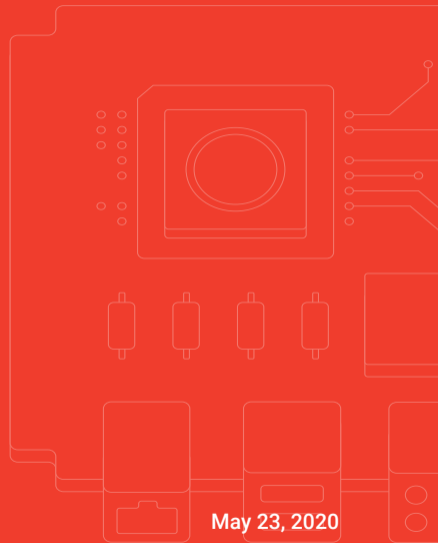


Zagreb, Cloud analysis, Algebra

Container technologies

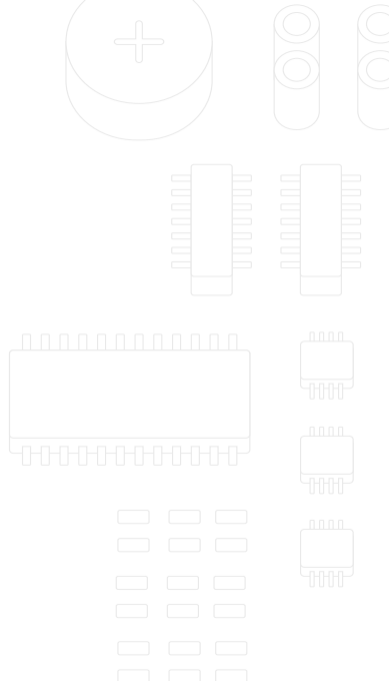
Davor Popović · Marko Ratkaj

sartura



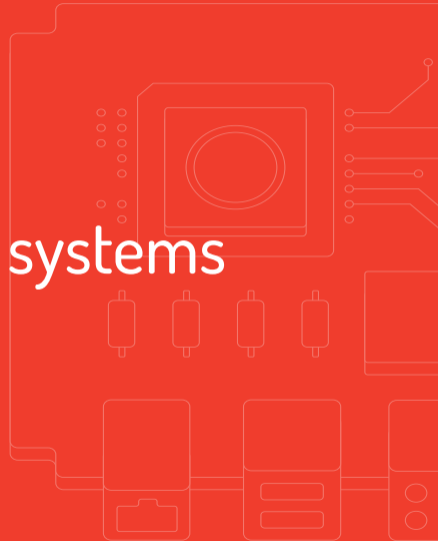
About us

- Embedded Linux development and integration
- Delivering solutions based on Linux, OpenWrt and Yocto
 - Focused on software in network edge and CPEs
- Continuous participation in Open Source projects
- www.sartura.hr



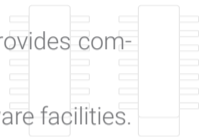
Introduction to operating systems

sartura



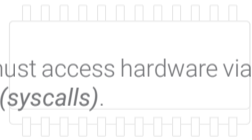
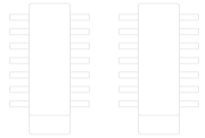
Operating system

- System software that manages computer hardware, software resources, and provides common services for computer programs.
- Operating system(s) provide useful abstractions for controlling underlying hardware facilities.
- These facilities can be shared at once between multiple users or programs.
- Typical hardware facilities controlled by the operating system(s) are:
 - Processor
 - RAM (primary memory)
 - Disks (hard disks, solid state disks)
 - Network interfaces
 - Input/output devices (display, keyboard, sound, mouse, ...)





- Operating systems have two spaces of operation:
 - Kernel space (privileged space) where the kernel operates.
 - Operates in a protected part of the memory.
 - Has full control over all hardware resources.
 - User space (unprivileged space) where all applications run.
 - Access to hardware resources restricted or forbidden.
 - To gain access, applications running in the user space must access hardware via the kernel with specialized set of calls named **system calls (syscalls)**.



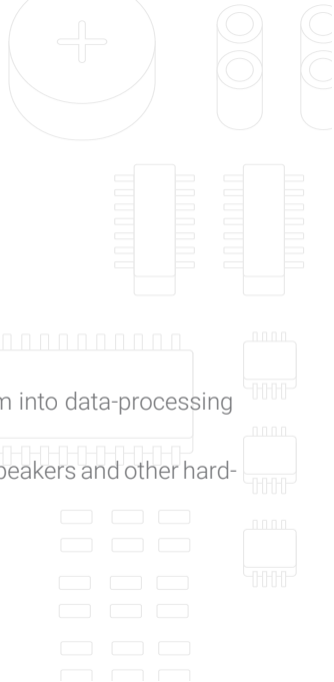
- Each space of operation contains certain components of the operating systems:
 - Kernel space contains the kernel – a program which controls all hardware resources.
 - Kernel is in charge of all operations over hardware – e.g. accessing the disk storage, accessing and organizing RAM for other programs, allowing access to displays, sound, and other devices,
- User space contains applications and libraries
 - Applications = programs written by programmers for performing specific tasks (word processors, multimedia processors, file explorers, Internet browsers, graphical user interfaces, ...)
 - Libraries = sets of resources used for writing programs.

○ Picture



Kernel

- The central core of the operating system.
- Complete control over everything that occurs in the system.
- First part of the OS to load into memory during the booting process
 - Handles startup
 - Handles input/output requests from software and translates them into data-processing instructions for the CPU.
 - Handles memory and peripherals (keyboards, monitors, printers, speakers and other hardware).




- Kernel code is loaded into a protected part of the memory to restrict other applications from overwriting the kernel at any point in time.
- Kernel code runs in a part of memory called kernel space.
 - Cannot be directly accessed from the user space.
 - Runs operating system kernel, kernel extensions, and most device drivers.
 - This separation prevents user data and kernel data from interfering with each other.
 - Prevents malfunctioning applications from crashing the entire OS.

System calls

- If a user space program wants to access a part of the hardware, it must contact the kernel first using a system call.
- Example: opening a file
 - Files are stored on the secondary memory (SSD, HDD) and to open a file (for read, write, append) hard disk as a hardware component must be accessed.
 - Kernel knows how to access the hard disk because it has appropriate code for accessing the hard disk written into it (code segments which allow interaction with a specialized hardware are called device drivers or simply drivers).
 - Before writing user space code, programmers must be familiar with libraries or APIs – collections of resources (functions and function calls) allowing programmers access to different parts of the system.

- On most systems today, there is a library written in C which is called *glibc* and contains code for accessing parts of the system.
- The functions which C programmers can use in their programs are defined in header files – parts of the library code which programmers can directly include into their code to utilize sets of functions for different operations.
- One header in glibc library is called `stdio.h` – contains a list of functions for allowing access to input/output devices.

1  Simple C program for opening a file:
2 `#include <stdio.h>`
3 `int main(void) {`
4 `FILE *fp;`
5 `fp = fopen("write.txt", "w");`
6
7 `return 0;`
8 `}`



- One of the functions defined in `stdio.h` is the `fopen` function which opens a file (if the file does not exist it will be created).
- The only thing the programmer must do is to call `fopen` to open a file.

- Looking into `stdio.h`:

```
1 FILE *  
2 fopen(const char * __restrict file, const char * __restrict mode)  
3 {  
4     ...  
5     if ((f = _open(file, oflags, DEFFILEMODE)) < 0) {  
6         fp->_flags = 0;    /* release */  
7         return (NULL);  
8     }  
9     ...
```

- This open function is a syscall telling the kernel to open this file for writing mode.

