Serviços

A unit configuration file whose name ends in ".service" encodes information about a process controlled and supervised by systemd.

multi-user.target is basically the closest equivalent of classic SysVinit runlevel 3 that systemd has. When a systemd system boots up, systemd is trying to make the system state match the state specified by default.target - which is usually an alias for either graphical.target or multi-user.target.

multi-user.target normally defines a system state where all network services are started up and the system will accept logins, but a local GUI is not started. This is the typical default system state for server systems, which might be rack-mounted headless systems in a remote server room.

`systemctl` commands

```
systemctl is-enabled elasticsearch.service
systemctl status elasticsearch --no-pager
systemctl enable --user elasticsearch
systemctl daemon-reload
systemctl start tvsapp.socket
```

- enabled - a service (unit) is configured to start when the system boots
- disabled - a service (unit) is configured to not start when the system boots
- active - a service (unit) is currently running.
- inactive - a service (unit) is currently not running, but may get started, i.e. become active, if something attempts to make use of the service.

A service:

```
[Unit]
Description=daemon for tvsapp service
Requires=nginx.service
Requires=tvsapp.socket

[Service]
ExecStart=/opt/isel/tvs/tvsctld/bin/tvsapp_server

[Install]
WantedBy=multi-user.target
```

Socket

UNIX domain sockets enable efficient communication between processes that are running on the machine. UNIX domain sockets support both stream-oriented, TCP, and datagram-oriented, UDP, protocols. Named pipes are one-way (half-duplex), so you'll need to use two of them in order to do two-way communication. Sockets of course are two way.

To create a UNIX domain socket, use the socket function and specify AF_UNIX as the domain for the socket. The z/TPF system supports a maximum number of 16,383 active UNIX domain sockets at any time. After a UNIX domain socket is created, you must bind the socket to a unique file path by using the bind function. Unlike internet sockets in the AF_INET domain where the socket is bound to a unique IP address and port number, a UNIX domain socket is bound to a file path.

```
struct sockaddr_un {
        unsigned short int sun_family; /* AF_UNIX */
        char sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

Socket server

```
if (main_fd < 0) {
    error("ERROR creating socket");
}
struct sockaddr_un srv_addr;
memset(&srv_addr, 0, sizeof(srv_addr));
srv_addr.sun_family = AF_UNIX;
sprintf(srv_addr.sun_path, "/tmp/%s", argv[1]); // should check max length
unlink(srv_addr.sun_path);  // just in case...

if (bind(main_fd, (struct sockaddr *) &srv_addr, sizeof(srv_addr)) < 0) {
    close(main_fd);
    error("ERROR on binding");
}
if (listen(main_fd, 5) < 0) {
    close(main_fd);
    unlink(srv_addr.sun_path);
    error("ERROR on listen");
}
puts(":: LISTENING ::");
for (;;) {
    struct sockaddr_un cli_addr;
    unsigned int cli_addr_len = sizeof cli_addr;
    int conn_fd = accept(main_fd, (struct sockaddr *)&cli_addr,
&cli_addr_len);
    if (conn_fd < 0) {
        close(main_fd);
        unlink(srv_addr.sun_path);
        error("ERROR on accept");
    }
    printf("server established connection with client\n");
    pthread_t thread;
    pthread_create(&thread, NULL, process_connection, (void *)
(intptr_t)conn_fd);
    pthread_detach(thread);
}
```

```
close(main_fd);
unlink(srv_addr.sun_path);
```

Socket client

```c
int conn_fd = socket(AF_UNIX, SOCK_STREAM, 0);
if (conn_fd < 0) {
    error("ERROR creating socket");
}

struct sockaddr_un srv_addr;
memset(&srv_addr, 0, sizeof(srv_addr));
srv_addr.sun_family = AF_UNIX;
sprintf(srv_addr.sun_path, "/tmp/%s", argv[1]); // should check max length

