# Processes and Signals

Marek Kozłowski

Faculty of Mathematics and Information Sciences
Warsaw University of Technology

# Outline of the Lecture

# Outline of the lecture

# Processes and Programs

- A *process* is an execution environment that consists of instruction, user-data and system-data segments as well as resources acquired at runtime.
- A *program* is an executable (`x` permission) regular file containing instructions and data which is used to initialize processes.

# Process Creation – *fork*

- If a program calls the system function `fork()` the kernel creates a new process (*child*) on behalf of the calling process (*parent*).
- The instruction, user-data and system-data blocks of the child are copied from the parent and only a few attributes change, so the child process is almost an exact copy of its parent.
- `fork()` returns child ID in the parent and 0 in the child (and -1 on error). Based on this value separate code blocks can be executed in the parent and the child processes:

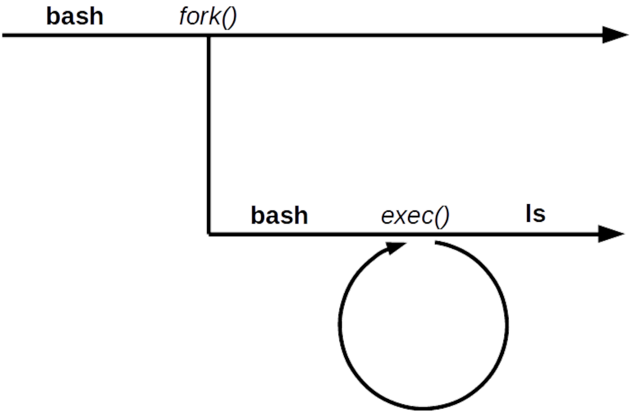
```
switch (id = fork ()) {
    case 0: child_work (); exit (EXIT_SUCCESS);
    case -1: error_occured (); exit (EXIT_FAILURE);
}
parent_work (id);
```

# Process Creation (2) – *exec*

- `exec()` system calls reinitialize the calling process from a designated program file.
- As a result of calling `exec()` the program changes while the process remains.
- `fork()` and `exec()` are typically used together to start a new program.

# Process Creation – Example

- The `ls` command is invoked from `bash`:



# *init* – First Process

- `init` is the first process started after initializing the kernel.
- Its *process ID* (`PID`) is equal to `1`.
- If a parent process terminates but its children are still running they are adopted by the `init` process.
- Note: under systemd Linux distros the `init` process is called `systemd` !

# Zombies and Orphans

- Each process can get its ID and parent's ID by system calls (`getpid()` and `getppid()` respectively) while the only way to access children's ID is to remember values returned by `fork()` .
- When a process ends the `SIGCHLD` signal (see next section) is sent to its parent. The parent process should then call the `wait()` (or `waitpid()`) system call to obtain child's *exit status*. If it doesn't the child is not executing but still exists in the process table. It is called a *zombie* process.
- `init` immediately waits all adopted zombies.

# Process Groups

- Processes started by a single command (for example: processes in a pipeline) are organized into a *process group* or *job*.
- Generally processes are free to migrate to other groups or start new ones (`setpgid()` or `setpgrp()` – depending on the system).
- Process groups are used to control the distribution of signals. For example: termination signals generated by `Ctrl-C` in terminal are delivered to all processes within a foreground (see the next slide) group.

# Sessions

- One or more process groups can form a *session*.
- Normally Unix shells create one session per login (except GUI logins); login shell process becomes a *session leader*.
- Processes cannot migrate to other existing sessions although they can start new ones – `setsid()`, `setsid` .
- A session can be assigned a *controlling terminal* (open by the leader which becomes a *controlling process*). Granting terminal access to distinct process groups (moving them to the *foreground* / *background*) is referred as *job control*.
- If no precautions are taken then terminating a controlling process or hanging up a controlling terminal terminates all processes in the foreground process group.

# Process Attributes – Summary

- Process attributes (stored in system-data) include:
  - *process ID* (*PID*),
  - *parent process ID* (*PPID*) (for orphans: PID of the `init`, 1),
  - *real user ID* and *real group ID*,
  - *effective user ID* and *effective group ID* – different to the above if the setuid or setgid bits are set,
  - *process group ID* (*PGID*),
  - *session leader ID*,
  - *terminal*.

# Process Tree – *pstree*

- The command `pstree` shows running processes as a tree.
- The `-p` option includes PIDs in parentheses after each process name.
- All orphaned processed are shown as children of the `init` process.
- Note that names in curly braces are *threads* not processes (identified by *thread IDs*)!

# Information on Running Processes – *ps*

- `ps` prints statistics on running processes. Numerous display options are defined; typically the `ps -ef` (full listing on every process) is used.
- BSD systems use different `ps` options and different syntax (no dash precedes options). Under BSD the `ps aux` command is commonly used for full listing.

**What is the parent (PPID) of this bash instance?**

```
$ ps -f
```

# Information on Running Processes – *pgrep*

- pgrep combines ps and grep .
- It takes a program name as an argument and prints PID(s) of process(es) executing it on *stdout*.
- pgrep in command substitution mode is commonly used when some other command requires a PID rather than a program name.