- The kernel then opens this file by invoking the driver for hard disk and invoking the underlying file system.
- If the kernel is successful with opening a file, it will return the file to the programmer for further operations.
- Every modern OS has its own set of system calls that are allowed towards the kernel:
 - Linux and OpenBSD each have over 300 different calls, NetBSD has close to 500, FreeBSD has over 500, Windows 7 has close to 700.
- Other syscall examples: read, write, close, wait, exec, fork, exit, and kill.

- Basic principle of every system call
- Drawing



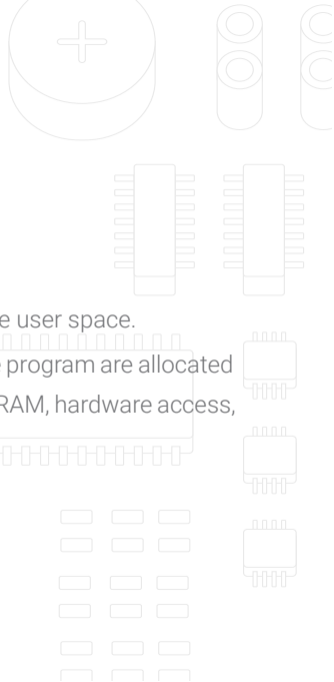


- System calls provide the following services from the kernel to applications:
 - Process creation and management
 - Main memory management
 - File Access, directory and file system management
 - Device handling (I/O)
 - Protection
 - Networking



Processes

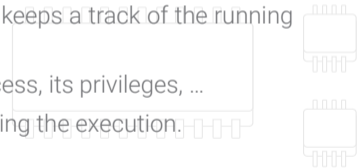
- Processes – programs in execution.
- Operating systems provide an environment for running programs in the user space.
- When a program is run, the kernel ensures all required resources for the program are allocated
 - Memory in RAM, loading the program binary file into the allocated RAM, hardware access, process priority, ...





○ A typical program consists of:

- Machine code of the program – an executable file
- Memory – a region of (virtual) memory used by the process which includes space for the program code, space for the program's subroutines (stack) and space for keeping the computation data generated during the program running time (heap).
- Descriptor – unique way using which the operating system keeps a track of the running process (process ID).
- Security attributes – information on who is running the process, its privileges, ...
- Processor state – a state of the processor and memory during the execution.



- The operating system keeps its processes separate and allocates the resources they need.
- The operating system may also provide mechanisms for inter-process communication – how programs communicate between themselves.
- Certain applications can show you the state of the processes running on the operating system – Task manager on Windows, (h)top on Linux

```

1  [|||||] 10.7% 5 [||||] 5.9%
2  [|||||] 7.7% 6 [||||] 3.3%
3  [|||||] 7.6% 7 [||||] 6.5%
4  [|||||] 12.6% 8 [||||] 6.4%
Mem[|||||] Tasks: 156, 1021 thr: 2 running
Swp[|||||] 8.62M/20.0G Load average: 0.60 0.59 0.67
Uptime: 18 days, 08:56:08

PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
136507 dpopovic 20 0 14036 6432 3968 S 0.0 0.0 0:00.69 -zsh
1667329 dpopovic 20 0 20152 12304 5808 S 0.0 0.1 0:29.91 |
| vim CMakeLists.txt
106352 dpopovic 20 0 124M 24312 11764 S 0.0 0.1 0:07.20 -zsh
106353 dpopovic 20 0 14100 6508 3980 S 0.0 0.0 0:01.66 |
| -zsh
25169 dpopovic 20 0 145M 56216 10388 S 0.0 0.3 2:18.55 |
| urxvt
25176 dpopovic 20 0 14024 6276 3772 S 0.0 0.0 0:00.50 |
| -zsh
4455 dpopovic 20 0 124M 66020 11100 S 0.0 0.4 1:03.62 |
| urxvt
4456 dpopovic 20 0 15448 7944 4008 S 0.0 0.0 0:12.72 |
| -zsh
1449 dpopovic 20 0 11408 3284 3068 S 0.0 0.0 0:00.00 -zsh
1546 dpopovic 20 0 1094M 326M 123M S 1.9 2.1 2h04:53 |
| chromium
4177305 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.00 |
| chromium
3602171 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.00 |
| chromium
3602176 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.01 |
| chromium
3438767 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 7:13.56 |
| chromium
2562105 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.00 |
| chromium
1915691 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.96 |
| chromium
1903658 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:13.79 |
| chromium
29674 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.00 |
| chromium
17642 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:01.25 |
| chromium
16281 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.14 |
| chromium
2897 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.63 |
| chromium
2239 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.61 |
| chromium
1927 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:05.61 |
| chromium
1759 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.00 |
| chromium
1715 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.00 |
| chromium
1674 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:12.76 |
| chromium
1665 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.79 |
| chromium
1662 dpopovic 20 0 547M 211M 88970 S 1.9 1.3 56:49.57 |
| /usr/lib/chromium/chromium --type=utility --field-trial-handle=13580019831686422627,170362767731344
1913625 dpopovic 20 0 547M 211M 88970 S 0.0 1.3 0:00.89 |
| /usr/lib/chromium/chromium --type=utility --field-trial-handle=13580019831686422627,170362767731
1903485 dpopovic 20 0 547M 211M 88970 S 0.0 1.3 0:03.05 |
| /usr/lib/chromium/chromium --type=utility --field-trial-handle=13580019831686422627,170362767731
17839 dpopovic 20 0 547M 211M 88970 S 0.0 1.3 0:06.17 |
| /usr/lib/chromium/chromium --type=utility --field-trial-handle=13580019831686422627,170362767731
1680 dpopovic 20 0 547M 211M 88970 S 0.0 1.3 0:00.00 |
| /usr/lib/chromium/chromium --type=utility --field-trial-handle=13580019831686422627,170362767731
1678 dpopovic 20 0 547M 211M 88970 S 1.3 1.3 51:57.23 |
| /usr/lib/chromium/chromium --type=utility --field-trial-handle=13580019831686422627,170362767731
1676 dpopovic 20 0 547M 211M 88970 S 0.0 1.3 0:02.40 |
| /usr/lib/chromium/chromium --type=utility --field-trial-handle=13580019831686422627,170362767731
1661 dpopovic 20 0 1094M 326M 123M S 0.0 2.1 0:00.00 |
| chromium

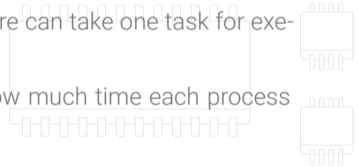
```

FIGURE 1 htop



○ Process management

- Modern computers can run multiple processes at once.
- This requires multitasking – sharing the processor between processes.
- If a computer contains one core processor, when one process goes idle another process can jump in and consume processor.
- If a computer contains a multi core processor, then each core can take one task for execution and execute multiple processes at the same time.



- The kernel contains a scheduling program which determines how much time each process spends executing.



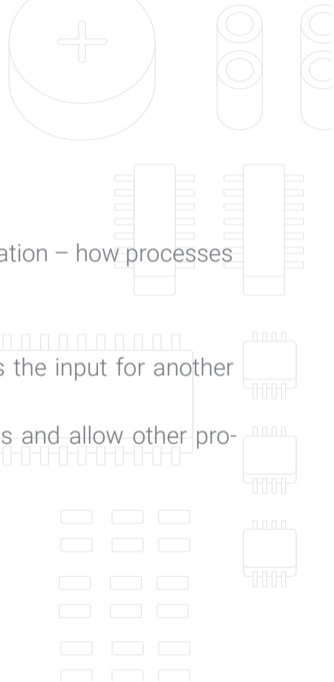
- On early OSs the allocation of time to programs was called cooperative multitasking.
 - Cooperative multitasking allows a program to take as much time as it needs for execution and then return the control back to other program(s).
 - Disadvantage?
- Modern OSs support preemptive multitasking
 - All programs are limited in how much time they are allowed to spend on the CPU without being interrupted.
- Linux and Unix are preemptive multitasking operating systems, Windows got their first support for preemptive multitasking with Windows NT but full support was added with Windows XP.
- Because of multitasking, processes are segregated by states: running, waiting, blocked, terminated, ...

