The „Init“ process

(System and services startup and shutdown)

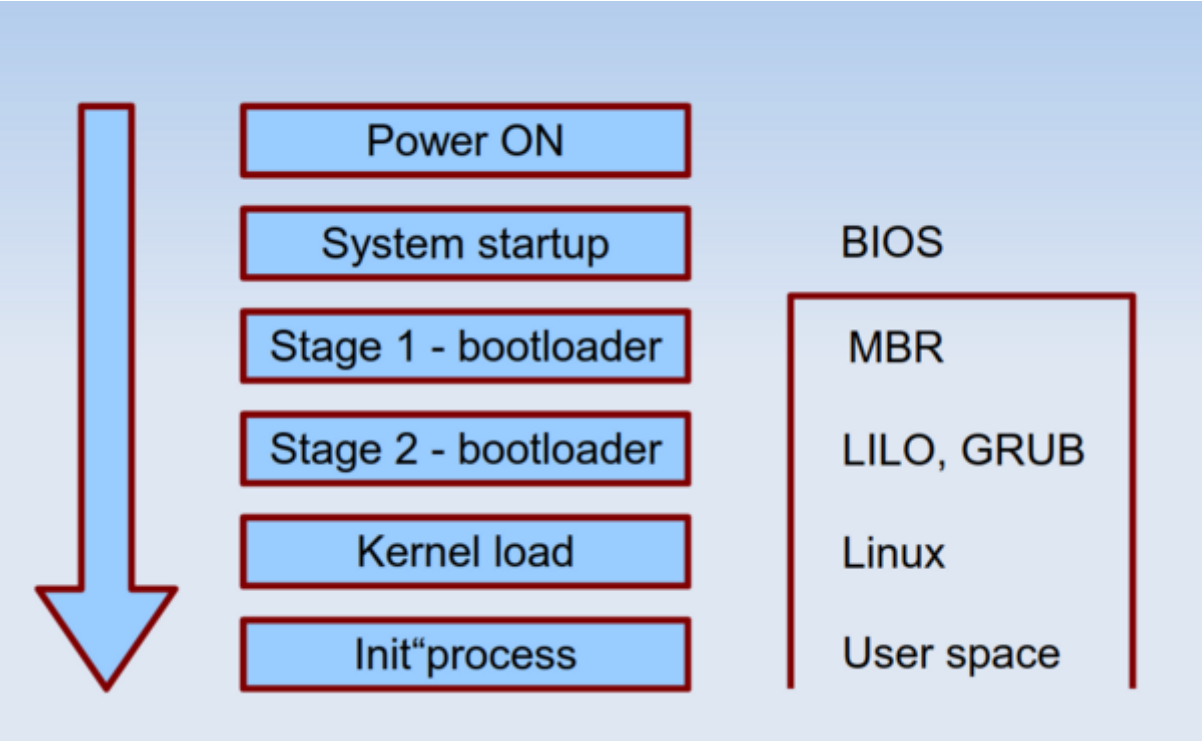
How to lay my hands on init?

- To play with the *init* system one needs a distribution that still uses the thing :)
- This gets harder as all the distros move towards `systemd`
- Use link to get the Debian 7 Virtualbox appliance (cca 400M).
  - This system is the last Debian using SysVinit (before switching to systemd) yet modern enough not to impose much other limitations (besides not having systemd ofc.)
- BTW sysVinit is still not dead and even develops new features here and there (2024)

