Università di Messina — CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

Hands-On #10

Container from scratch — costruire un container in C

Last Update: April 15, 2026

Obiettivo

In questo hands-on leggerete, compilerete ed eseguirete `container.c`: un container Linux completo scritto in circa 800 righe di C puro, senza librerie esterne.

Non dovete scrivere codice. Dovete capirlo. Ogni sezione spiega un blocco del sorgente, poi chiede di eseguire un comando e osservare l'output. Alla fine saprete esattamente cosa fa il kernel ogni volta che lanciate `docker run`.

Cosa imparerete

- Come funzionano i **Linux namespace** (PID, UTS, Mount, Net, IPC) e perché sono la base di ogni container.
- Cosa fa `clone()` che `fork()` non fa.
- Come `pivot_root()` isola il filesystem in modo sicuro.
- Come i **cgroups v2** limitano memoria e processi.
- Come la **veth pair** connette container e host senza bridge, usando le **Netlink socket** direttamente in C.

Prerequisiti e setup

Setup — esegui questi comandi prima di iniziare

```
# 1. Scarica il sorgente (o copia il file distribuito dal
   docente)
#    Assicurati di avere container.c nella directory corrente

# 2. Compila
gcc -Wall -Wextra -o container container.c

# 3. Scarica un rootfs Alpine minimale (~3.5 MB)
mkdir -p rootfs
curl -sL https://dl-cdn.alpinelinux.org/alpine/v3.19/releases/
   x86_64/\
alpine-minirootfs-3.19.0-x86_64.tar.gz | tar xz -C rootfs

# 4. Verifica
ls rootfs/    # deve mostrare: bin  dev  etc  home  lib  ...
```

[!] Attenzione

Il programma richiede i privilegi di `root`. Tutti i comandi `./container` vanno eseguiti con `sudo`.

Mappa del codice

Prima di entrare nel dettaglio conviene avere una visione d'insieme. `container.c` è organizzato in blocchi funzionali indipendenti. La tabella seguente mostra l'ordine in cui appaiono nel sorgente e il loro scopo.

Funzione / blocco	Chi la chiama	Cosa fa
`nl_open()`	setup_veth, setup_container_net	Apre socket Netlink
`nl_send()`	veth_create, veth_move_to_ns, if_add_addr	Invia msg al kernel
`nl_addattr()`	varie	Aggiunge attributo TLV
`veth_create()`	setup_veth	Crea coppia veth0/veth1
`veth_move_to_ns()`	setup_veth	Sposta veth1 nel container
`if_set_up()`	setup_veth, setup_container_net	Porta iface UP
`if_add_addr()`	setup_veth, setup_container_net	Assegna IP
`setup_veth()`	main	Orchestrazione rete host
`setup_container_net()`	child_fn	Rete dentro il container
`cleanup_veth()`	main	Rimuove veth0 alla fine
`write_file()`	setup_cgroup	Scrive in sysfs/cgroup
`setup_cgroup()`	main	Limita memoria e PID
`cleanup_cgroup()`	main	Rimuove la directory cgroup
`pivot_root_to()`	child_fn	Cambia il rootfs
`setup_mounts()`	child_fn	Monta proc/sys/dev
`child_fn()`	clone()	Codice del container
`main()`	—	Punto di ingresso

Il flusso di esecuzione è sempre lo stesso: `main()` chiama `clone()`, che biforca il processo in due rami che operano in parallelo. Il **parent** configura cgroup e rete; il **child** esegue `child_fn()` che isola filesystem e rete, poi lancia il comando richiesto.

Parte 1 — Include e costanti (righe 17–57)

_GNU_SOURCE e gli header di sistema

```
17 #define _GNU_SOURCE
18 #include <sched.h>          /* clone(), CLONE_NEW*              */
19 #include <sys/mount.h>
20 #include <sys/syscall.h>    /* SYS_pivot_root                   */
21 #include <linux/rtnetlink.h>/* RTM_NEWLINK, RTM_NEWADDR, IFLA_* */
22 #include <linux/veth.h>     /* VETH_INFO_PEER                   */
```

`_GNU_SOURCE` abilita le estensioni GNU della libreria C. Senza questa macro, `clone()` non sarebbe visibile nell'header `<sched.h>` perché è una syscall Linux-specifica, non POSIX.