○ Picture



IPC

- Operating systems usually provide the way of interprocess communication – how processes talk between each other.
- This can be achieved in multiple ways:
 - Pipes – which allow the output from one process to be used as the input for another process.
 - Buses – programs which will catch messages from one process and allow other processes to catch those messages for their own purposes



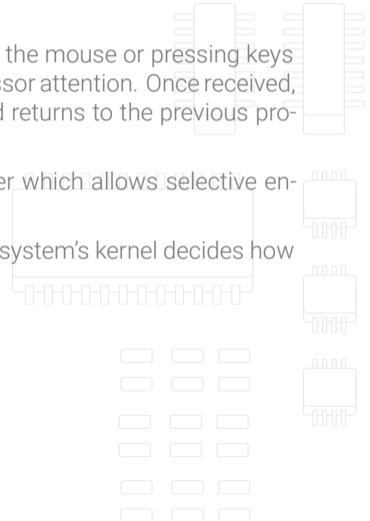
Interrupts

- Interrupts – an event for the processor signaling it has to stop the current process, switch to another process and after finishing this process, it can return back on executing the original process.
- Interrupts are commonly used by hardware devices to indicate electronic or physical state changes that require action.
- Common in multitasking systems and in real-time operating systems.
- We can divide interrupts into hardware or software interrupts.



○ Hardware interrupts

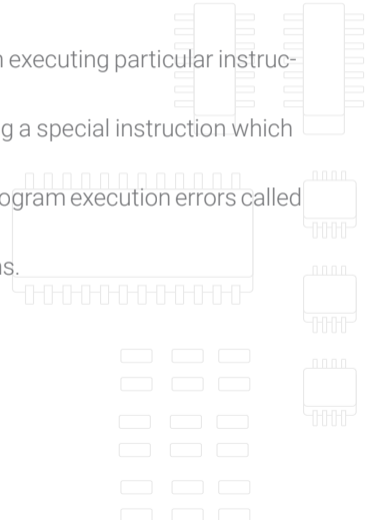
- Electronic signals from hardware components (e.g. moving the mouse or pressing keys on keyboard triggers hardware interrupts) that require processor attention. Once received, processor saves its current state, handles the interrupt, and returns to the previous process.
- Processors typically have an internal interrupt mask register which allows selective enabling and disabling of hardware interrupts.
- When a hardware device triggers an interrupt, the operating system's kernel decides how to deal with this event.





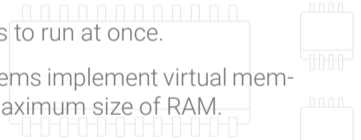
○ Software interrupts

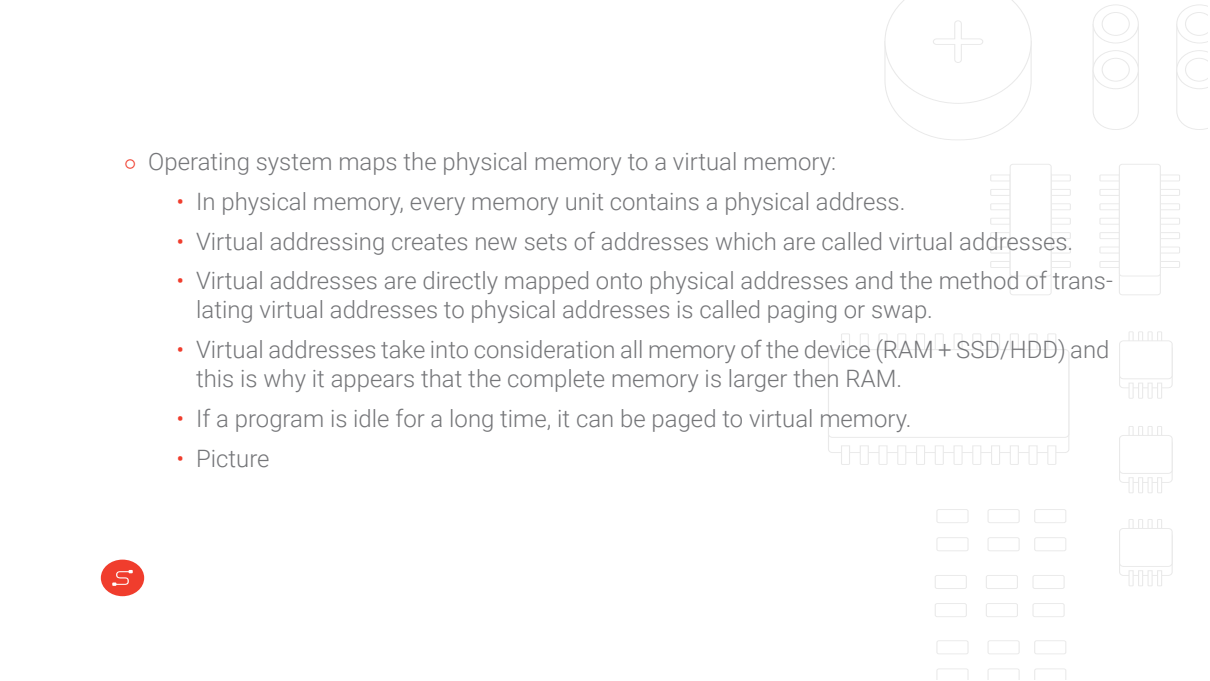
- A software interrupt is requested by the processor itself upon executing particular instructions or when certain conditions are met.
- A software interrupt may be intentionally caused by executing a special instruction which invokes an interrupt.
- Software interrupts may also be unexpectedly triggered by program execution errors called traps or exceptions (e.g. division by 0).
- The operating system will catch and handle these exceptions.



Memory management

- Allocation of memory blocks for starting processes, executing and stopping processes (freeing the memory for further use by another process).
- A vital part of the operating system is to allow multiple processes to run at once.
- To increase the memory management today most operating systems implement virtual memory – a method for expanding the size of the memory over the maximum size of RAM.



- 
- Operating system maps the physical memory to a virtual memory:
 - In physical memory, every memory unit contains a physical address.
 - Virtual addressing creates new sets of addresses which are called virtual addresses.
 - Virtual addresses are directly mapped onto physical addresses and the method of translating virtual addresses to physical addresses is called paging or swap.
 - Virtual addresses take into consideration all memory of the device (RAM + SSD/HDD) and this is why it appears that the complete memory is larger than RAM.
 - If a program is idle for a long time, it can be paged to virtual memory.
 - Picture

- Virtual memory handling is usually handled by a part of processor called MMU (memory management unit)
- Users are unaware when paging happens because it is handled by the kernel in the background of process execution.
- If there are too many processes running and consuming too much of memory at once, then the constant swap/paging between HDD and RAM can slow things down (especially on HDDs).
- In virtual memory systems, the OS limits how processes access the memory – this is called memory protection.
- Memory protection is used to disallow a process to read or write to memory not allocated to it, preventing malicious or malfunctioning code in one program from interfering with the operation of another.
- Processes sometimes need to communicate over the memory and this memory part is then called a shared memory.