Other init alternatives

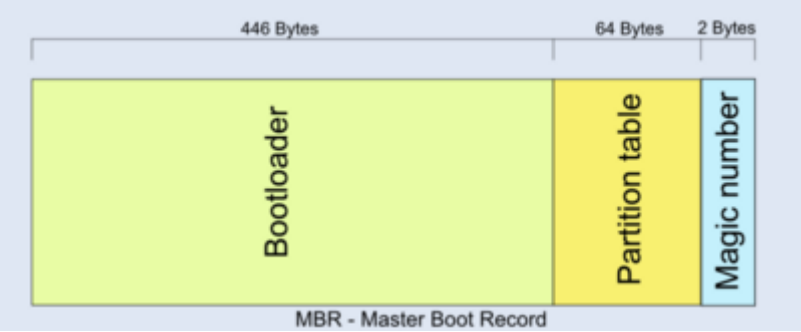
- **systemd** - does not need more comment for now, we will play with it the whole next lesson
- **upstart** - Ubuntus attempt to improve the init process (peaked around 2014), now Ubuntu is fully switched to systemd too.
- **s6** - s6 is a small suite of programs for UNIX, designed to allow process supervision (a.k.a service supervision)
- **runit** - runit cross-platform init system that can run on GNU/Linux, Solaris, *BSD, and Mac OS X and it is an alternative to SysV init, which offers service supervision.
- **openRC** - openRC is a dependency-based init system for Unix-like systems that maintains compatibility with the system-provided init system, normally located in /sbin/init. OpenRC is Gentoo's native init system.
- **others** Nice list/comparison of other mostly all relevant init-like systems.

Linux Boot procedure



Boot stage 1

- MBR – reside on first 512B of HDD
- Search for active partition
- Load active partition boot record
  - continue to stage 2



Stage 2 - kernel loader

- More sofisticated than MBR
- Can load different OS, eg. Windows, Linux, etc.
  - Can load different versions of Linux kernel
- You can modify some kernel parameters before kernel loading
  - We may show the **password recovery process** and init killing here
- Loads kernel image into memory `/boot/vmlinuz`
- Loads initial ramdisk `/boot/initramfs`
- Gives the control to Linux kernel

Initramfs

An **initramfs** (initial ram file system) is used to prepare Linux systems during boot before the init process starts.

- The initramfs usually takes care of mounting important file systems (*by loading the proper kernel modules and drivers*) such as `/usr` or `/var`, preparing the `/dev` file structure, etc.
- Users who use an encrypted file system will also have the initramfs ask them for the passphrase before it can mount the file systems.
- When the file systems are mounted, control is passed on to init which then takes care of further starting all necessary services and booting up the remainder of the system.

Kernel loaders

- **LILO** (Linux Loader) – basic simple loader
  - **ELILO** - like LILO but with EFI support
- **GRUB** - (GRand Unified Bootloader)
  - more advanced
  - default for most distributions
  - under active development
  - suitable for most applications
- but OFC more boot-loaders do exist
- Read more about EFI-UEFI boot options here
- No More Boot Loader NMBL initiative loads kernel directly,... not used in any official ditro so far (2024)

Kernel loading

- Check available RAM
  - Reserve small amount of memory for kernel structures
- Hardware detection and initialization
- Kernel is small with reduced set of drivers
  - Kernel supports loadable modules
  - Kernel can dynamically load modules (drivers) for HW support
- Starts special daemons, processes, kernel threads,...
  - INIT, kflushd, kswapd,...

Kernel threads

- **`kflushd`**: The kflushd kernel thread is responsible for **flushing dirty pages from the page cache to the disk** in a Linux system. When data is written to files, it is first stored in the page cache in memory for performance reasons. kflushd ensures that these changes are periodically written (flushed) to the actual storage devices. This process is **essential for data integrity**, as it prevents data loss in the event of a system crash. kflushd operates based on algorithms that determine when and which dirty pages should be flushed, optimizing system performance while maintaining data consistency.