Gli header `linux/rtnetlink.h` e `linux/veth.h` espongono le costanti del kernel per le Netlink socket: tipi di messaggio (`RTM_NEWLINK`, `RTM_NEWADDR`), attributi (`IFLA_*`) e strutture dati (`nlmsghdr`, `ifinfomsg`).

Costanti di configurazione

```
46 #define STACK_SIZE  (1024 * 1024)    /* 1 MB stack per il child
       */
47 #define CGROUP_BASE "/sys/fs/cgroup/container_demo"
48 #define HOSTNAME    "container"
49 #define MEM_LIMIT   "67108864"       /* 64 MB
       */
50 #define PIDS_MAX    "32"
51
52 #define VETH_HOST   "veth0"
53 #define VETH_CONT   "veth1"
54 #define IP_HOST     "172.17.0.1"
55 #define IP_CONT     "172.17.0.2"
56 #define IP_PREFIX   24
```

`STACK_SIZE` merita una spiegazione: a differenza di `fork()`, `clone()` non copia lo stack del parent. Dobbiamo allocarne uno manualmente e passarlo come parametro. 1 MB è sufficiente per quasi tutti i casi d'uso.

`CGROUP_BASE` è la directory che creeremo in `/sys/fs/cgroup/`: scrivendo file in questa directory diciamo al kernel di applicare limiti al nostro container.

Parte 2 — Netlink: parlare al kernel in C (righe 67–346)

Cos'è Netlink

Netlink è il canale ufficiale con cui lo userspace configura la rete del kernel Linux. Quando esegui `ip link add veth0 type veth peer name veth1`, il comando `ip` apre internamente un socket `AF_NETLINK` e invia esattamente lo stesso tipo di messaggio che costruiamo noi a mano. La differenza è che noi lo facciamo in C, senza shell.

nl_open() — aprire la connessione con il kernel

```
85 static int nl_open(void) {
86     int fd = socket(AF_NETLINK, SOCK_RAW | SOCK_CLOEXEC,
           NETLINK_ROUTE);
87     ...
88     struct sockaddr_nl sa = {
89         .nl_family = AF_NETLINK,
90         .nl_pid    = 0,   /* 0 = invia al kernel */
91         .nl_groups = 0,
92     };
93     bind(fd, (struct sockaddr *)&sa, sizeof(sa));
94     return fd;
95 }
```

`socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE)` apre un socket Netlink di tipo *ROUTE*: quello che gestisce interfacce di rete, indirizzi IP e tabelle di routing. Il `bind()` con `nl_pid=0` dice al kernel "questo socket appartiene a me e vuole ricevere risposte dal kernel".

nl_send() — inviare un messaggio e leggere l'ACK

```
110 static int nl_send(int fd, struct nlmsghdr *nlh) {
111     send(fd, nlh, nlh->nlmsg_len, 0);
112 
113     char buf[NL_BUFSIZE];
114     recv(fd, buf, sizeof(buf), 0);
115 
116     struct nlmsghdr *resp = (struct nlmsghdr *)buf;
117     if (resp->nlmsg_type == NLMSG_ERROR) {
118         struct nlmsgerr *err = NLMSG_DATA(resp);
119         if (err->error != 0) {
120             errno = -err->error;   /* kernel ritorna errno negato
                */
121             return -1;
122         }
123     }
124     return 0;
125 }
```

Il kernel risponde *sempre* con un messaggio di tipo `NLMSG_ERROR`. Se il campo `error` è 0, l'operazione è riuscita; altrimenti è il negativo del codice `errno` corrispondente all'errore. Notate il pattern: `send` seguito subito da `recv` — è una comunicazione sincrona richiesta/risposta, esattamente come con un server TCP.