- If a program tries to access a part of memory which is not mapped into virtual memory, the hardware calls a page fault – an exception – then the kernel must handle it (e.g. if something is stored on the disk but for some reason it has not been loaded into virtual memory as if it does not exist).
 - “Major page faults on conventional computers using hard disk drives for storage can have a significant impact on performance, as an average hard disk drive has an average rotational latency of 3 ms, a seek time of 5 ms, and a transfer time of 0.05 ms/page. Therefore, the total time for paging is near 8 ms (= 8,000 μ s). If the memory access time is 0.2 μ s, then the page fault would make the operation about 40,000 times slower. ”

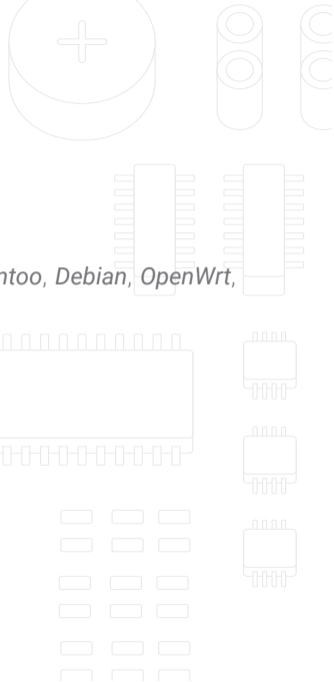


Introduction to GNU/Linux

sartura

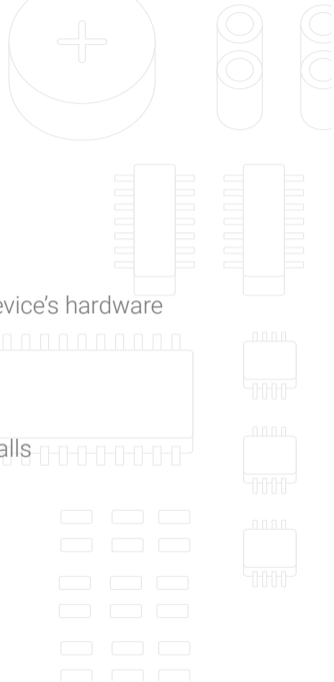


- Linux = operating system kernel
- GNU/Linux distribution = kernel + userspace (*Ubuntu, Arch Linux, Gentoo, Debian, OpenWrt, Mint, ...*)
- Userspace = set of libraries + system software



Linux kernel

- Operating systems have two spaces of operation:
 - **Kernel space** – protected memory space and full access to the device's hardware
 - **Userspace** – space in which all other application run
 - Has limited access to hardware resources
 - Accesses hardware resources via kernel
 - Userspace applications invoke kernel services with system calls



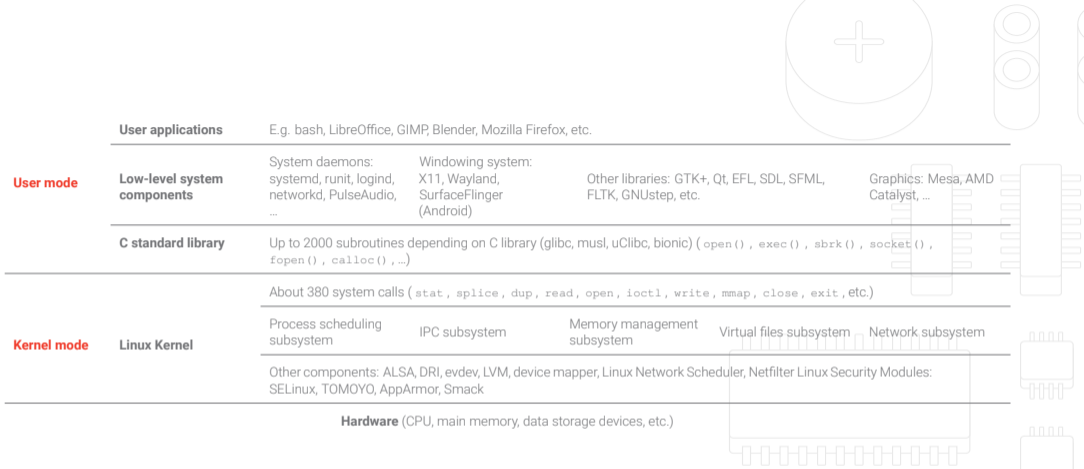
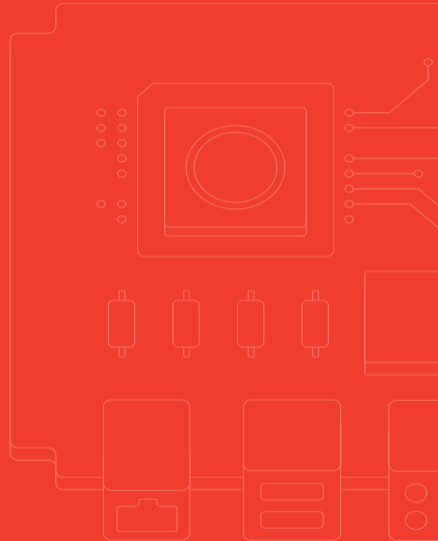


FIGURE 2 Layers within Linux

Virtualization

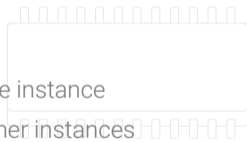
sartura



Virtualization Concepts

Two virtualization concepts:

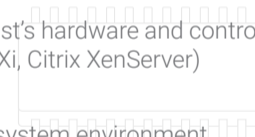
- *Hardware virtualization* (full/para virtualization)
 - Emulation of complete hardware (virtual machines - VMs)
 - VirtualBox, QEMU, etc.
- *Operating system level virtualization*
 - Utilizing kernel features for running more than one userspace instance
 - Each instance is isolated from the rest of the system and other instances
 - Method for running isolated processes is called a *container*
 - *Docker, LXC, Solaris Containers, Microsoft Containers, rkt*, etc.



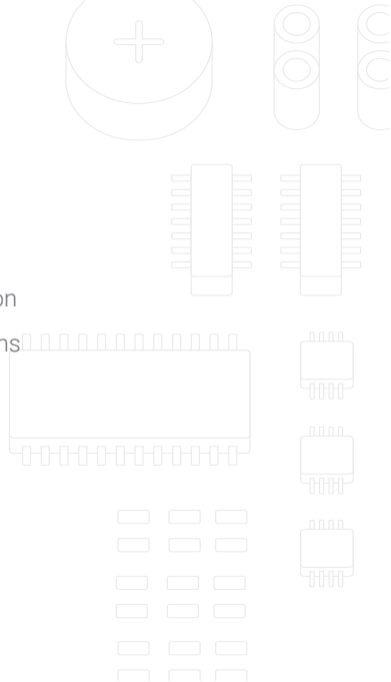


Virtual machines use hypervisors (virtual machine managers – *VMM*)

- Allows multiple guest operating systems (OS) to run on a single host system at the same time
- Responsible for resource allocation – each VM uses the real hardware of the host machine but presents system components (e.g. CPU, memory, HDD, etc.) as their own
- Two types of hypervisors (VMMs)
 - Type 1
 - Native or bare-metal hypervisors running directly on host's hardware and controlling the resources for Vms (Microsoft Hyper-V, VMware ESXi, Citrix XenServer)
 - Type 2
 - Hosted hypervisors running within a formal operating system environment.
 - Host OS acts as a layer between hypervisor and hardware.