**The command *kill* (see next session) requires a PID:**

```
$ kill process_ID
$ kill `pgrep program_name`
```

# Information on Running Processes – *top*

- `top` displays dynamic, real-time information on running processes including CPU (default sorting key) and memory utilization they generate.
- `<` and `>` keys change sorting criteria; `q` – quits.
- Note that `top` causes relatively high CPU utilization itself.

# *top* – Screenshot

```
last pid: 86494;  load averages:  0.83,  0.65,  0.69   up 67+22:48:43  14:44:15
227 processes: 1 running, 224 sleeping, 2 zombie
CPU: 20.2% user,  0.0% nice,  6.5% system,  0.2% interrupt, 73.1% idle
Mem: 1657M Active, 1868M Inact, 273M Wired, 190M Cache, 112M Buf, 11M Free
Swap: 4500M Total, 249M Used, 4251M Free, 5% Inuse

  PID USERNAME   THR PRI NICE   SIZE    RES STATE  C   TIME   WCPU COMMAND
86460 www          1   4    0   150M 30204K accept 1   0:02 11.18% php-cgi
86458 www          1   4    0   150M 29912K accept 0   0:02  8.98% php-cgi
86463 pgsql        1   4    0   949M    99M sbwait 1   0:01  7.96% postgres
85885 www          1   4    0   150M 35204K accept 2   0:07  7.57% php-cgi
85274 www          1   4    0   149M 40868K sbwait 3   0:27  5.18% php-cgi
85267 www          1   4    0   151M 40044K sbwait 2   0:33  4.59% php-cgi
85884 www          1   4    0   150M 41584K accept 2   0:14  4.59% php-cgi
85887 pgsql        1   4    0   951M   128M sbwait 1   0:04  4.20% postgres
85886 pgsql        1   4    0   949M   161M sbwait 0   0:08  3.37% postgres
86459 pgsql        1   4    0   949M 75960K sbwait 2   0:01  3.37% postgres
85279 pgsql        1   4    0   950M   192M sbwait 2   0:14  2.39% postgres
85269 pgsql        1   4    0   950M   199M sbwait 1   0:19  2.20% postgres
85268 www          1   4    0   152M 44356K sbwait 2   0:32  1.17% php-cgi
85273 pgsql        1   4    0   950M   215M sbwait 0   0:19  1.17% postgres
97082 pgsql        1  44    0 26020K  6832K select 0  46:55  0.00% postgres
  892 root         1   4    0  3160K    8K -      2  13:33  0.00% nfsd
 1796 root         1  44    0 19780K 13660K select 3  12:43  0.00% Xvfb
```

# Other *top*-like utilities

- `htop` is an improved alternative task manager to `top`.
- Other system statistics can be acquired by:
  - `iotop` – I/O (disk) utilization,
  - `iftop` – network connections,
  - `ntop` – network status.
- All above utilities are not standardized, platform-dependent and usually offered as optional software.

# */proc* File System Revised

- Any information on given process can be accessed via procfs file system – in a form of files in the `/proc/<PID>/` directory. The files are system-specific, see `man 5 proc` of your system for details.
- Under Linux the following files may be available:
  - `/proc/<PID>/cmdline` – full command which started the process,
  - `/proc/<PID>/cwd` – symlink to the working directory,
  - `/proc/<PID>/environ` – environment variables for the process,
  - `/proc/<PID>/exe` – symlink to the executable file,
  - `/proc/<PID>/fd` and `/proc/<PID>/fdinfo` – file descriptors; positions and flags,
  - `/proc/<PID>/status` – process statistics.
- Later during this presentation we'll show that the utilities like `pgrep`, `ps`, `pstree` or `top` just parse above files.

# Outline of the lecture

# Signals

- The simplest interprocess communication method. No data except signal type is delivered to a process.
- Notify about system event (for example: dividing by 0 generates a `SIGFPE`) or can be generated by a process.
- Kernel and root processes can send signals to all processes.
- Non-root processes can send signals to processes running on behalf of the same user.
- Sequence of the same signals can be reduced to a single one.

# Signals (2)

- Most Unix systems define approx. 28 signals.
- Only two signals: `SIGUSR1` and `SIGUSR2` are left for user-defined purposes.
- Additional user-defined signals (`SIGRTMIN` to `SIGRTMAX`) are introduced by the *Real Time Signals* POSIX extension. RTS signals are discussed during *Operating Systems* and *Unix Programming* however are beyond the scope of this course.

# Signal Handling

- All signals except `SIGKILL` and `SIGSTOP` can be *blocked*, *ignored* or *intercepted* and handled by a receiving process.
- Blocked signals are postponed (not delivered to the process until unblocked).
- Ignored signals are not delivered to the process.
- Signals which are not blocked nor ignored immediately interrupt current process or function execution.
- A *signal handler* is a short function invoked when receiving a signal.
- If no handler is defined then the default action is invoked (for majority of signals: a process termination).

# Signal Classification

- By origin:
  - *natural* – generated by some event,
  - *synthetic* – by calling a system call (usually: `kill()`).