nl_addattr() — costruire i messaggi TLV

Ogni messaggio Netlink ha una struttura ad albero: un header fisso (`nlmsghdr`) seguito da una sequenza di attributi **TLV** (Type-Length-Value) che possono essere annidati.

```
144 static struct rtattr *nl_addattr(struct nlmsghdr *nlh, int type,
145                                    const void *data, int data_len) {
146     int attr_len = RTA_LENGTH(data_len);
147     struct rtattr *rta = (struct rtattr *)
148         ((char *)nlh + NLMSG_ALIGN(nlh->nlmsg_len));
149     rta->rta_type = type;
150     rta->rta_len  = attr_len;
151     if (data && data_len > 0)
152         memcpy(RTA_DATA(rta), data, data_len);
153     nlh->nlmsg_len = NLMSG_ALIGN(nlh->nlmsg_len) + RTA_ALIGN(
        attr_len);
154     return rta;
155 }
```

La funzione scrive l'attributo immediatamente dopo l'ultimo byte del messaggio corrente, poi aggiorna `nlmsg_len`. Il punto importante è l'allineamento: il kernel richiede che ogni attributo inizi a un multiplo di 4 byte (`RTA_ALIGN`). Quando `data` è `NULL` si crea un "contenitore" vuoto — usato per gli attributi annidati come `IFLA_LINKINFO`.

veth_create() — creare la coppia virtuale

Creare una veth pair richiede un messaggio `RTM_NEWLINK` con attributi annidati su tre livelli:

```
172 static int veth_create(int nl_fd) {
173     struct nlmsghdr *nlh = ...;
174     nlh->nlmsg_type  = RTM_NEWLINK;
175     nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_CREATE | NLM_F_EXCL |
        NLM_F_ACK;
```

```
176
177     /* nome del lato host */
178     nl_addattr(nlh, IFLA_IFNAME, VETH_HOST, strlen(VETH_HOST)+1);
179
180     /* contenitore: tipo di link */
181     struct rtattr *linkinfo = nl_addattr(nlh, IFLA_LINKINFO, NULL,
            0);
182       nl_addattr(nlh, IFLA_INFO_KIND, "veth", 5);
183       struct rtattr *info_data = nl_addattr(nlh, IFLA_INFO_DATA,
             NULL, 0);
184         struct rtattr *peer_rta = nl_addattr(nlh, VETH_INFO_PEER,
              NULL, 0);
185           /* ifinfomsg del peer */
186           nl_addattr(nlh, IFLA_IFNAME, VETH_CONT, strlen(VETH_CONT)
                +1);
187         peer_rta->rta_len  = end - peer_rta;    /* chiudi il peer */
188       info_data->rta_len = end - info_data;    /* chiudi info_data
             */
189     linkinfo->rta_len  = end - linkinfo;       /* chiudi linkinfo
            */
190
191     nl_send(nl_fd, nlh);
192 }
```

La struttura ad albero è: `IFLA_LINKINFO` contiene `IFLA_INFO_KIND` (il tipo `"veth"`) e `IFLA_INFO_DATA`, che a sua volta contiene `VETH_INFO_PEER` con il nome della seconda interfaccia. Ogni livello deve essere "chiuso" aggiornando la sua `rta_len` dopo aver aggiunto i figli — altrimenti il kernel vede lunghezze errate e scarta l'intero messaggio.

Il flag `NLM_F_EXCL` fa fallire il comando se `veth0` esiste già, evitando duplicati in caso di riavvio.

Osservazione 1 — la veth pair

Avvia il container in un terminale e in un secondo terminale osserva le interfacce di rete:

```
# terminale 1
sudo ./container ./rootfs /bin/sh

# terminale 2 (sull'host, mentre il container gira)
ip link show veth0
# veth0@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> ...
#     link/ether ...
#     inet 172.17.0.1/24

# veth1 NON appare sull'host -- e' dentro il container
ip link show veth1   # -> "Device veth1 does not exist"

# dentro il container (terminale 1):
ip addr show
# 1: lo: ...  127.0.0.1/8
# 2: veth1@if3: ... 172.17.0.2/24
```

Cosa osservate: `veth0` è visibile sull'host, `veth1` è visibile solo dentro il container. Sono collegate come un tubo: un pacchetto che entra da `veth0` esce da `veth1` e viceversa.

veth_move_to_ns() — spostare veth1 nel container

```c
static int veth_move_to_ns(int nl_fd, pid_t child_pid) {
    struct nlmsghdr *nlh = ...;
    nlh->nlmsg_type  = RTM_NEWLINK;

    nl_addattr(nlh, IFLA_IFNAME, VETH_CONT, ...);   /* quale iface */

    uint32_t pid = (uint32_t)child_pid;
    nl_addattr(nlh, IFLA_NET_NS_PID, &pid, sizeof(pid)); /* dove */

    nl_send(nl_fd, nlh);
}
```