- Containers do not use hypervisors
- Containers sometimes come with *container managers*
 - Used for managing containers rather than resource allocation
- Containers use direct system calls to the kernel to perform actions
 - Kernel is shared with the host



- Idea behind containers is to pack the applications with all their dependencies and run them in an environment that is isolated from the host
- Two types of containers:
 - Full OS containers – contain full root file system of the operating system
 - Meant to run multiple applications at once
 - Provide full userspace isolation
 - *LXC, systemd-nspawn, BSD jails, OpenVZ, Linux VServer, Solaris Zones*
 - Application containers – contain an application which is isolated from the rest of the system (*sandboxing*)
 - Application behaves at runtime like it is directly interfacing with the original operating system and all the resources managed by it
 - *Docker, rkt*

Parameter	VMs	Containers
Size	Few GBs	Few MBs
Structure	Full contained environment	Rely on underlying OS
Resources	Contains full OS with no dependencies on the underlying OS (e.g. Windows running on Linux and vice-versa)	Rely on underlying OS
Boot time	Few second overhead	Millisecond overhead

FIGURE 3 VMs vs containers - Differences

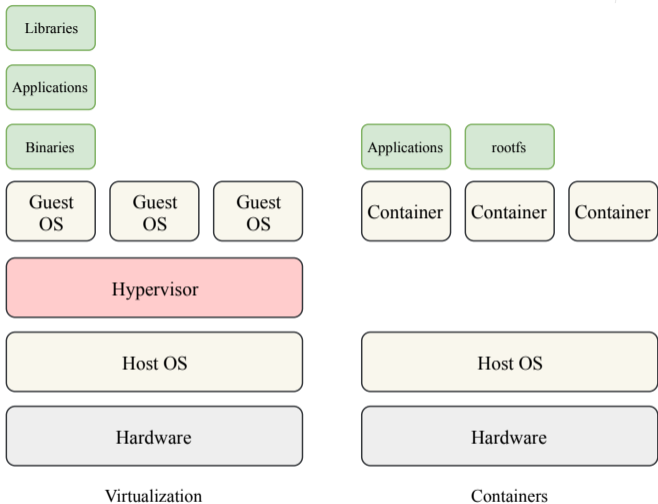


FIGURE 4 Virtualization vs Containers

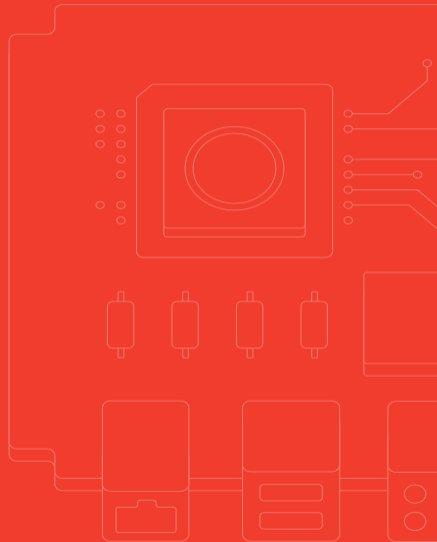
Why use virtualization?

- Cost effective, resource savings
 - Multiple machines can be virtualized on a single machine
- Management
 - Everything can be managed from a single point, usually using a management software for virtual machines and/or containers
- Maintenance
 - Once deployed, machines can be easily switched for new machines if needed in the future



Linux Features

sartura



Namespaces

- Lightweight process virtualization
- Namespaces = way of grouping items under the same designation
 - Kernel feature which organizes resources for processes
 - One process sees one set of resources
 - Another process sees another set of resources
 - One process cannot see other processes' set of resources
 - Each process has its own namespace – a set of resources uniquely allocated for that process
 - Namespaces allow processes to see the same parts of the system differently

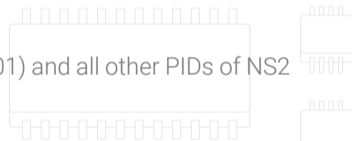
Namespaces

- GNU/Linux kernel supports the following types of namespaces:
 - network
 - uts
 - PID
 - mount
 - user
 - time



Namespaces - PID

- PID namespaces kernel feature enables isolating PID namespaces, so that different namespaces can have the same PID
- Kernel creates two namespaces - NS1 and NS2
- NS1 contains PID 1 and all other PIDs
- NS2 contains PID1 and all other PIDs
- NS1 can see PID 1 of NS2 but with some other PID (e.g. PID 10001) and all other PIDs of NS2
- NS2 can not see PIDs from NS1
- This way, isolation is achieved from these namespaces – processes inside NS2 are only functional if NS2 is isolated from NS1
 - In NS2 a process cannot send signal (e.g. `kill`) to a host machine



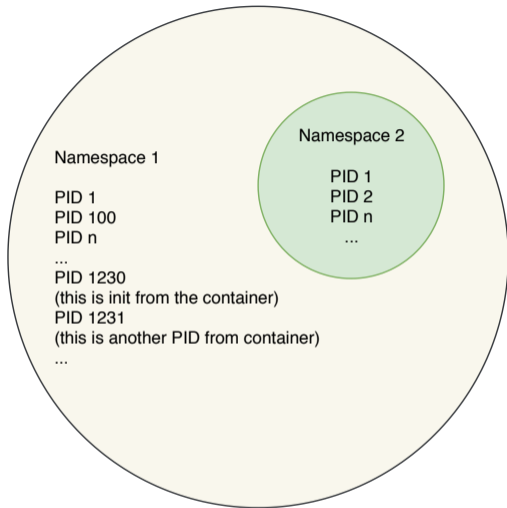
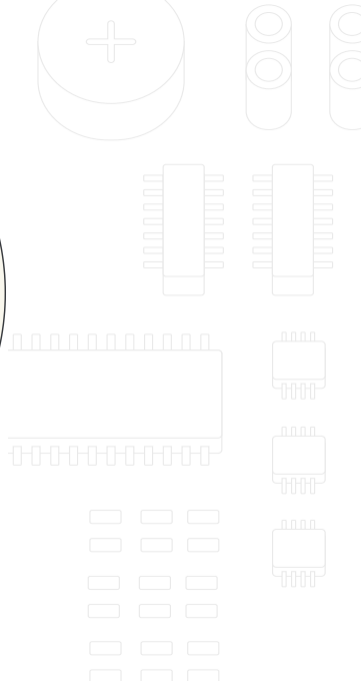


FIGURE 5 Namespaces - PID



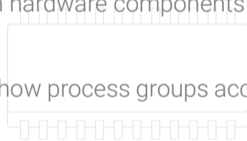
Namespaces — Linux configuration

- Namespace feature requires build-time kernel configuration
- Example configuration from a system with Docker and LXC

```
1  ...  
2  CONFIG_NAMESPACES=y  
3  CONFIG_UTS_NS=y  
4  CONFIG_IPC_NS=y  
5  CONFIG_USER_NS=y  
6  CONFIG_PID_NS=y  
7  CONFIG_NET_NS=y  
8  ...
```

CGroups

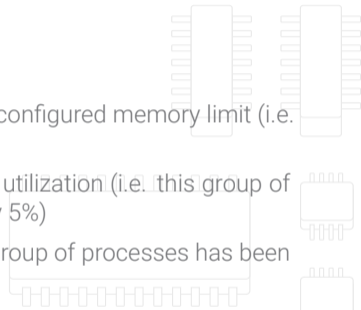
- PID namespace allows processes to be grouped together in isolated environments
- Group of processes (or a single process) needs access to certain hardware components
 - E.g. RAM, CPU, ...
- Kernel provides the control groups (CGroups) feature for limiting how process groups access and use these resources



CGroups

- 4 main purposes

- Limiting resources = groups can be set to not exceed a pre-configured memory limit (i.e. this group of processes can access X MB of RAM)
- Prioritization = some groups may get a larger share of CPU utilization (i.e. this group of processes can utilize 43% of CPU 1, while another group only 5%)
- Accounting = measures a group's resource usage (i.e. this group of processes has been using 5% of CPU)
 - Useful for statistics
- Control = used for freezing, snapshotting/checkpointing and restarting

