Linux
Security and Isolation APIs
Essentials

Michael Kerrisk
man7.org

January 2025

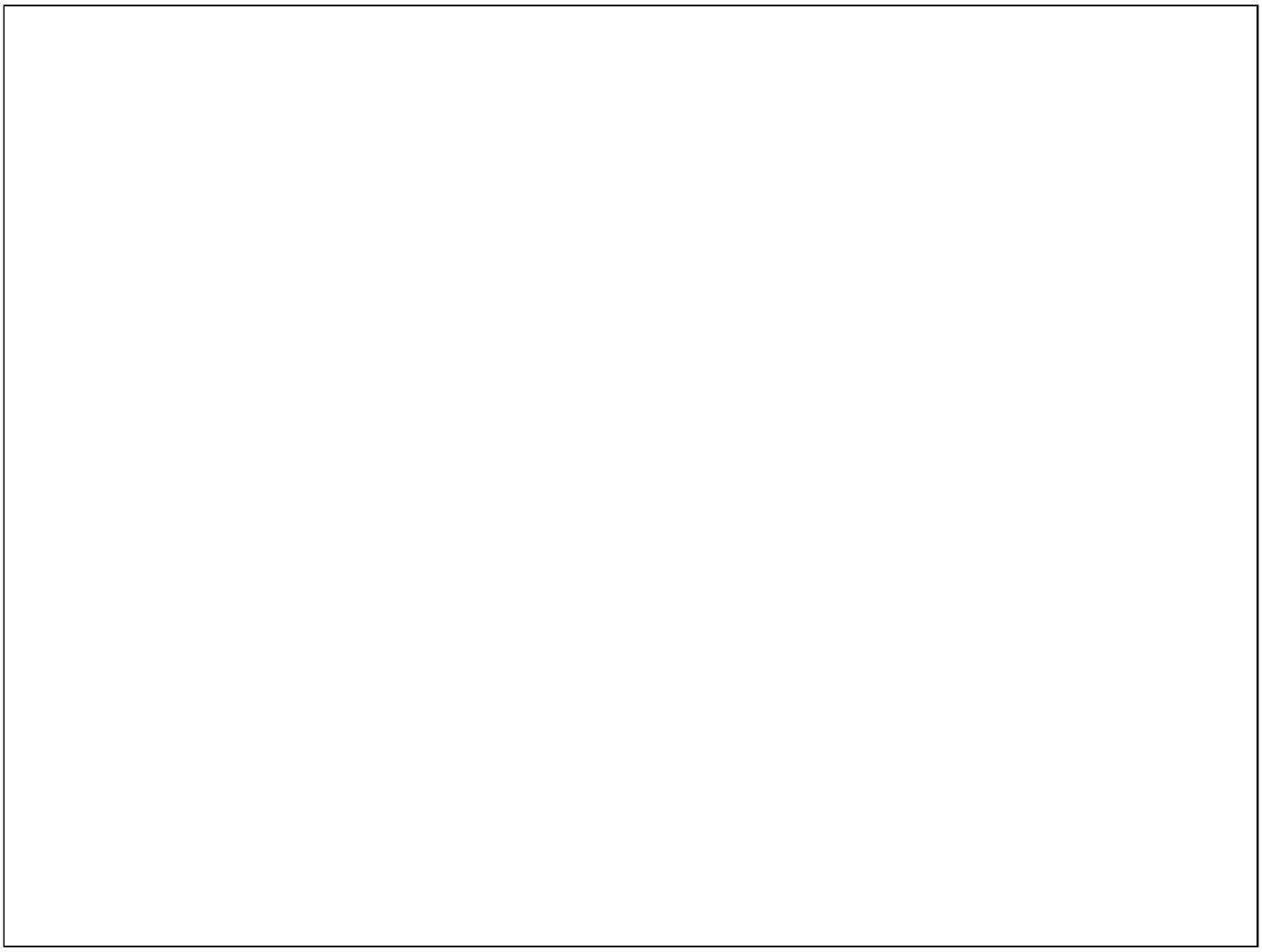
For information about this course, visit http://man7.org/training/.