`IFLA_NET_NS_PID` è la chiave: dice al kernel "sposta questa interfaccia nel net namespace del processo con PID `child_pid`". È un'operazione atomica: non c'è una finestra in cui `veth1` non appartiene a nessun namespace.

if_set_up() — portare un'interfaccia UP

```c
static int if_set_up(const char *ifname) {
    int sock = socket(AF_INET, SOCK_DGRAM, 0);
    struct ifreq ifr;
    strncpy(ifr.ifr_name, ifname, IFNAMSIZ - 1);

    ioctl(sock, SIOCGIFFLAGS, &ifr);    /* leggi flag correnti */
    ifr.ifr_flags |= IFF_UP;            /* aggiungi IFF_UP */
    ioctl(sock, SIOCSIFFLAGS, &ifr);    /* scrivi i nuovi flag */
}
```

Qui usiamo `ioctl` invece di Netlink. Entrambi funzionano; `ioctl` è più semplice per questa operazione singola. `SIOCGIFFLAGS` legge i flag attuali dell'interfaccia, `SIOCSIFFLAGS` li riscrive con `IFF_UP` aggiunto. Senza `IFF_UP` l'interfaccia esiste ma non trasmette nulla.

if_add_addr() — assegnare un indirizzo IP

```c
static int if_add_addr(int nl_fd, const char *ifname,
                       const char *ip_str, int prefix) {
    unsigned int ifindex = if_nametoindex(ifname); /* nome -> indice */

    struct in_addr addr;
    inet_pton(AF_INET, ip_str, &addr);  /* stringa -> binario */

    struct nlmsghdr *nlh = ...;
    nlh->nlmsg_type = RTM_NEWADDR;

    struct ifaddrmsg *ifa = NLMSG_DATA(nlh);
    ifa->ifa_family    = AF_INET;
    ifa->ifa_prefixlen = prefix;   /* es. 24 per /24 */
    ifa->ifa_index     = ifindex;

    nl_addattr(nlh, IFA_LOCAL,   &addr.s_addr, 4);
    nl_addattr(nlh, IFA_ADDRESS, &addr.s_addr, 4);

    nl_send(nl_fd, nlh);
}
```

`if_nametoindex()` converte il nome testuale (`"veth0"`) nell'indice numerico che il kernel usa internamente per identificare le interfacce. `inet_pton()` converte la stringa `"172.17.0.1"` in 4 byte in formato binario big-endian. `RTM_NEWADDR` è il tipo di messaggio Netlink per assegnare indirizzi IP.

▷ Suggerimento

Quando si assegna un indirizzo con prefisso `/24`, il kernel crea **automaticamente** la route `172.17.0.0/24` verso l'interfaccia. Non serve un messaggio `RTM_NEWROUTE` separato.

Parte 3 — Cgroups v2: limitare le risorse (righe 430–475)

Namespace vs. cgroups

I **namespace** *nascondono*: il container non vede i processi e le risorse degli altri. I **cgroups** *limitano*: anche se il container cerca di usare tutta la RAM o di lanciare mille processi, il kernel lo blocca. I due meccanismi sono complementari e insieme formano la base dell'isolamento dei container.

```
436 static int setup_cgroup(pid_t child_pid) {
437     /* 1. Crea la directory del cgroup */
438     mkdir(CGROUP_BASE, 0755);
439
440     /* 2. Scrivi i limiti */
441     write_file(CGROUP_BASE "/memory.max",      "67108864"); /* 64
            MB */
442     write_file(CGROUP_BASE "/memory.swap.max", "0");        /* no
            swap */
443     write_file(CGROUP_BASE "/pids.max",        "32");
444
445     /* 3. Assegna il container al cgroup */
446     char pid_str[32];
447     snprintf(pid_str, sizeof(pid_str), "%d", child_pid);
448     write_file(CGROUP_BASE "/cgroup.procs", pid_str);
449     return 0;
450 }
```

In cgroups v2, il filesystem `/sys/fs/cgroup/` è un filesystem virtuale: creare una directory equivale a creare un nuovo gruppo di controllo. Scrivere un valore in `memory.max` imposta il limite; scrivere il PID in `cgroup.procs` aggiunge il processo al gruppo. Da quel momento in poi il kernel applica i limiti a quel processo e a tutti i suoi discendenti.

