OS10: Processes

Based on Chapter 7 and Section 8.3 of [Hai19]

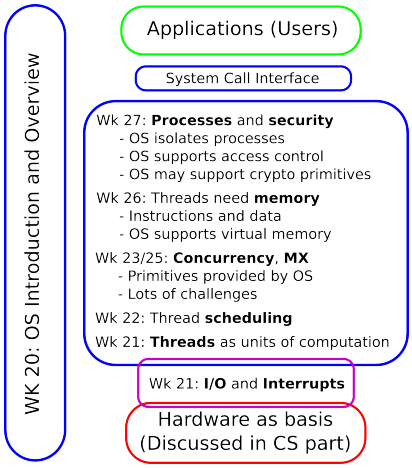
Jens Lechtenbörger

Computer Structures and Operating Systems 2019

1 Introduction

1.1 OS Plan

- OS Motivation (Wk 20)
- OS Introduction (Wk 20)
- Interrupts and I/O (Wk 21)
- Threads (Wk 21)
- Thread Scheduling (Wk 22)
- Mutual Exclusion (MX) (Wk 23)
- MX in Java (Wk 23)
- MX Challenges (Wk 25)
- Virtual Memory I (Wk 26)
- Virtual Memory II (Wk 26)
- Processes (Wk 27)
- Security (Wk 27)
- Wrap-up (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

  - View directories and files, inspect files under `/proc`, build pipelines, redirect in- or output, list processes with `ps`

- Explain access control, access matrix, and ACLs
- Discuss state transitions for threads under different thread implementations

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?

1.4.3 Quiz 1

1.4.4 Quiz 2

1.4.5 Quiz 3

Contents

2 Processes

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
- See [Hai19] for some details on **single address space systems**
    - 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

2.3 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

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**
    - Typical operations: Open, close, read, write
  - Files provide abstraction for "real" files, devices, network access, and more
- POSIX standard describes three descriptors 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 Streams for Inter-Process Communication

- Streams of bytes can be **redirected**
  - E.g., send output to file instead of terminal
    - `ls -alR > ls-alR.txt`
      - Process for `ls -alR` generates (recursive) directory listing

            - The `>` operator **redirects** `stdout` of process to file `ls-alR.txt`
            - File overwritten if existing, else newly created

- Streams can be **connected via pipes**
  - E.g., send `stdout` of one process to `stdin` of another
    - `ls -alR | wc -l`
      - Here, `stdout` of process for `ls` is connected via **pipe operator** (`|`) to `stdin` of process for `wc`
      - (`wc` counts words of its input, with option `-l` it counts lines; `ls` with option `-l` lists one file/directory per line)

3.3.1 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.)

3.4 JiTT Assignment

Answer the following questions in Learnweb.

3.4.1 Bash as Command Line

- Processes and GNU/Linux command line
  - Command line implemented by a process called "shell"
    - Lots of shell variants; Bash used here
  - (Shell and GUI are not part of the OS; each provides another layer of abstraction and interface)
  - (Depending on your OS, some subsequent commands may not work; e.g., `/proc` does not exist on Mac OS X, and `ps` does not seem to know option `-C` used below)
- Execution of commands (programs) in shell creates new processes
  - Typical command creates one single-threaded process (via system call)
- I suppose that you worked on The Command Line Murders for basic Bash knowledge

3.4.2 Ticket to Exam

- Explore file descriptors under GNU/Linux (some alternatives) and interpret the results.
  - `/proc/<pid>/fd`: Sub-directory of /proc with file descriptors
    - `<pid>` needs to be replaced with a process ID
  - Start processes

1. `less /etc/passwd`
2. `less < /etc/passwd`
3. `cat /etc/passwd | less`

- All show contents of file `/etc/passwd`
  - Via process that executes `less` (exit with "q"; as usual, `man less` tells you more)
  - Use `ps -o pid,lstart -C less` to identify process IDs (start time may help finding "your" process in case several users started `less`)
  - Use `ls -l /proc/<pid>/fd` (with process IDs identified in the previous step) to see file descriptors
- In each of the 3 cases: How many processes are created? Which file descriptors are used for what by each process?
  - You need one terminal to run `less`; keep that running
  - Open another terminal (e.g., second `ssh` connection) to run `ps`

3.4.3 JiTT Assignment

- Why might your instructor believe that the pipeline `cat /etc/passwd | less` of the previous slide follows an anti-pattern? (Actually, no pipeline of the form `cat <file> | ...` should ever be used.)

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** $\approx$ 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 JiTT Assignment

Answer the following question in Learnweb.

What did you find difficult or confusing about the **contents** of the presentation? Please be as specific as possible. For example, you could describe your current understanding (which might allow us to identify misunderstandings), ask questions that allow us to help you, or suggest improvements (maybe on GitLab). You may submit individual questions as response to this task or ask questions in our Riot room and the Learnweb forum. Most questions turn out to be of general interest; please do not hesitate to ask and answer in forum and Riot room. If you created additional original content that might help others (e.g., a new exercise, an experiment, explanations concerning relationships with different courses, ...), please share.

5 In-Class Meeting

5.1 `/proc` Revisited

- Recall: `/proc` is a pseudo-filesystem
  - Process listing command `ps` inspects `/proc`
    - E.g., `ps -e` shows some details on all processes (IDs, time, etc.)
- `/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

5.1.1 Drawing on `/proc`

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.)

5.1.2 Quiz

- Consider two infinite loops
  1. `while true; do true; done`
     - Command `true` immediately returns successfully
  2. `while true; do sleep 1; done`
     - Command `sleep` causes thread to sleep for indicated number of seconds
- For which loop do you expect what context switch counter(s) to increase (dominantly)?