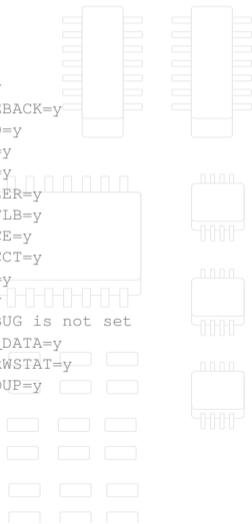


CGroups — Linux configuration



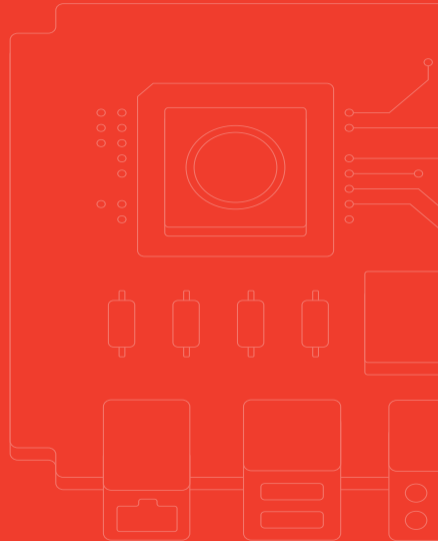
- o CGroups feature requires build-time kernel configuration
- o Example configuration from a system with Docker and LXC

```
1 ...
2 CONFIG_CGROUPS=y
3 CONFIG_BLK_CGROUP=y
4 CONFIG_CGROUP_WRITEBACK=y
5 CONFIG_CGROUP_SCHED=y
6 CONFIG_CGROUP_PIDS=y
7 CONFIG_CGROUP_RDMA=y
8 CONFIG_CGROUP_FREEZER=y
9 CONFIG_CGROUP_HUGETLB=y
10 CONFIG_CGROUP_DEVICE=y
11 CONFIG_CGROUP_CPUACCT=y
12 CONFIG_CGROUP_PERF=y
13 CONFIG_CGROUP_BPF=y
14 # CONFIG_CGROUP_DEBUG is not set
15 CONFIG_SOCK_CGROUP_DATA=y
16 CONFIG_BLK_CGROUP_RWSTAT=y
17 CONFIG_NET_CLS_CGROUP=y
18 ...
```



Linux Containers (LXC)

sartura



- LXC is a userspace interface for the GNU/Linux kernel containment features
 - Allows operating system level virtualization on GNU/Linux systems
- In-between *chroot* and complete VM
 - Sometimes referred to as *chroot-on-steroids*
- Does not depend on hardware support for virtualization
 - Ideal for containerization/virtualization on embedded devices
- Configurable as a full feature file system (rootfs) or minimized for running single applications
- Relies *heavily* on kernel features

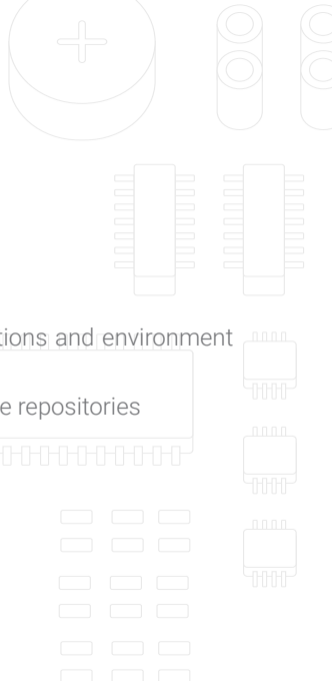
Container security

- LXC uses the following Linux features to improve security:
 - *Namespaces*
 - ipc, uts, mount, pid, network and user
 - user namespaces, privileged and unprivileged containers
 - *Apparmor* and *SELinux* profiles
 - *Seccomp* policies
 - Kernel *capabilities*
 - *CGroups*
 - *Chroots* (using *pivot_root*)



Working with LXC

- Each container needs its own configuration file
- Each container needs its own root file system
 - The root file system contains all the necessary libraries, applications and environment settings
 - Needs to be manually prepared or downloaded from remote online repositories
- Place the configuration file and root file system in the same location
 - `/var/lib/lxc/<container_name>/`



Configuring the container

- `/etc/lxc/default.conf`, `$HOME/.config/lxc/default.conf` or container specific in container directory
- Container configuration defines the following components:
 - Capabilities – what the container is allowed to do from an administrative perspective
 - Cgroups – which resources of the host are allowed for the container (e.g. configuring which devices can the container use)
 - Mount namespaces – which of the host folders/virtual file systems will be allowed for mounting inside the container (virtual file systems such as `proc` or `sys`)
 - Network namespaces – which devices will be created inside the container and how they connect to the outside network

- First part of the file handles capabilities
- A list of all the capabilities dropped (not allowed) for the container
- Usually best to consult with `man` pages <http://man7.org/linux/man-pages/man7/capabilities.7.html>

```
1 lxc.cap.drop = mac_admin
2 lxc.cap.drop = mac_override
3 lxc.cap.drop = sys_admin
4 lxc.cap.drop = sys_boot
5 lxc.cap.drop = sys_module
6 lxc.cap.drop = sys_nice
7 lxc.cap.drop = sys_pacct
8 lxc.cap.drop = sys_ptrace
9 lxc.cap.drop = sys_rawio
10 lxc.cap.drop = sys_resource
11 lxc.cap.drop = sys_time
12 lxc.cap.drop = sys_tty_config
13 lxc.cap.drop = syslog
14 lxc.cap.drop = wake_alarm
```



- The second part is CGroup – what is allowed for this container (as mentioned before, a container can be seen as a set of processes grouped together, meaning this shows what these processes can access)
- It uses traditional Linux designations for devices (try running `ls -l /dev` which will list all the devices with corresponding major:minor numbers)
- E.g. bold entry is for console on PC

```
1 | lxc.cgroup.devices.deny = a
2 | lxc.cgroup.devices.allow = c 1:1 rwm
3 | lxc.cgroup.devices.allow = c 1:3 rwm
4 | lxc.cgroup.devices.allow = c 1:5 rwm
5 | lxc.cgroup.devices.allow = c 5:1 rwm
6 | lxc.cgroup.devices.allow = c 5:0 rwm
7 | lxc.cgroup.devices.allow = c 4:0 rwm
8 | lxc.cgroup.devices.allow = c 4:1 rwm
9 | lxc.cgroup.devices.allow = c 1:9 rwm
10 | lxc.cgroup.devices.allow = c 1:8 rwm
11 | lxc.cgroup.devices.allow = c 1:11 rwm
12 | lxc.cgroup.devices.allow = c 136:* rwm
13 | lxc.cgroup.devices.allow = c 5:2 rwm
14 | lxc.cgroup.devices.allow = c 254:0 rwm
15 | lxc.cgroup.devices.allow = c 10:200 rwm
```



- Some metadata about the container
- Where is the rootfs located
- Hostname of the container
- What to mount from the host
- `/proc` and `/sys`

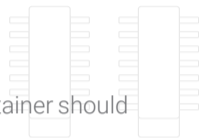
```
1 # Distribution configuration
2 lxc.arch = x86_64
3 # Container specific configuration
4 lxc.rootfs.path = dir:/var/lib/lxc/openwrt/rootfs
5 lxc.uts.name = openwrt
6 # Mount entries
7 lxc.mount.entry = /proc proc /proc nodev,noexec,
8                   nouid 0 0
9 lxc.mount.entry = sysfs sys sysfs default 0 0
```

- Network namespace configuration
- We can read this as follows:
 - Create `eth0` device inside a container
 - Use a `veth` (virtual cable) to connect this `eth0` from container to `lxcbr0` interface (a bridge interface) on the host
- It can be configured in many different ways – depends on the use case

```
1 | # Network configuration
2 | lxc.net.0.type = veth
3 | lxc.net.0.link = lxcbr0
4 | lxc.net.0.flags = up
5 | lxc.net.0.name = eth0
```

Working with LXC

- Once configuration and root file system are ready issue `lxc-ls`
 - If the container is configured properly and its root file system is valid, the container should appear on the list
- Start the container with
`lxc-start -n <container_name>`
- There will be no output, but the container should start
- Check that the container is running
`lxc-info -n <container_name>`
- Container runs in the background and we can now run applications inside it