Osservazione 2 — i cgroups

```
# mentre il container gira (terminale 2):
ls /sys/fs/cgroup/container_demo/
# cgroup.procs  memory.max  memory.swap.max  pids.max  ...

cat /sys/fs/cgroup/container_demo/memory.max
# 67108864

cat /sys/fs/cgroup/container_demo/pids.max
# 32
```

```
# PID del container sull'host:
cat /sys/fs/cgroup/container_demo/cgroup.procs
# 7842  (o qualsiasi sia il PID attuale)

# memoria usata in tempo reale:
cat /sys/fs/cgroup/container_demo/memory.current
```

Cosa osservate: la directory esiste solo mentre il container è in esecuzione. `cleanup_cgroup()` la rimuove con `rmdir()` alla fine. I file dentro non sono file normali — sono interfacce verso il kernel.

Parte 4 — Isolamento del filesystem (righe 484–612)

setup_mounts() — montare proc, sys, dev

Prima di cambiare il root, montiamo i filesystem virtuali necessari dentro il rootfs Alpine.

```
540 static int setup_mounts(const char *rootfs) {
541     char path[256];
542
543     /* /proc: necessario per ps, top, /proc/self, ecc. */
544     snprintf(path, sizeof(path), "%s/proc", rootfs);
545     mkdir(path, 0555);
546     mount("proc", path, "proc",
547           MS_NOSUID | MS_NOEXEC | MS_NODEV, NULL);
548
549     /* /sys: read-only */
550     snprintf(path, sizeof(path), "%s/sys", rootfs);
551     mount("sysfs", path, "sysfs",
552           MS_NOSUID | MS_NOEXEC | MS_NODEV | MS_RDONLY, NULL);
553
554     /* /dev: tmpfs + device node mknod */
555     snprintf(path, sizeof(path), "%s/dev", rootfs);
556     mount("tmpfs", path, "tmpfs", MS_NOSUID | MS_STRICTATIME,
557           "mode=755,size=65536k");
558     mknod(".../dev/null",   S_IFCHR | 0666, makedev(1, 3));
559     mknod(".../dev/zero",   S_IFCHR | 0666, makedev(1, 5));
560     mknod(".../dev/random", S_IFCHR | 0666, makedev(1, 8));
561 }
```

I flag `MS_NOSUID`, `MS_NOEXEC`, `MS_NODEV` sono misure di sicurezza: impediscono rispettivamente di eseguire file setuid, di eseguire binari, e di creare device node su quei mount.

`mknod()` con `makedev(major, minor)` crea i device node character: `(1,3)` è `/dev/null`, `(1,5)` è `/dev/zero`, ecc. Questi numeri sono convenzioni UNIX stabili da decenni.

pivot_root_to() — cambiare il root in modo sicuro

```
484 static int pivot_root_to(const char *new_root) {
485     char old_root[256];
486     snprintf(old_root, sizeof(old_root), "%s/.pivot_old", new_root)
        ;
487
488     /* Il new_root deve essere un mount point: bind mount su se
        stesso */
489     mount(new_root, new_root, NULL, MS_BIND | MS_REC, NULL);
```

```
490
491     mkdir(old_root, 0700);
492
493     /* pivot_root non ha wrapper glibc: syscall diretta */
494     syscall(SYS_pivot_root, new_root, old_root);
495
496     chdir("/");    /* siamo ora nel nuovo root */
497
498     /* smonta e rimuove il vecchio root */
499     umount2("/.pivot_old", MNT_DETACH);
500     rmdir("/.pivot_old");
501     return 0;
502 }
```

[!] chroot() vs. pivot_root()

`chroot()` è facilmente aggirabile: un processo con i privilegi giusti può fare `chroot("..")` ripetutamente e uscire dalla gabbia. `pivot_root()` è un'operazione atomica del kernel che *rimpiazza* il root mount — il vecchio root viene smontato e non è più raggiungibile. Docker usa `pivot_root()` per questa ragione.