    - **`kswapd`**: Responsible for **freeing up system memory by swapping out less frequently used pages** from the RAM to the swap space on disk. This helps prevent the system from running out of physical memory and keeps it responsive.
    - **`pdflush`**: pdflush (per-disc flush) is another kernel thread that works alongside kflushd to flush dirty pages from the page cache to the disk. It's responsible for **maintaining the integrity of the file system by ensuring that all modified data is safely written** to the storage devices.
- **Kernel threads continue…**
    - **`ksoftirqd`**: **handle software interrupts in a preemptive kernel**. In Linux, interrupts are mechanisms by which the kernel can respond immediately to certain events. When a hardware interrupt occurs, the appropriate interrupt handler runs. However, some interrupts, known as softirqs, are deferred and handled by ksoftirqd threads. This helps maintain system responsiveness.
    - **`kworker`**: kworker threads are **a pool of kernel worker threads that perform various tasks** in the background. These tasks can include I/O operations, network processing, and other kernel-related activities. kworker threads are a part of the Linux kernel's workqueue system, which allows asynchronous execution of work items.
    - **`kthreadd`**: kthreadd is the kernel thread daemon that creates and **manages other kernel threads during the system's boot process**. It serves as the parent thread for many kernel threads.
    - **`migration/N`**: These threads are responsible for **migrating processes between CPU cores** to balance the system's load and improve overall performance. The number of migration threads can vary based on the system configuration.
    - and more…

The „init“ process

- short for **initialization**
- spawns all other processes
- It runs as a daemon typically with PID 1
- Compatible with System V (SVR4)
- SVR4 init examines the `/etc/inittab`
    - Default runlevel
    - Switching among runlevels
- Spawns getty at the end of initialization
- It's also used for system shutdown or reboot
- Manual controll (press I at start of „Init“) vs. Automatic run (this may differ per distribution/setting)
    - search for “interactive boot”&your_distro_name

„Init“ runlevels

- There are generally 8 runlevels (0-6, S)
    - 0 Halt
    - 1 (S) Single user mode
    - 6 Reboot
- Other runlevels depends on distribution of Unix
    - Behavior is defined in /etc/inittab
- 2-5 full multi-user modes

    - 2 – text console login /Debian
    - 3 – text console login /Other
    - 5 – X Window (GUI)

„Init“ runlevels

- **Run levels** location in FS tree
    - Run level 0 /etc/rc0.d
    - Run level 1 /etc/rc1.d
    - Run level 2 /etc/rc2.d
    - Run level 3 /etc/rc3.d
    - Run level 4 /etc/rc4.d
    - Run level 5 /etc/rc5.d
    - Run level 6 /etc/rc6.d
- telinit / init : `telinit 3` - switches to RL3
    - Man pages(Ubuntu 2020): telinit may be used to change the SysV system runlevel. Since the concept of SysV runlevels is obsolete the runlevel requests will be transparently translated into systemd unit activation requests.
- `who -r` returns the current run level

```
root@machine:~# who -r
         run-level 5  Sep 26 12:42
```

...the upper listing is from Ubuntu 20.04 (2020), even modern systemd based systems retain some degree of **backwards compatibility with SysVinit runlevels**.

„Init“ tasks

- Sets machine hostname
- Sets timezone
- Check filesystems (fsck utility)
- Mounts file systems
- Removes content of `/tmp` dir or mounts /tmp as ramdisk
- Configures network interfaces
- Starts daemons and network services, etc.

Using „init“

- Based on system of links in `/etc/rcX.d/S[XX]servicename`
- Run `/etc/init.d/sshd start` when switching to run level 2:

```
root@machine:~# ln -s /etc/init.d/sshd /etc/rc2.d/S99sshd
```

- Run `/etc/init.d/sshd stop` when switching to run level 0:

```
root@machine:~# ln -s /etc/init.d/sshd /etc/rc0.d/K99sshd
```

- Special utilities to maintain and manipulate the services(links) may exist

        - support and compatibility depends usually on dirtribution
        - `chkconfig –level 2345 sshd` (RedHat)
        - `rc-update add sshd.wlan0 office` (Gentoo)
    - `/etc/rc.local`
        - It's started at the end of runlevels 2-5
        - It's good place for your own tasks after OS boot

Running processes

- Start a daemon
    - using start script
        - `/etc/init.d/ssh start` (start, reload, restart, stop, status)
    - Manually running the binary
        - `/usr/sbin/sshd -p 2020`
- Stop it then
    - `/etc/init.d/ssh stop`
    - `kill -s TERM $PID #more about PIDs/PIDfiles later`

Keep the service running

Traditional SysV init scripts do not have built-in mechanisms for automatically restarting services in the event of a crash.

