UNIX process handling

Tamás Mészáros
http://www.mit.bme.hu/~meszaros/

Department of Measurement and Information Systems
Budapest University of Technology and Economics

Typical problems to solve

- “The system is slow”
  - What's happening?
  - Who is doing what?
- “An application is eating up CPU power”
  - Why is it too slow?
  - What is it doing?
- “The battery depletes too fast”
  - What is running? Is it necessary to run?
  - What is consuming the more power?
- “Core dumped”, “kernel panic”
  - Why is it terminated? What's happened?
  - What causes kernel errors? What apps were running, what's happened

Overview (two lectures)

- Introduction
  - What is a process? How does it start? How to monitor its execution?
  - Its relation to the kernel
  - Context and execution mode
- Processes
  - administrative data
  - state and state transition
  - life cycle: creation, working, waiting, zombie and termination
- Classical UNIX scheduling in practice
  - priority, time sharing, preemptivity
- System calls
  - create a new process
  - execute a new program

The user's view: what is happening?

- Listing active processes
  - `ps`, `ps -ef`, `ps axu`, `ps -u <user>`, **`pstree`**, `...`
  - `top`, `atop`, `htop` és graphical tools (System monitor, `gkrellm`, `procexp`, ...)

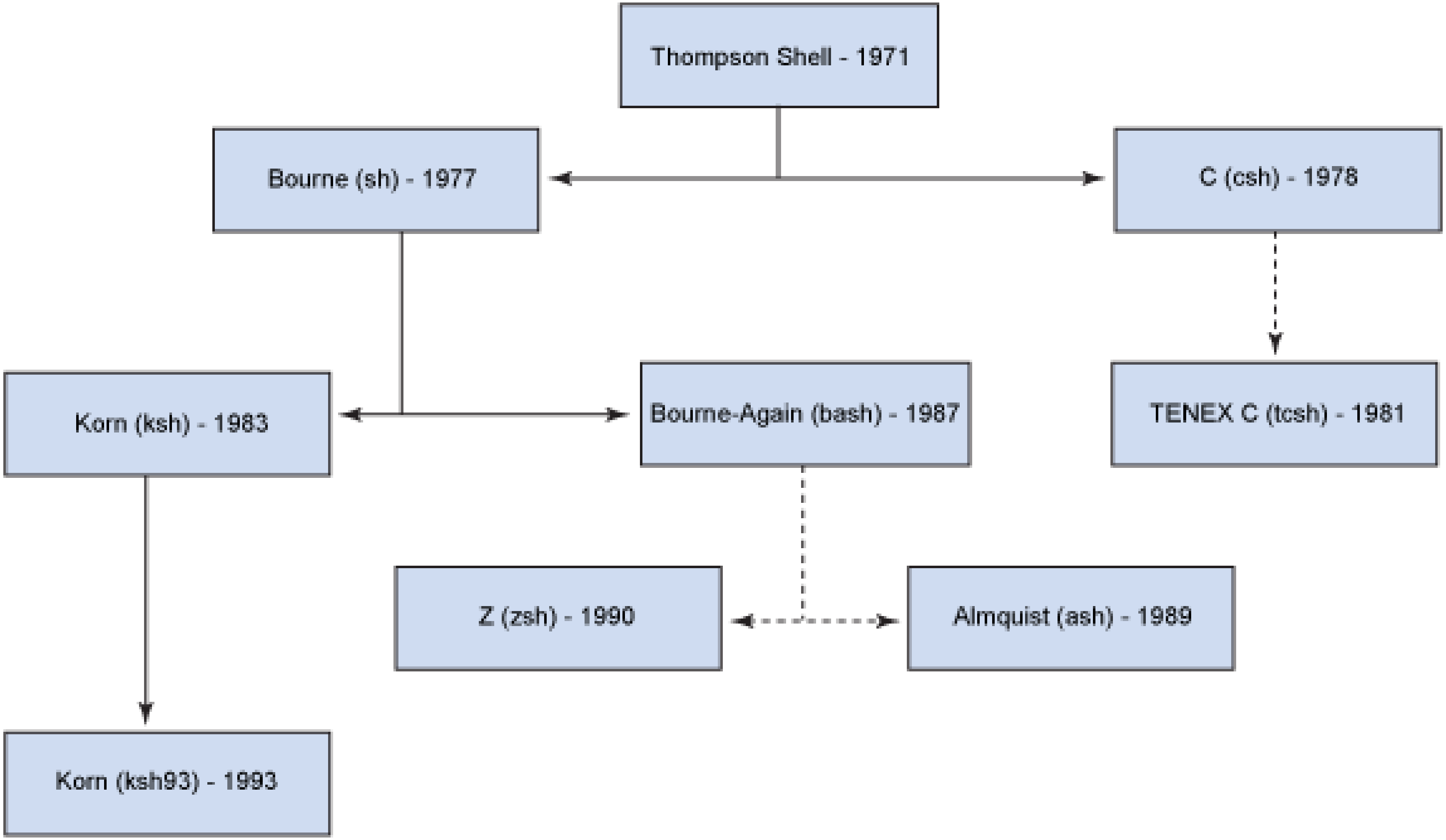
- What do we see in these lists?
  - PID (Process ID): unique identifier (PPID: parent ID)
  - State (running, sleeping, ready to run, etc.)
  - Scheduling informantion (e.g. priority)
  - Credentials (UID, GID, EUID, EGID, UID=0 *root* / *superuser*)
  - STIME: start time
  - TIME: time on CPU
  - CMD: which program is running
  - Statistical data
  - Session info: terminal device (TTY)
  - Aggregate statistics: CPU%, MEM%, DSK%, NET%, etc.
  - ...