Il bind mount (`MS_BIND`) su se stesso è obbligatorio perché `pivot_root()` richiede che `new_root` sia già un mount point. Senza questo passaggio la syscall fallisce con `EINVAL`.

Osservazione 3 — il filesystem isolato

```
# dentro il container:
ls /
# bin   dev   etc   home   lib   media   mnt   opt   proc
# root  run   sbin  srv    sys   tmp     usr   var
# --> e' il rootfs Alpine, non il filesystem dell'host

cat /etc/os-release
# NAME="Alpine Linux"

# il filesystem dell'host non e' accessibile:
ls /home    # solo la home di Alpine

# ps mostra solo i processi del container:
ps aux
# PID   USER     COMMAND
#   1   root     /bin/sh
#   5   root     ps aux
# PID 1 = la nostra shell, non systemd!
```

Parte 5 — clone() e i namespace (righe 614–740)

child_fn() — il codice eseguito dentro il container

Quando `clone()` crea il processo figlio, la prima funzione che esegue è `child_fn()`. Tutto ciò che c'è dentro opera già nei nuovi namespace.

```
617 static int child_fn(void *arg) {
618     child_args_t *args = (child_args_t *)arg;
619
```

```
620     prctl(PR_SET_PDEATHSIG, SIGKILL); /* muori se il parent muore
            */
621
622     sethostname(HOSTNAME, strlen(HOSTNAME)); /* UTS namespace */
623
624     mount(NULL, "/", NULL, MS_REC | MS_PRIVATE, NULL); /* mount
            privato */
625
626     setup_mounts(args->rootfs);   /* monta proc/sys/dev */
627     pivot_root_to(args->rootfs);  /* cambia il root */
628
629     mount("proc", "/proc", "proc", MS_NOSUID|MS_NOEXEC|MS_NODEV,
            NULL);
630
631     setup_container_net();   /* alza veth1, assegna IP */
632
633     execvp(args->argv[0], args->argv); /* lancia il comando */
634     return 1; /* mai raggiunto se execvp ha successo */
635 }
```

`prctl(PR_SET_PDEATHSIG, SIGKILL)` garantisce che il child riceva `SIGKILL` se il parent termina inaspettatamente — evita container "orfani" che continuano a girare dopo che il processo gestore è morto.

`mount(NULL, "/", NULL, MS_REC | MS_PRIVATE, NULL)` rende privato il mount namespace: qualsiasi mount fatto d'ora in poi non si propagherà all'host. Questo è necessario perché il namespace è stato appena creato e per default eredita il comportamento del parent (`MS_SHARED`).

La sequenza è vincolata: non si può fare `pivot_root` prima di avere un mount point valido, e non si può montare `/proc` nel rootfs corretto prima di `pivot_root`.

main() — clone() con i flag dei namespace

```
728 int flags = CLONE_NEWPID   /* nuovo albero PID          */
729           | CLONE_NEWUTS   /* hostname indipendente     */
730           | CLONE_NEWNS    /* mount namespace isolato   */
731           | CLONE_NEWNET   /* stack di rete vuoto       */
732           | CLONE_NEWIPC   /* IPC/shm separati          */
733           | SIGCHLD;       /* notifica al parent        */
734
735 pid_t child_pid = clone(child_fn, stack_top, flags, &args);
```

Ogni flag `CLONE_NEW*` dice al kernel di creare un namespace nuovo e distinto per il figlio. Possono essere combinati liberamente con l'`OR` bit a bit. `SIGCHLD` in fondo non è un flag di namespace: specifica il segnale che il kernel deve inviare al parent quando il figlio termina (necessario per `waitpid()`).

Osservazione 4 — namespace e PID

```
# dentro il container:
echo $$          # -> 1  (siamo PID 1 nel namespace)
cat /proc/1/cmdline | tr '\0' ' '  # -> /bin/sh

# sull'host (terminale 2):
# il container ha un PID diverso
pgrep -a container   # es. 9341 ./container ./rootfs /bin/sh
```

```
cat /proc/9341/ns/pid
# pid:[4026532193]  <-- namespace diverso dall'initial

cat /proc/1/ns/pid
# pid:[4026531836]  <-- initial namespace dell'host

# i due numeri sono diversi: ambienti isolati
```

Cosa osservate: lo stesso processo ha PID 1 dentro il container e un PID alto sull'host. I file in `/proc/<pid>/ns/` mostrano l'inode del namespace: numeri diversi = namespace diversi.