if (connect(conn_fd, (struct sockaddr *)&srv_addr, sizeof (srv_addr)) < 0)
{
    error("ERROR connecting socket");
}
puts(":: CONNECTED ::");
char line[LINE_LEN+1];
size_t len;
char data[LINE_LEN+1];
size_t dlen;
for (;;) {
    if (!fgets(line, LINE_LEN, stdin)) {
        error("ERROR reading stdin");
    }
    len = strlen(line);
    while (len > 0 && (line[len-1] == '\r' || line[len-1] == '\n')) {
        line[--len] = 0;
    }
    if (write(conn_fd, line, len+1) < 0) {
        error("ERROR writing to socket");
    }

    dlen = read(conn_fd, data, LINE_LEN);

    if (dlen == 0) {
        close(conn_fd);
        break;
    }
    if (dlen < 0) {
        error("ERROR reading from socket");
    }
    data[dlen] = 0;
    puts(data);
}
close(conn_fd);
```

A socket:

```
[Unit]
Description=TVS daemon activation socket

[Socket]
ListenStream=/run/isel/tvsctld/request

[Install]
WantedBy=sockets.target
```

Misc

In addition to sending data, processes may send file descriptors across a Unix domain socket connection using the sendmsg() and recvmsg() system calls. This allows the sending processes to grant the receiving process access to a file descriptor for which the receiving process otherwise does not have access

Signals

- SIGTERM -> Requests a process to terminate but allows it to perform cleanup operations before terminating. kill -SIGTERM 1234
- SIGKILL -> Forces a process to terminate immediately. kill -SIGKILL 1234
- SIGSTOP -> Pauses a process. kill -SIGSTOP 1234
- SIGCONT -> Resumes a paused process. kill -SIGCONT 1234
- SIGHUP -> Sends a signal to a process when the terminal controlling it is closed. kill -SIGHUP 1234
- SIGINT -> Sends an interrupt signal, usually initiated by the user. kill -SIGINT 1234

Docker

Cache

- The only way to force a rebuild is by making sure that a layer before it has changed, or by clearing the build cache using docker builder prune.
- Starting with a parent image that's already in the cache, the next instruction is compared against all child images derived from that base image to see if one of them was built using the exact same instruction. If not, the cache is invalidated.
- For the ADD and COPY instructions, the modification time and size file metadata is used to determine whether cache is valid. During cache lookup, cache is invalidated if the file metadata has changed for any of the files involved.
- Once the cache is invalidated, all subsequent Dockerfile commands generate new images and the cache isn't used.

Layers

- The order of Dockerfile instructions matters. A Docker build consists of a series of ordered build instructions. Each instruction in a Dockerfile roughly translates to an image layer
- When you run a build, the builder attempts to reuse layers from earlier builds

- If your build contains several layers and you want to ensure the build cache is reusable, order the instructions from less frequently changed to more frequently changed where possible.

Running a container

1. **Image Pull (if not available locally):** If the `ubuntu:latest` image is not already available locally, Docker will fetch it from the Docker Hub or the configured container registry. This involves downloading the necessary layers that compose the Ubuntu image.

2. **Container Creation:** Docker creates a new container based on the specified image. This involves setting up the container file system, network configuration, and other container-specific settings. As though you had run a docker container create command manually.

3. **Namespace Isolation:** Docker uses Linux namespaces to isolate the container from the host system. Namespaces provide separate views of resources such as process IDs, network, mount points, and more.

- Allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.