The user's view: what are the processes?

- Kernel processes
  - shown between `[ ]` in the lists
  - examples: `kjournald`, `kswapd`, `init` (PID=1)
- Service processes (or daemon processes)
  - Usually started by `init` by running scripts from `/etc/init.d/`
  - The start sequence is specified by `/etc/rc?.d/` file order.
  - Examples: networking, time, file systems maintenance, firewall, LDAP, …
  - `init` is getting replaced by Systemd (see RHEL 7, Ubuntu 15.04)

    Interesting reading: http://0pointer.de/blog/projects/systemd.html
  - Configuring startup services: **`ntsysv, bum`**
- User processes
  - special process: shell (command interpreter)
  - application processes (Firefox, Chrome, Thunderbird, Libreoffice, etc.)

Simple family tree of UNIX shells



Forrás: http://www.ibm.com/developerworks/

Runlevel

- UNIX systems have different service levels (called runlevel)
  - It is identified by a number
  - The system admin can change the runlevel
  - Services start and stop at different levels
- Runlevels
  - See **`/etc/inittab`**
  - There are slight differences among the UNIX variants
    - **0**: full stop
    - **1** or **S**: single-user (admin) mode
    - **2-5**: multi-user modes
    - **3** or **5**: default multi-user mode with graphical user interface
    - **6**: reboot
- Commands to change the runlevel