Osservazione 5 — hostname isolato

```
# dentro il container:
hostname          # -> container

# sull'host (terminale 2):
hostname          # -> il nome della tua macchina (invariato)

# modifica l'hostname dentro il container:
hostname nuovonome
hostname          # -> nuovonome

# sull'host (non cambia nulla):
hostname          # -> invariato
```

Cosa osservate: il namespace UTS isola completamente hostname e domainname. Le modifiche dentro il container non toccano l'host.

Parte 6 — Orchestrazione in main() (righe 683–784)

```
683 int main(int argc, char *argv[]) {
684     /* ... validazione argomenti ... */
685
686     child_args_t args = { .rootfs = argv[1], .argv = &argv[2] };
687
688     char *stack     = malloc(STACK_SIZE);
689     char *stack_top = stack + STACK_SIZE; /* x86-64: stack cresce
            in giu' */
690
691     pid_t child_pid = clone(child_fn, stack_top, flags, &args);
692
693     setup_cgroup(child_pid);    /* limiti di risorse */
694     setup_veth(child_pid);      /* rete: crea veth, sposta veth1 nel
            ns */
695
696     waitpid(child_pid, &status, 0);    /* attendi la fine */
697
698     cleanup_veth();
699     cleanup_cgroup();
700     free(stack);
701 }
```

L'ordine delle operazioni nel parent è fondamentale. `clone()` deve venire prima di `setup_veth()` perché la funzione `veth_move_to_ns()` ha bisogno del PID del child, che non esiste prima

della `clone()`. Allo stesso tempo, `setup_veth()` deve completare prima che il child raggiunga `setup_container_net()`, perché quest'ultima si aspetta che `veth1` sia già nel net namespace del container.

Queste due operazioni — parent che configura e child che si inizializza — avvengono in parallelo. Il fatto che normalmente funzionino è dovuto ai tempi di esecuzione: il child ha molto lavoro da fare (mount, pivot_root) prima di chiamare `setup_container_net()`, e il parent completa `setup_veth()` in pochi millisecondi.

> ▷ **Suggerimento**
>
> In una implementazione robusta, si userebbe una **pipe** o un **semaforo** per sincronizzare esplicitamente parent e child. Docker usa una pipe con un byte di sincronizzazione: il child attende su `read(pipe)` fino a quando il parent ha finito la configurazione e scrive un byte, poi il child prosegue. Questo elimina la race condition.

La pipe di sincronizzazione — eliminare la race condition

Senza sincronizzazione esplicita esiste una **race condition**: se il child è abbastanza veloce da arrivare a `setup_container_net()` prima che il parent abbia eseguito `veth_move_to_ns()`, troverà `veth1` inesistente e fallirà. Lo schema con la pipe risolve il problema in modo deterministico.

```
/* Nel main(), prima di clone(): */
int sync_pipe[2];
pipe(sync_pipe);

child_args_t args = {
    .rootfs    = rootfs,
    .argv      = cmd,
    .sync_pipe = { sync_pipe[0], sync_pipe[1] },
};
```

La pipe viene creata *prima* di `clone()`: entrambi i capi sono quindi ereditati dal child. Il write end `sync_pipe[1]` appartiene al parent, il read end `sync_pipe[0]` al child.

```
/* Nel main(), dopo clone(): */
close(sync_pipe[0]);        /* parent non legge mai */

setup_cgroup(child_pid);
setup_veth(child_pid);      /* crea veth, sposta veth1 nel ns */

/* Sblocca il child: veth1 e' pronta */
write(sync_pipe[1], "\x01", 1);
close(sync_pipe[1]);
```

```
/* In child_fn(), dopo pivot_root: */
char byte;
read(args->sync_pipe[0], &byte, 1);  /* attende il parent */
close(args->sync_pipe[0]);

setup_container_net();   /* ora veth1 c'e' con certezza */
```

Il child si blocca su `read()` — una chiamata bloccante — finché il parent non scrive il byte `0x01`. La sequenza garantita diventa:

Parent	Child
`clone()`	`child_fn()` parte
`close(sync_pipe[0])`	... pivot_root ...
`setup_cgroup()`	... setup_mounts ...
`setup_veth()` ⟵ veth pair creata	`read(pipe)` *(bloccato)*
`veth_move_to_ns()` ⟵ veth1 nel ns	*(ancora bloccato)*
`write(pipe, 1)`	`read()` si sblocca
`close(sync_pipe[1])`	`setup_container_net()`
`waitpid()`	`execvp("/bin/sh")`

Perché chiudere i lati inutilizzati?