- Docker creates a network interface to connect the container to the default network, since you didn't specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine's network connection.

4. **Container Initialization:** Docker initiates the container by running its default command, which, in this case, is determined by the `CMD` instruction in the Ubuntu image. Since you used the `-it` flags, the container starts an interactive shell within the container.

5. **Terminal Interaction:** Docker starts the container and executes /bin/bash. Because the container is running interactively and attached to your terminal (due to the -i and -t flags), you can provide input using your keyboard while Docker logs the output to your terminal.

6. **Process Execution and Isolation:** The container's user space is executed as an isolated process on the host system. The processes inside the container are separate from those on the host, thanks to namespace isolation.

7. **Resource Management:** Docker employs control groups (cgroups) to manage and limit system resources (CPU, memory, etc.) allocated to the container. Resource constraints specified during container creation are enforced by the Docker runtime.

terminal commands

- docker run -> runs a command in a new container, pulling the image if needed and starting the container.
- docker exec -> runs a new command in a running container. (forma facil de lembrar: exec ~ existing container)
- docker run -d -p 4001:4003 --name tvs-ex1-c1 tvs-ex1
- docker run -it tvs-ex1 /bin/sh
- docker compose up -d --scale webapp=2

run vs cmd vs entrypoint

RUN is an image build step, the state of the container after a RUN command will be committed to the container image. A Dockerfile can have many RUN steps that layer on top of one another to build the image.

CMD is the command the container executes by default when you launch the built image. A Dockerfile will only use the final CMD defined. The CMD can be overridden when starting a container with docker run $image $other_command.

ENTRYPOINT is also closely related to CMD and can modify the way a CMD is interpreted when a container is started from an image. Consider `docker run -i -t --rm -p 80:80 nginx`, Command line arguments to docker run will be appended after all elements in an exec form ENTRYPOINT, and will override all elements specified using CMD.

Both CMD and ENTRYPOINT instructions define what command gets executed when running a container. There are few rules that describe their co-operation.

1. Dockerfile should specify at least one of CMD or ENTRYPOINT commands.
2. ENTRYPOINT should be defined when using the container as an executable.
3. CMD should be used as a way of defining default arguments for an ENTRYPOINT command or for executing an ad-hoc command in a container.
4. CMD will be overridden when running the container with alternative arguments.

nginx & docker

valid=5s is the time the DNS query results are cached. By setting a shorter valid duration, NGINX remains more responsive to changes in DNS configurations. It enables NGINX to adapt quickly to changes in DNS mappings, facilitating dynamic updates within the Docker network, especially in scenarios where container instances may scale up or down, and IP addresses change frequently. After this duration, NGINX will re-query the DNS resolver. In essence this line ensures that NGINX uses Docker's internal DNS resolver (127.0.0.11) for DNS resolution, sets a short duration for DNS record validity, and aids in maintaining accurate and up-to-date DNS resolution within the Docker container environment

dockerfile commands

Always combine RUN apt-get update with apt-get install in the same RUN statement

sing apt-get update alone in a RUN statement causes caching issues and subsequent apt-get install instructions to fail.