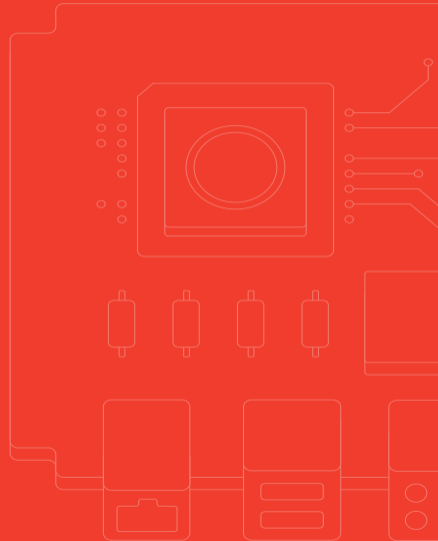


Working with LXC

- Entering the shell of the container (attaching inside of the container)
`lxc-attach -n <container_name>`
- From the shell, it is possible to do everything as on host GNU/Linux system
- To exit, type `exit`
- To stop the container
`lxc-stop -n <container_name>`
- This demonstration is a simple case of a single container created by root user and with no particular functionalities – so what can be done with the container?

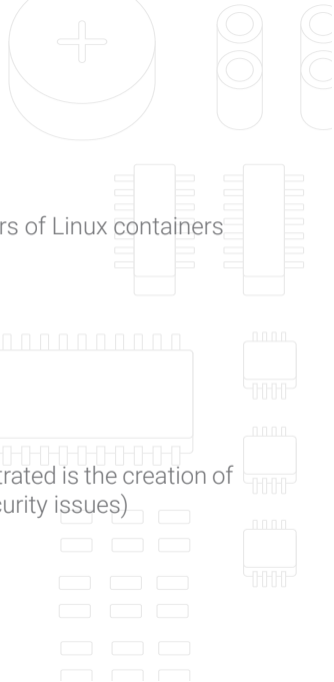
LXD Container Manager

sartura



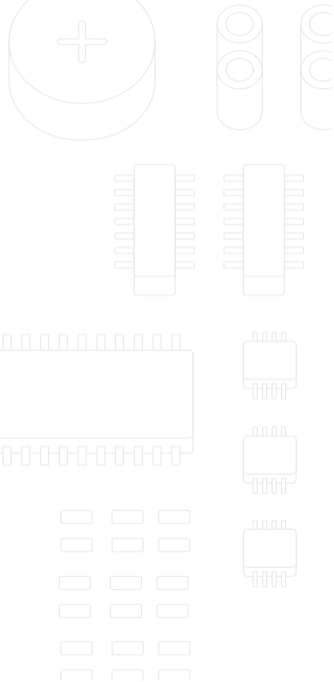
LXD

- Container manager, useful when running and configuring large numbers of Linux containers
- Concept
 - Server + client side (communicating over REST API)
 - Accessible locally and remotely over network
 - Command line tool for working with containers
- Supports the full LXC feature set
 - By default, LXD creates unprivileged containers (what we demonstrated is the creation of privileged containers by the root user which might have some security issues)



LXD - Prerequisites

- Initialized LXD daemon
- Root file system and metadata
 - Metadata = data about the container
- Container image
 - Image from which the container will be created
 - Image = rootfs + metadata
- Container profile
 - Basic container configuration



LXD init

o `lxd init` Configuring the LXD daemon



Prepare rootfs

- 1 ○ Create `gentoo` directory inside the home directory

```
2 | cd ~  
   | mkdir gentoo
```

- Copy compressed root file system on that location

```
cp gentoo-rootfs.tar.gz ~/gentoo
```

- In the same directory, create metadata file for the container

- Metadata describes basic information about the container
- Can be written in YAML format or JSON (examples below use YAML)



Minimal metadata template file

```
1 vim metadata.yaml
2 architecture: "aarch64"
3 creation_date: 1554382805      # Mandatory, must be unique for each container. Take this value:
4     date +%s
5 properties:
6     architecture: "aarch64"
7     description: "Example of Gentoo virtual router"
8     os: "Gentoo Linux"
9     release: "0.1"
10    variant: "Custom"
```



Import rootfs and metadata as image

- Compress both image and metadata

```
tar cf gentoo-metadata.tar metadata.yaml
```

- Import compressed root file system and metadata into LXD

```
lxc image import gentoo-metadata.tar.gz gentoo-rootfs.tar.gz --alias GentooImage
```

- If everything went well, the image should appear on LXD image list

```
lxc image list
```

Prepare container profile



o Create minimal YAML file to define the container profile: `vim gentoo-profile.yaml`

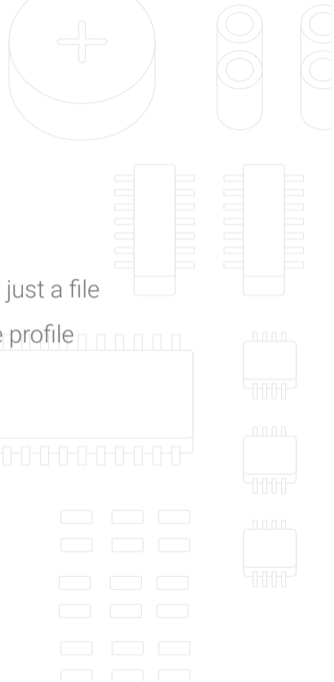
```
1 config: {}
2 description: Gentoo LXD profile
3 devices:
4   eth0:
5     name: eth0
6     nictype: macvlan
7     parent: eth1 #this can vary, depending on how is the interface named on host (enpXXX,
8     ethXXX, enoXXX...)
9     type: nic
10  root:
11    path: /
12    pool: workstation-pool
13    type: disk
14  name: default
```



LXD

- At this point the profile is not attached to any container, and is actually just a file
- First step is to create a profile for LXD and apply the YAML file with the profile
- This profile can be used over n number of containers

```
1 lxd profile create Gentoo-profile  
2 lxd profile edit Gentoo-profile < gentoo-profile.yaml
```



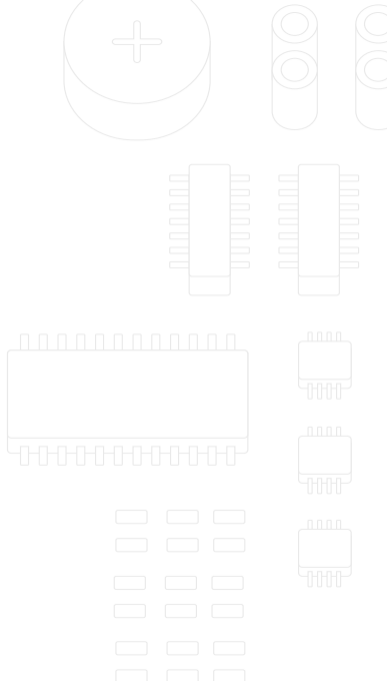
LXD

- Next step is to apply the profile to a container
- First, the container must be created from the image

```
lxc init GentooImage GentooContainer
```

- Then, apply the profile to the initialized container

```
lxc profile apply Gentoo-Profile GentooContainer
```



Verifying the process

- To verify what has been done (and if it went OK)

- Checking images
`lxc image list`

- Checking containers
`lxc ls`

- Checking available profiles
`lxc profile list`



Verifying the process

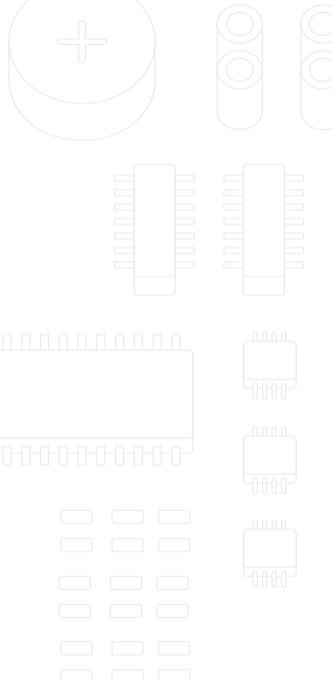
- If necessary, profiles can be modified on the fly and all changes applied in real time