Se il parent tenesse aperto `sync_pipe[0]`, la pipe non sarebbe mai in stato "rotta". Se il parent morisse prima di scrivere il byte, il child resterebbe bloccato su `read()` per sempre, senza ricevere `EOF`. Chiudendo subito i lati inutilizzati si garantisce che un crash del parent sblocchi immediatamente il child con un errore gestibile.

Esercizi di verifica

Esercizio 1 — connettività

1. Avvia il container in un terminale.
2. Da dentro il container, esegui `ping 172.17.0.1`.
   Cosa succede? Perché funziona?
3. Da dentro il container, esegui `ping 8.8.8.8`.
   Cosa succede? Perché non funziona (ancora)?
4. Sull'host, esegui questi due comandi e riprova il ping:

```
echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward
sudo iptables -t nat -A POSTROUTING \
     -s 172.17.0.0/24 -o eth0 -j MASQUERADE
```

   Perché ora funziona? Cosa fa ciascuno dei due comandi?

Esercizio 2 — limiti cgroup

1. Dentro il container, installa uno stress test:

```
apk add --no-cache stress-ng
stress-ng --vm 1 --vm-bytes 100M --timeout 30s
```

2. Sull'host, monitora il cgroup:

```
watch -n 0.5 cat /sys/fs/cgroup/container_demo/memory.current
```

   Il valore supera mai 67108864 (64 MB)? Cosa fa il kernel quando il processo cerca di allocare più memoria?
3. Prova la fork bomb dentro il container:

```
:(){ :|:& };:
```

   Cosa succede? Perché il container non crasha l'host?

Esercizio 3 — ispezione namespace

1. Trova il PID host del container con `pgrep container`.
2. Elenca i namespace del container:

```
ls -la /proc/<pid>/ns/
```

3. Confronta ogni inode con quelli del processo `init` (PID 1):

```
ls -la /proc/1/ns/
```

Quali namespace sono diversi? Quali sono uguali e perché? (Suggerimento: quale namespace *non* abbiamo creato con `CLONE_NEW*`?)

Esercizio 4 — due container

1. Avvia un secondo container in un terzo terminale:

```
sudo ./container ./rootfs /bin/sh
```

2. Cosa succede? Perché fallisce? (Leggi il messaggio di errore.)
3. Il problema è in `veth_create()`: il flag `NLM_F_EXCL` impedisce di creare `veth0` se esiste già. Come modifichereste il codice per supportare più container contemporaneamente?
   Non serve implementarlo — basta descrivere l'approccio (es. nomi dinamici, parametro aggiuntivo, ecc.).

Domande di riflessione

Q1. `clone()` crea un nuovo processo; `fork()` anche. Qual è la differenza concreta in termini di namespace? Cosa succederebbe se usassimo `fork()` al posto di `clone()` con i flag?

Q2. Perché `pivot_root()` è più sicuro di `chroot()`? Descrivete uno scenario di attacco che funziona con `chroot()` ma non con `pivot_root()`.

Q3. Il child esegue `setup_container_net()` dopo `pivot_root()`. A questo punto siamo nel rootfs Alpine. Eppure `nl_open()` funziona e vede `veth1`. Perché? Cosa isola il namespace di rete e cosa *non* isola?

Q4. Se rimuovessimo `CLONE_NEWPID` dai flag di `clone()`, cosa cambierebbe dall'interno del container? Il container sarebbe ancora funzionale?

Q5. Il codice non usa `CLONE_NEWUSER`. Cosa significherebbe aggiungerlo? Qual è il vantaggio principale (suggerimento: pensate a `sudo`)?