Nevertheless one can implement a custom logic within the init script, such as using a wrapper script that monitors the service process and restarts it if it crashes. This level of process supervision and automatic restarts is not a native feature of SysV init.

- ```
  #!/bin/bash

  SERVICE="/path/to/your-service-binary"
  SERVICE_OPTIONS="-a 123 -b 456 --whatever"
  SERVICE_NAME="your-service"
  SLEEPTIME=10

  while true; do
      if pgrep -x "$SERVICE_NAME" > /dev/null
      then
          # Service is running, do nothing
          sleep $SLEEPTIME
      else
          # Service is not running, restart it
          echo "Service '$SERVICE_NAME' is not running. Restarting..."
          $SERVICE $SERVICE_OPTIONS
          sleep $SLEEPTIME
      fi
  done
  ```
  ```

PID files

- **PID (Process ID) file** is a file that stores the process ID of a running application or service. The PID file is commonly used to track and manage daemons or background processes.
- stored in the `/var/run/` directory or, more commonly in modern systems, in the `/run/` directory. The naming convention often follows the pattern `processname.pid` or `processname.pidfile`. The content of the PID file is simply the numeric process ID of the running process.
- Usage of `$!` which lists the PID of last executed command

```
$> myprogram &
$> echo $!
4035
```

Generate & use a PID file

primitive exapme of PIDfile usage

```
#...run the service,... some java server in this example
SELENIUM_PROGRAM=/usr/bin/java
SELENIUM_OPTS="-jar /home/vagrant/selenium-server-
standalone-2.24.1.jar"
SELENIUM_PID_NAME=selenium
SELENIUM_LOG_FILE=selenium.logfile
$SELENIUM_PROGRAM $SELENIUM_OPTS 2>&1 >>$SELENIUM_LOG_FILE &
PID=$!
#save the PID to file
echo $PID > /var/run/$SELENIUM_PID_NAME.pid
```

use the pidfile later to kill the service

```
$> cat /var/run/selenium.pid | xargs kill
```

```
jano@jano-ThinkPad-P71:~$ ping root.cz &> /dev/null&
[1] 5861 #ping now runs in the background
jano@jano-ThinkPad-P71:~$ echo $! > pidfilePing.pid
#PID was saved to the pidfile for later use
jano@jano-ThinkPad-P71:~$ ps -A | grep -f pidfilePing.pid
   5861 pts/0    00:00:00 ping #the search was successful
jano@jano-ThinkPad-P71:~$ echo $?
0 #that generates a happy exit code of that grep
jano@jano-ThinkPad-P71:~$ cat pidfilePing.pid | xargs kill
[1]+  Terminated              ping root.cz &> /dev/null
jano@jano-ThinkPad-P71:~$ ps -A | grep -f pidfilePing.pid
jano@jano-ThinkPad-P71:~$ echo $?
1 #now the ping process was dead already, so the exitcode is 1
```

```
jano@jano-ThinkPad-P71:~$
```

- ...usage of grep + if in bash scripts to make decisions based on pidfile

```
#...
$PIDFILE=pidfile.pid
if ps -A | grep -f $PIDFILE; then
    echo running process found
else
    echo not found
fi
#...
```

Signals

- `man 7 signal` - go see the documentation
  - What do those numbers in man even mean?
- `kill` – sends signal to process / man kill

Linux signals