- Checking a specific profile
`| ixc profile show gentoo-profile`
- Modifying a specific profile
`| ixc profile edit gentoo-profile`



Starting the container

- o The container is ready to start at this point
`lxc start GentooContainer`
- o How does this all fit together?
 - Inspect `htop`
 - Network namespace

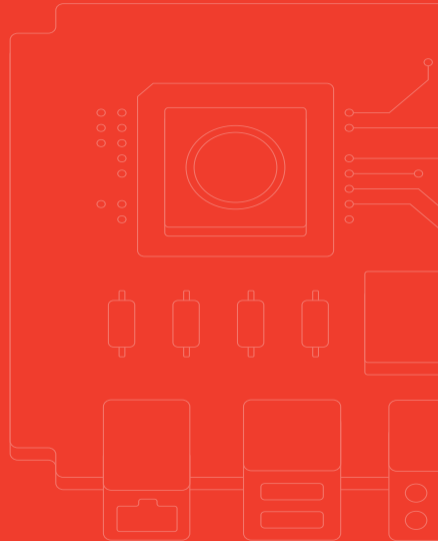


Starting the container - next steps

- Run any application inside the container
 - To attach inside the container, execute `bash`
`1xc exec CentOSContainer /bin/bash`
 - From this shell we can do everything as regular Linux users
 - Any other application can be run in the same way
`1xc exec CentOSContainer /bin/bash`
- With this principle different servers and applications can be run inside the container to isolate them from the rest of the host system

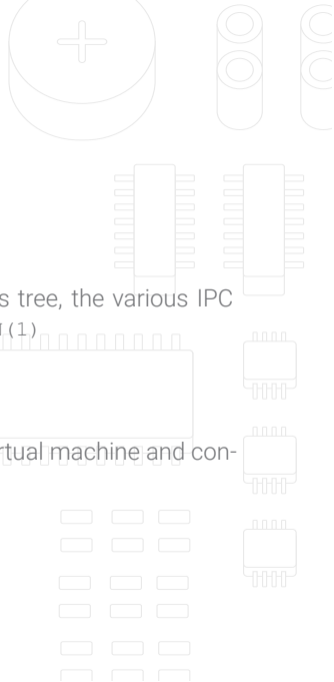
systemd-nspawn

sartura



systemd-nspawn

- Part of the *systemd* project – a *chroot* alternative
 - “Fully virtualizes the file system hierarchy, as well as the process tree, the various IPC subsystems and the host and domain name” — `SYSTEMD-NSPAWN(1)`
- Concept
 - Command line tool for working with containers
 - Integrates with *systemd* on the host via the *systemd-machined* virtual machine and container registration manager



systemd-nspawn — Prerequisites

- *systemd-nspawn* does not require special metadata to boot containers
- Usually, container root filesystem directory is placed in the `/var/lib/machines` directory, which can also be a symlink to a directory
- Container filesystems for some distributions can be created via appropriate utilities e.g. `pacstrap` for Arch Linux, and `debootstrap` for Debian

Launching containers using the CLI

- Launching a container is straightforward:

```
systemd-nspawn --boot --directory=/var/lib/machine/gentoo
```

- CLI tool allows specifying configuration options such as capabilities and networking

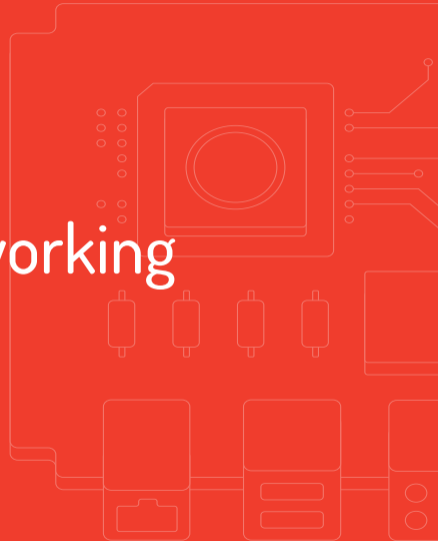
```
1 systemd-nspawn --boot --directory=/var/lib/machine/gentoo  
2   --capability=CAP_NET_ADMIN \  
3   --network-macvlan=eth0
```

Starting containers as a service

- When using the CLI tool containers are foreground processes
- Containers can be run in the background with the `systemd-nspawn@.service` unit
 - Enable and start the `machines.target`
`systemctl enable machines.target`
 - Enable and start the `systemd-nspawn@<machine>.service`, where `<machine>` specifies an nspawn container in `/var/lib/machines`
`systemctl enable --now systemd-nspawn@gentoo.service`
- Optionally configure container in the `/etc/systemd/nspawn/<machine>.nspawn` file

Example: setting up networking

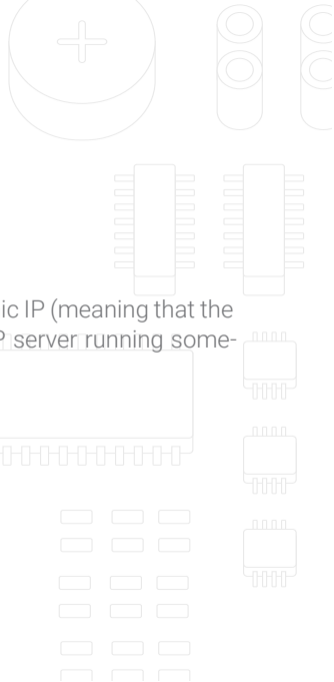
sartura



Setting up network inside the container

- Practical example – setting up the network inside the container
- As defined in the container profile, the interface from the container is connected directly to a physical interface on the host machine with `macvlan` interface
- `macvlan` creates a new interface with a different MAC address than the host one and allows traffic to go directly through (as opposed to a bridge where it has to hit the bridge first)

- In theory, there is nothing wrong with this configuration
- In practice, network has to be configured inside the container as well
- The user can either set up static IP on the inside interface or set dynamic IP (meaning that the IP on the container interface will be offered by someone else – DHCP server running somewhere in the network)

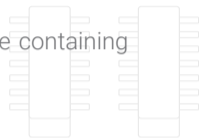




o How?

- `systemd`
- Daemon in role of PID 1 – master process, initial process from which all other processes are spawned
- One of the domains directly under `systemd` control is networking
- As any other program, `systemd` and its components are configured with different configuration files located under `/etc/systemd/(network)`

- Listing out `/etc/systemd/network` might show that it is empty so a new file containing network configuration must be created
- A good practice is to name the file `<file_name>.network`
- This file will define the following:
 - Match the given interface
 - Assign it with IPv4 address from a DHCP server



○ Create a file

```
vim /etc/systemd/network/eth0.network
```

○ Add the following:

```
1 [Match] # Match this interface
2 Name=eth0
3
4 [Network] # Assign IPv4 address from a DHCP server
5 DHCP=ipv4
6
7 [DHCP]
8 RouteMetric=10
```



- Restart systemd networking service

```
systemctl restart systemd-networkd
```

- At this point, on the `eth0` interface an IP address should appear and it should be from the same subnet as the IP address offered on the physical interface of the host

- Try pinging the Internet

```
ping 8.8.8.8
```



Container technologies

Davor Popović · Marko Ratkaj

Feedback form: forms.gle/WKPBDoS69gssafAE9



info@sartura.hr · www.sartura.hr