  All natural signals can be generated synthetically; `SIGUSR1` and `SIGUSR2` signals can be generated synthetically only.
- By default action:
  - *terminate* (and optional *core dump*),
  - *stop/continue*,
  - *ignore* – `SIGCHLD` to parent on child termination.
- By related information:
  - *error notification*,
  - *user / application generated*,
  - *process state change / job control*,
  - *timer*.

# Error Signals

- `SIGFPE` – arithmetic operation exception,
- `SIGILL` – illegal instruction,
- `SIGPIPE` – writing to a pipe with no readers,
- `SIGSEGV` – memory segmentation error.

# Termination Request Signals

- `SIGTERM` – default signal, polite request to terminate; sent to all processes on system shutdown,
- `SIGKILL` – unconditional, immediate process termination request; can result in data lost or corruption,
- `SIGINT` – generated by `Ctrl-C` in terminal,
- `SIGQUIT` – generated by `Ctrl-\` in terminal; same as `SIGINT` except it creates a *core dump*,
- `SIGHUP` – sent on terminal hang-up; commonly used for reloading daemons.

# Sending Signals with *kill*

***kill* syntax:**

```
$ kill [-SIGTYPE] <PID>
```

- The command `kill` wraps the `kill()` which is the basic system call for sending signals.
- Any signal (even `SIGCHLD` or `SIGFPE`) can be send with `kill`. The default one is `SIGTERM`.
- Signals can be specified by name (`-SIGKILL`) or number (`-9`) .
- `kill` specifies process by its PID. Other utilities that allow reference processes by names may be available. Depending on the system `killall` and `pkill` may be used.

# Signals and Terminal Control Sequences

- Many signals can be send by terminal control sequences to process in foreground process groups.
- Note that control sequences may be remapped by programs, especially those which use advanced terminal functions. For example: `nano` uses `Ctrl` sequences for internal commands.
- The most common control sequences are:
  - `Ctrl-C` – `SIGINT`, terminate,
  - `Ctrl-\` – `SIGQUIT`, terminate with a core dump,
  - `Ctrl-Z` – `SIGSTOP`, stop (can be resumed, it doesn't terminate nor release resources),
  - `Ctrl-D` – sends end-of-file (EOF), doesn't generate any signal.

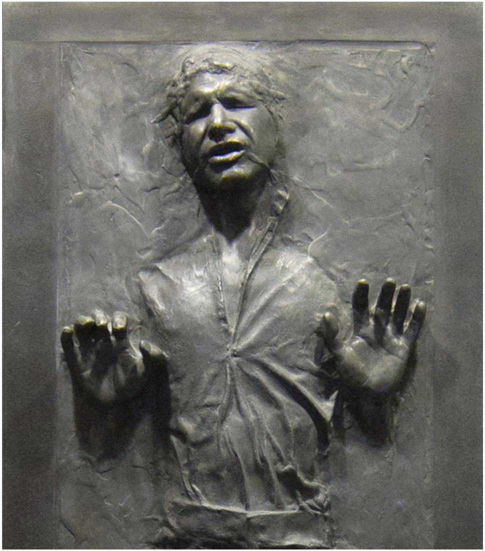
# Terminating vs. Stopping a Process

- Terminating a process results in calling `exit()` – aborting process execution, releasing system resources and changing process state to *dead*.
- Stopping a process means *freezing* it. The process execution is suspended but it remains *alive* and still occupies all allocated resources. Stopped process can be resumed.

---

- When discussing daemons (the next labs) the term *stop* is used as a synonym of *terminate*. Yes, it IS confusing.

# Han Solo Receives a SIGSTOP Signal



## Background Processes

- `jobs` displays status of all jobs in the current session.
- Running a command with *`command-name`* `&` (following it by an ampersand) starts it in the background.
- Alternatively, if a process is running in the foreground it can be stopped by `<Ctrl-Z>` and then resumed in the background with the `bg` command.
- `fg` places a job in the foreground.
- If there are multiple jobs in the current session then `bg` and `fg` require a job number as an argument – for a list of job numbers run `jobs` .

# Outline of the lecture

# Process Tracing

- Most Unix systems provide tools for tracing and displaying all system calls made by a process.
- Process tracing may be especially useful for:
  - determining causes of freezes or crashes,
  - checking which configuration files and in which order are read by a process,
  - identifying resource files utilized by a process,
  - reverse engineering (determining *how it works*).
- Linux provides `strace`, other systems include `ktrace` (BSD) or `truss` (AIX, Solaris)

# Process Tracing in Linux – *strace*

**Tracing a process – typical usage:**

```
$ strace [-f|-ff] [-o logfile] [-e what] [command|-p <pid>]
```

- The options are as follows:
  - `-f` and `-ff` – include children processes (if any) for tracing,
  - `-o logfile` – log output to a file; if `-ff` is set then each child's system calls are logged to a separate `logfile.<pid>` file,
  - `-e` allows specifying which events to trace and how to trace them,
  - `-p` – attach to and trace a currently running process (by PID). Regular users cannot trace processes of other users.
- With `strace` it's trivial to show that `ps`, `pstree` and `top` get process information from procfs (`/proc/<PID>/`) files.