5.2 File ACLs in GNU/Linux

5.2.1 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.)

5.2.2 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`
    - `drwx rwx rwx 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`)
      - 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

5.2.3 File ACL Management

- Management of ACLs via `chmod`, `chown`, `umask`
- Examples in class

5.3 Thread Implementation

5.3.1 Questions and Alternatives

- Design questions
  - Can threads be blocked **individually**?
  - How **costly** are context switches?
- Design alternatives
  - Kernel threads
    1. Pure in-kernel system threads (Fig. 7.8 (a))
    2. Threads scheduled by kernel on behalf of applications (Fig. 7.8 (b))
  - User threads (Fig. 7.8 (c))
    - Invisible to kernel
  - (Hybrids)

5.3.2 Thread Alternatives

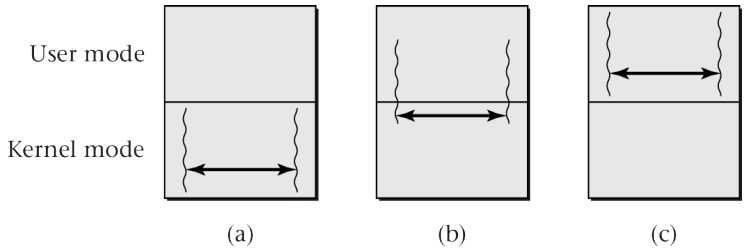


Figure 1: "Figure 7.8 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

- Relationships between threads, scheduling and dispatching code, and CPU's operating mode
  - (a) Threads, scheduler, and dispatcher in kernel mode
  - (b) Threads run mostly in user mode, scheduled and dispatched in kernel
  - (c) Threads along with a user-level scheduler and dispatcher in user mode

Three possible relationships between threads, code for scheduling and dispatching, and the CPU's processor mode are visualized here and explained on subsequent slides.

In this figure, the vertical curvy lines represent threads, while the arrows between them indicate whether switches between threads occur in user mode or kernel mode. (The different heights of arrows in variants (a) and (b) has no special meaning.)

Note that on OSs with multithreading support, the usual threads created by programmers are those of variant (b). E.g., when you create a thread in Java on an OS with multithreading support, the JVM invokes a system call for the creation of a thread by the kernel. Most of your thread's code runs in user mode, but occasionally execution switches to kernel mode for system calls. Also scheduling of threads is performed by the OS in kernel mode.

If you want to program with multiple threads on a singlethreaded OS, you need to resort to variant (c) and manage threads in user mode.

Variant (a) is used by OS developers.

5.3.3 Kernel Threads, Variant (a) of Fig. 7.8

- Kernel manages processes and threads

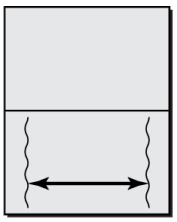


Figure 2: "Part of Figure 7.8 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; cut and converted from GitHub

  - Scheduling on thread basis
- Figure shows threads that are internal to the kernel
  - Housekeeping, e.g., zeroing memory or writing dirty pages to disk
- Execution, context switching, scheduling all **within** kernel mode
  - Almost no overhead

5.3.4 Kernel Threads, Variant (b) of Fig. 7.8

- Kernel manages processes and threads

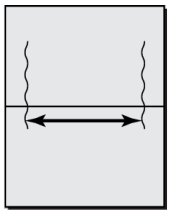


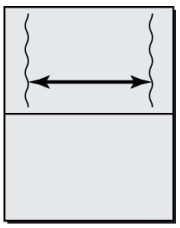
Figure 3: "Part of Figure 7.8 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; cut and converted from GitHub

  - Scheduling on thread basis

- Threads invoke **system calls** for OS functionality
  - **Overhead**
    * Mode switch from user to kernel mode
    * Maybe scheduling decision (e.g., blocking system call)
    * Mode switch from kernel to user mode
  - Note: Kernel address space may be (inaccessible) part of user address space
    * E.g., 3 GiB for processes, 1 GiB for kernel on IA-32
      · In this case, switching from thread to kernel and back to same thread leaves address space unchanged
      · Above overhead reduced

5.3.5 User Threads, Variant (c) of Fig. 7.8

- Kernel may or may not support multithreading



(c)

Figure 4: "Part of Figure 7.8 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; cut and converted from GitHub

- Application performs thread management on its own
  - A thread (managed by the OS) creates and manages its own user threads
    * The thread distributes its time slice among its own user threads
    * User threads are unknown to the kernel
  - E.g., GNU Pth – The GNU Portable Threads
    * Library with functions in user mode
      · Creation, termination of threads
      · Scheduling

5.3.6 User Threads

- Pro
  - Portability

        - Multithreading, even without kernel support
    - Flexibility
        - E.g., application specific scheduling
    - Speed for context switches
        - Without switch to kernel mode
- Con
    - If one user thread gets blocked, entire managing thread gets blocked
    - No real parallelism
        - All user threads belonging to managing thread on same CPU core
        - (No issue on single-core systems)

6 Conclusions

6.1 Summary

- Process as unit of management and protection
    - Threads with address space and resources
        - Including file descriptors
    - Access control as one protection mechanism
- User vs kernel threads

Bibliography

[Hai17] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3, 2017. URL: https://gustavus.edu/mcs/max/os-book/.

[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

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.

Except where otherwise noted, this work, "OS10: Processes", is © 2017, 2018, 2019 by Jens Lechtenbörger, published under the Creative Commons license CC BY-SA 4.0.

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

In particular, trademark rights are *not* licensed under this license. Thus, rights concerning third party logos (e.g., on the title slide) and other (trade-) marks (e.g., "Creative Commons" itself) remain with their respective holders.