems 2022

1 Introduction

1.1 OS Plan

- OS Overview (Wk 20)
- OS Introduction (Wk 20)
- Interrupts and I/O (Wk 21)
- Threads (Wk 22)
- Thread Scheduling (Wk 22)
- Mutual Exclusion (MX) (Wk 24)
- 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)

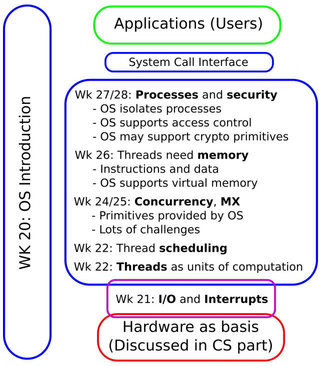

Figure 1: OS course plan, summer 2022

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

---

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

- 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

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

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

1.4.4 Quiz 2

1.4.5 Quiz 3

Table of Contents

2 Processes

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 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

3 File Descriptors

(See Section 8.3 in [Hai19])

3.1 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.)

3.2 File Descriptors

- OS represents open files via **integer numbers** called **file descriptors**
  - Files are abstracted as **streams of bytes**
  - Files provide abstraction for “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)
- Streams can be used for inter-process communication

3.3 Redirection of Streams

- Streams of bytes can be **redirected**
  - E.g., send output to file instead of terminal
    - `head names.txt > first10names.txt`
      - (Recall The Command Line Murders)
      - Process for `head` outputs first lines of file `names.txt`
      - Code for `head` invokes system call 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`

3.4 Streams for Inter-Process Communication

- Streams can be **connected via pipes**
  - E.g., send `stdout` of one process to `stdin` of another
    - `head names.txt | grep "Steve"`
      - (Recall The Command Line Murders)
      - Here, `stdout` of process for `head` is connected via **pipe operator** (`|`) to `stdin` of process for `grep`
      - (`grep` searches for patterns)

3.4.1 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.)

3.5 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
    - (Beyond class, if you are curious: `/dev/tty` is mostly the same as `/dev/pts/3` here)

3.5.1 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

3.5.2 A Quiz

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

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.)

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
      * -rwsr-xr-x  1 root root    47032 Jan 27 01:40 /usr/bin/passwd*
      ```
    - `ls -ld /tmp`

      ```
      * drwxrwxrwt  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

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

License Information