  `telinit, init, shutdown, halt, reboot`

The user's view: process management

- Process life cycle
  - Starting, ready to run, running, sleeping, stopping

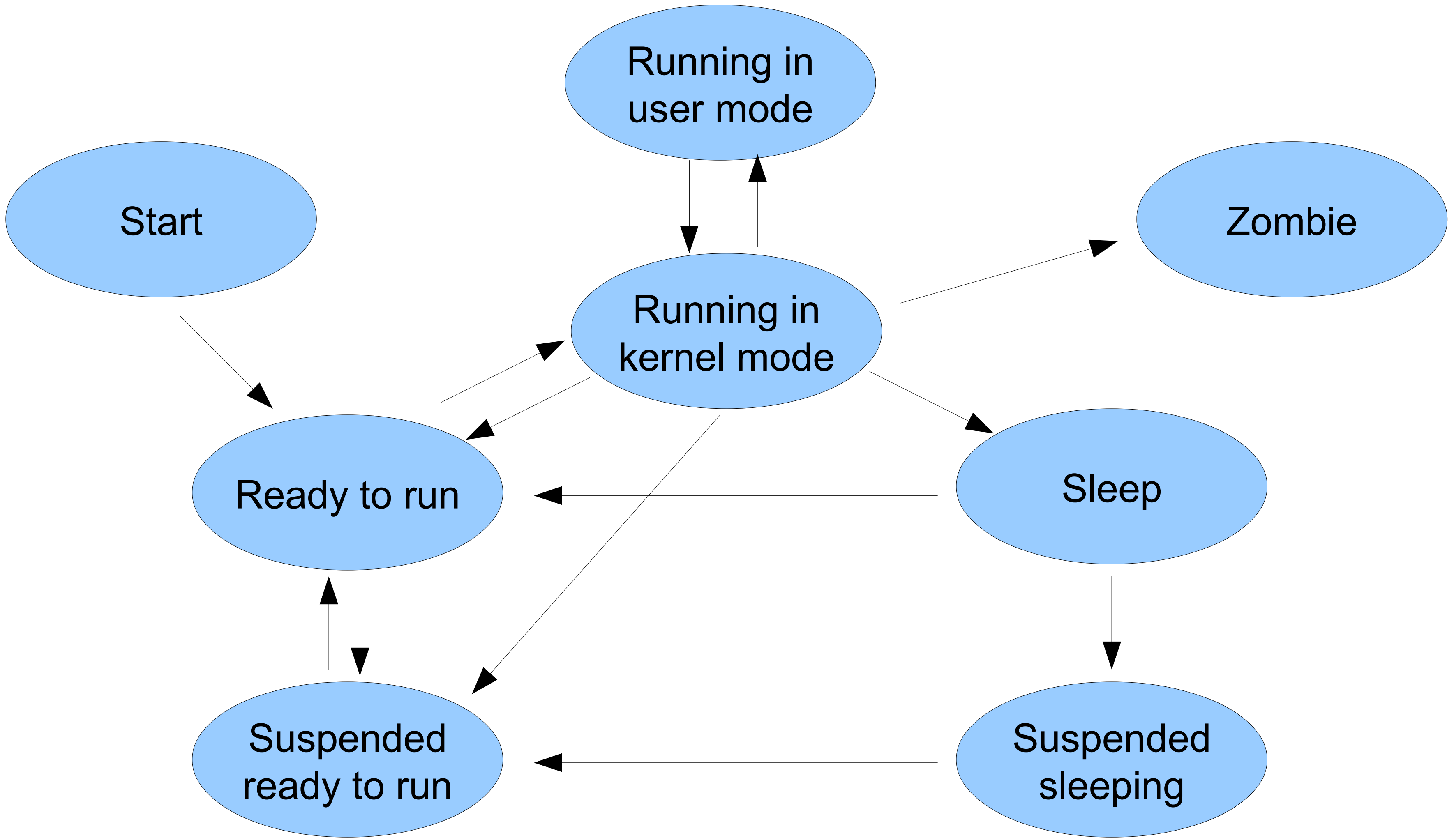
- How and when do they start?
  - System starts: the kernel starts it's own processes and init (PID=1)
  - Boot procedure: daemon processes and terminal monitors
  - The user logs in: shell or GUI processes
  - The user starts applications from the shell or GUI
  - On demand: an event yields a process startup

- How to control them?
  - (in addition to their regular user interface)
  - Signals: `CTRL+C`, `CTRL+Z`, `kill <SIGNAL> <PID>`
  - Setting the priority: `renice`

UNIX Process Life-cycle

- Creation
  - fork(): create a new process
  - exec(): load a new program code\
- Normal operation: running and waiting
  - there are two running states: kernel and user
- Termination
  - exit() system call
  - enters a zombie state first
  - notification of the parent process
  - adopting children
  - final termination

Classical UNIX process states and transitions



fork() and exec() system calls

- `fork()` returns with a different value for the child and parent processes
- `exec()` does not return on success
- Code sample

```
if ((res = fork()) == 0) {
    // child
    exec(...);
    // won't reach this line on successful exec
} else if ( res < 0 ) {
    // fork error (can't create more processes)
}
// res = CHILD_PID (>0), parent
```