```
USER node
ENV NODE_PORT=4003
EXPOSE 4003
# we put "--chown=node:node" because we set the USER before hand, se this gives file permissions to node
# we do individual copy of tvsapp.js to cache it, which makes an extra layer
COPY --chown=node:node /src/tvsapp.js /home/node/app/tvsapp.js
COPY --chown=node:node /src /home/node/app
WORKDIR /home/node/app
RUN npm install
```

```
CMD ["node", "tvsapp.js"]
```

Containers

Do containers have a hypervisor? -> Since containers are isolated, they provide security, thus allowing multiple containers to run simultaneously on the given host. Also, containers are lightweight because they do not require an extra load of a hypervisor

A container are essencially used to isolate processes. They live in the host, but they can't anything outside the container. They have their own filesystem, processes and their own network interfaces.

Docker uses a linux kernel feature called namespaces to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

As explained, there is no need to create entire and new VM's for each container.

you can run containers using different distros of Linux because they do share the same kernel (has to be 3.10 or higher)

The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

Namespaces and control groups

Docker technology is not a replacement for LXC. "LXC" refers to capabilities of the Linux kernel (specifically namespaces and control groups (allows an administrator to allocate resources such as CPU, memory, and I/O bandwidth to groups of processes)) which allow sandboxing processes from one another, and controlling their resource allocations. On top of this low-level foundation of kernel features, Docker offers a high-level tool with several powerful functionalities:

- Portable deployment across machines.
- Versioning
- Sharing

Docker originally used LinuX Containers (LXC), but later switched to runC (formerly known as libcontainer), which runs in the same operating system as its host. This allows it to share a lot of the host operating system resources. Also, it uses a layered filesystem (AuFS) and manages networking.

WSL

Windows Subsystem for Linux (WSL) 2 is a full Linux kernel built by Microsoft, which lets Linux distributions run without managing virtual machines. With Docker Desktop running on WSL 2, users can leverage Linux workspaces and avoid maintaining both Linux and Windows build scripts

The primary differences between WSL 1 and WSL 2 are the use of an actual Linux kernel inside a managed VM, support for full system call compatibility, and performance across the Linux and Windows operating systems.

A traditional VM experience can be slow to boot up, is isolated, consumes a lot of resources, and requires your time to manage it. WSL 2 does not have these attributes.

WSL 2 provides the benefits of WSL 1, including seamless integration between Windows and Linux, fast boot times, a small resource footprint, and requires no VM configuration or management. While WSL 2 does use a VM, it is managed and run behind the scenes, leaving you with the same user experience as WSL 1.

WSL 2 is available on all Desktop SKUs where WSL is available, including Windows 10 Home and Windows 11 Home. The newest version of WSL uses Hyper-V architecture to enable its virtualization.

To support WSL2, Hyper-V has been split up in Windows 10. The hypervisor – known as the "Virtual Machine Platform" is available on all Windows versions and is the minimum that is required to run WSL as a light-weight virtual machine that is tightly integrated with the host.

If you rely on a Linux distribution to have an IP address in the same network as your host machine, you may need to set up a workaround in order to run WSL 2. WSL 2 is running as a hyper-v virtual machine. This is a change from the bridged network adapter used in WSL 1, meaning that WSL 2 uses a Network Address Translation (NAT) service for its virtual network, instead of making it bridged to the host Network Interface Card (NIC) resulting in a unique IP address that will change on restart.

VM's e Linux

Hipervisor tipo 1 - hardware takeover

- Os hipervisores tipo 1 acessam diretamente os recursos subjacentes da máquina. São capazes de implementar as próprias estratégias personalizadas de alocação de recursos para atender as respectivas VMs. Os hipervisores tipo 1 oferecem maior performance para suas VMs. Isso ocorre porque eles não precisam negociar recursos com o sistema operacional nem percorrer a camada do sistema operacional.
- O hipervisor tipo 1, ou hipervisor bare metal, interage diretamente com o hardware subjacente da máquina. O hipervisor bare metal é instalado diretamente no hardware físico da máquina host, não por meio de um sistema operacional. Em alguns casos, o hipervisor tipo 1 é incorporado ao firmware da máquina. O hipervisor tipo 1 negocia diretamente com o hardware do servidor para alocar recursos dedicados às VMs.

Hipervisor tipo 2 - hosted

- Os hipervisores tipo 2 negociam a alocação de recursos com o sistema operacional, tornando o processo mais lento e menos eficiente.
- O hipervisor tipo 2, ou hipervisor hospedado, interage com o hardware subjacente da máquina host pelo sistema operacional da máquina host. Você o instala na máquina, onde ele é executado como uma aplicação.

Paravirtualization

- The hypervisor is the monitor of the virtual machine

- Paravirtualization is a type of virtualization where software instructions from the guest operating system running inside a virtual machine can use “hypercalls” that communicate directly with the hypervisor. This provides an interface very similar to software running natively on the host hardware.
- The main benefits of paravirtualization are where instructions are not compatible with full virtualization or where more immediate access to underlying hardware is required for performance reasons. For timing-critical functions, paravirtualization can provide the speed of native code alongside some of the benefits of virtualization, such as sharing hardware between multiple operating systems.

O que significa o código ser escalável? Quando um código é escalável, significa que ele é capaz de lidar com uma carga crescente de trabalho de maneira eficiente e sem falhar. Com o uso de named pipes o código não é escalável pois os named pipes são unidirecionais e não permitem escrita e leitura ao mesmo tempo.