```
Number    Signal Name       Default Action    Comment
POSIX (Yes/No)
-------------------------------------------------------------------------
---------------------------------------
1         SIGHUP            Terminate         Hangup detected on
controlling terminal    Yes
2         SIGINT            Terminate         Interrupt from keyboard
Yes
3         SIGQUIT           Core dump         Quit from keyboard
Yes
4         SIGILL            Core dump         Illegal instruction
Yes
5         SIGTRAP           Core dump         Trace/breakpoint trap
Yes
6         SIGABRT           Core dump         Abort signal from
abort(3)                Yes
7         SIGBUS            Core dump         Bus error (bad memory
access)             Yes
8         SIGFPE            Core dump         Floating-point exception
Yes
9         SIGKILL           Terminate         Kill signal (cannot be
caught or ignored) Yes
10        SIGUSR1           Terminate         User-defined signal 1
Yes
11        SIGSEGV           Core dump         Invalid memory reference
Yes
12        SIGUSR2           Terminate         User-defined signal 2
```

```
Yes
13        SIGPIPE           Terminate        Broken pipe: write to
pipe with no readers Yes
14        SIGALRM           Terminate        Timer signal from
alarm(2)                Yes
15        SIGTERM           Terminate        Termination signal
Yes
16        SIGSTKFLT         Terminate        Stack fault on
coprocessor               No
17        SIGCHLD           Ignore            Child status has changed
Yes
18        SIGCONT           Continue          Continue the stopped
process            Yes
```

- ...continue

```
19        SIGSTOP           Stop              Stop process (cannot be
caught or ignored) Yes
20        SIGTSTP           Stop              Stop typed at terminal
Yes
21        SIGTTIN           Stop              Terminal input for
background process      Yes
22        SIGTTOU           Stop              Terminal output for
background process     Yes
23        SIGURG            Ignore            Urgent condition on
socket                Yes
24        SIGXCPU           Core dump        CPU limit exceeded
Yes
25        SIGXFSZ           Core dump        File size limit exceeded
Yes
26        SIGVTALRM         Terminate        Virtual timer expired
Yes
27        SIGPROF           Terminate        Profiling timer expired
Yes
28        SIGWINCH          Ignore            Window size change
Yes
29        SIGIO             Terminate        I/O is available for
asynchronous input or output Yes
30        SIGPWR            Terminate        Power failure or system
restart          Yes
31        SIGSYS            Core dump        Bad system call
Yes
34        SIGRTMIN          Terminate        First real-time signal
Yes
35        SIGRTMIN+1        Terminate        Second real-time signal
Yes
...       ...               ...               ...
...
63        SIGRTMAX-1        Terminate        Second-to-last real-time
signal            Yes
64        SIGRTMAX          Terminate        Last real-time signal
```

```
Yes
```

Remember the process control?

- **ctrl + z** - stop current process
- `myprogram &` - starts myprogram on background
- `myprg1 && myprg2` – if „myprg1“ return exit code 0 (OK) then run „myprg2“
  - examine yourself what || usage (instead of &&) does
- `fg [job spec …]` - run process on foreground
- `bg [job_spec …]` - run process on background
- `jobs` - show process list (for current instance of shell)
- `echo $?` - return exit code of last command
- `echo $$` - returns $PID (Process ID)
- …
  - `$$` holds the PID of the current script or shell.
  - `$?` holds the exit status of the last executed command.
  - `$!` holds the PID of the last background command.

Exercise

- Look at the structure of `/etc/inittab`
  - Disable reload on **CTRL+ALT+DEL**
  - Set „3“ as default run level
- Make your OS to copy `/etc/rc.local` to `/tmp` after the system startup
- Start ssh daemon/service listening on port 9999
  - Use `netstat -l` to check if process is listening on given port
  - Terminate this process by sending Terminate signal
- Make a custom service run after the system startup
  - you can use python simple http server for example
    - Python 2 —`python -m SimpleHTTPServer 8000`
    - Python 3 — `python -m http.server 8000`
- Advanced: wrap the service in a watch script to restart it after the eventual crash