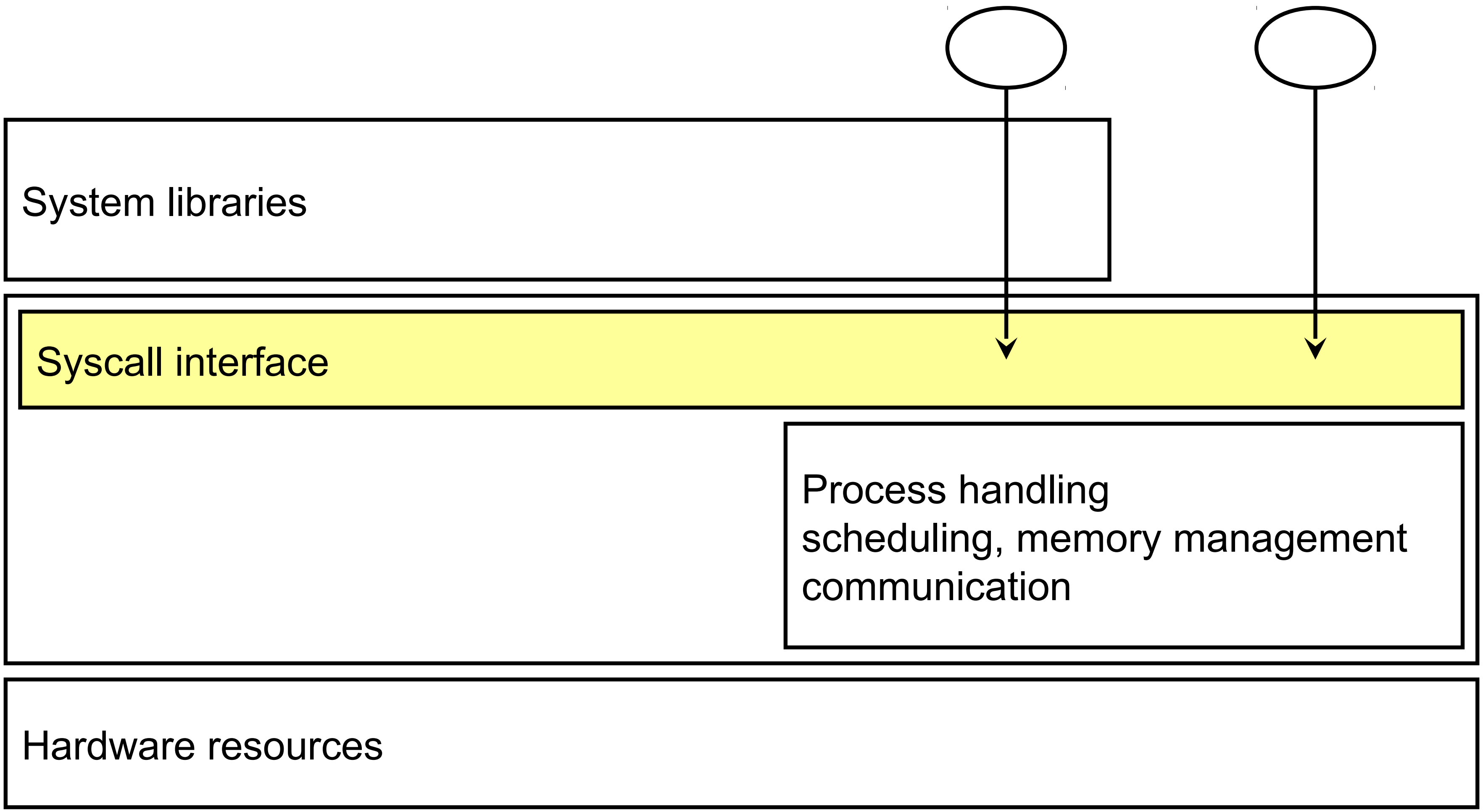
Family tree

- Processes are created by other processes (except PID 1)
  - every process has a parent
  - processes may have children
- fork() gives the PID of the child process to the parent
- The Origin: PID 1 (typically called init, upstart, systemd, ...)
  - the anchestor of all processes
  - runs until the system is running
  - takes over abandoned child processes
  - monitors (sometimes even restarts) important system services
- Family is important in UNIX
  - the parent has to ACK when a child dies

UNIX processes – the kernel's view

- Separating processes from the kernel
  - execution mode: protected or user
  - context: kernel or process data
- Execution mode:
  - Kernel („protected") mode
    - performing restricted actions that need to be protected
  - User („free") mode
    - execution of the user's program code
- Execution context:
  - Kernel (or interrupt) context
    - data needed by the kernel's own tasks
  - Process context (handled by virtual memory management)
    - program code, data, stack, etc.
    - administrative data to handle the process

Processes and the kernel



Running programs: execution mode and context

	user mode	kernel mode
process context	application	system call
kernel context	(empty)	interrupts, kernel tasks

More details on the process context

- Program text, data, stack, etc.
- Hardware context (registers)
- Administrative data (to handle processes)
  - needed only when the process actually runs — **u-area** **Part of the process' adress space**
    - access control data
    - system call state and data
    - open file handles
    - etc.
  - always good to be at hand — **proc structure** **Part of the kernel addr. space**
    - IDs (PID, user, etc.)
    - running state and scheduling data
    - memory management data (including the address of the u-area)
- Environment (inherited from the parent process)
  - *attributum* = *value* pairs (e.g. terminal type, shell, language, etc.)
  - `set`, `setenv`, `export`

Switching from user mode to kernel mode

- This is typically performed during a system call issued by a process
  - Wishes to execute an operation that can only be done in protected mode (e.g. opening, reading, writing a file, querying the system time, etc.)
  - The process calls the appropriate system call (e.g. open(), read(), etc.)
    - This seems like a classical function call but it is not.
    - It is implemented in **libc** that will start the real system call.
  - libc issues the SYSCALL interrupt (this is a CPU instruction)
    - This depends on the actual CPU architecture: SYSCALL, TRAP, SYSENTER
  - The CPU enters protected mode
  - The kernel processes the interrupt and executes the system call program
  - The kernel returns from the interrupt (IRET, SYSEXIT)
  - The CPU leaves the protected mode
  - libc processes the results and returns from the system call
  - The process gets the return values from the system call
- Other hardware interrupts and exceptions (errors) also yield to CPU mode change

Demo: process tracing

- Let's look at the system calls performed by a process
  - trace command: `strace`
  - more information and examples: `man strace`
  - There are other solutions, like the Solaris DTrace
- Let's have a look at the syscalls performed by the *ps* command!

```
strace -c ps
strace -e open ps
```

- Let's peek into the Firefox Web browser's system calls

  RHEL 5, Firefox 3.0.12

```
ps -ef | grep firefox
strace -c -p <Firefox_PID>
```

The /proc filesystem

- We can access kernel data through a special filesystem location
  - /proc
  - see man proc
  - Every process has a directory here named by its PID
  - ps and other process listing programs read these directories
  - We can read them using classical file reading apps (cat, less, more)

- Process data in the /proc filesystem
  - These set of files depends on the UNIX (and kernel) version
  - the program and its parameters (cmd, cmdline)
  - working directory (cwd) and the process environment (environ)
  - file descriptors (fd, fdinfo)
  - memory info (maps, statm)
  - process state (stat – it is not easy to read, use ps instead)
  - system call info (wchan)

    Linux: http://www.lindevdoc.org/wiki//proc/pid/status

Virtual system calls

- The problem: many syscalls, interrupts, context switches take time
  - See the Firefox example: it is calling gettimeofday() way too often
  - gettimeofday() – libc – SYSCALL – mode change – ... – IRET – libc
- There are simple cases when we could try to shorten this path
  - No security, reliability, etc. risk
  - Try to avoid hardware interrupts and execution mode changes
  - If we don't have mode change the calll must be accessible in user space
  - We transfer certain system calls into the process' own address space
- Virtual system calls (Linux)
  - map a special kernel page to the process address space
  - put safe system calls (e.g. gettimeofday()) there
  - no interrupts, no mode changes, fast execution
  - we don't have to modify the user program (it issues the same syscall)

Summary

- Basic knowledge about processes
  - commands: `ps, kill, nice`
  - execution mode and context
  - system calls
  - administrative data (u-area and process table)

- Life-cycle
  - creation: `fork()` system call
  - Loading a new program code: `exec()`
  - states (note the two running and the suspended states)
  - termination: zombie state

- Family tree
  - `fork()` builds a tree, the master process is called `init` (PID 1)
  - parents are notified when a child dies