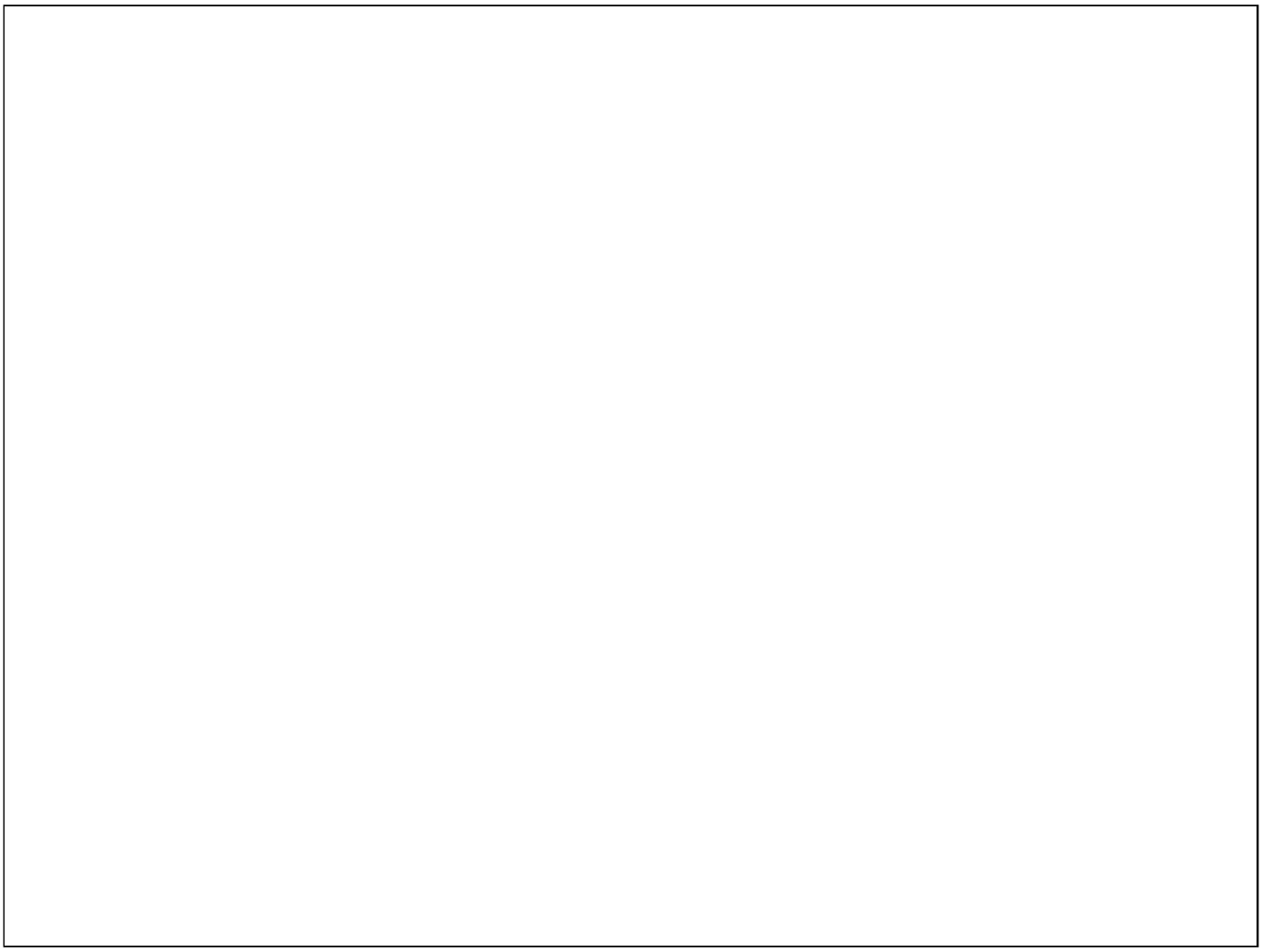
For inquiries regarding training courses, please contact us at training@man7.org.

Please send corrections and suggestions for improvements to this course material to training@man7.org.

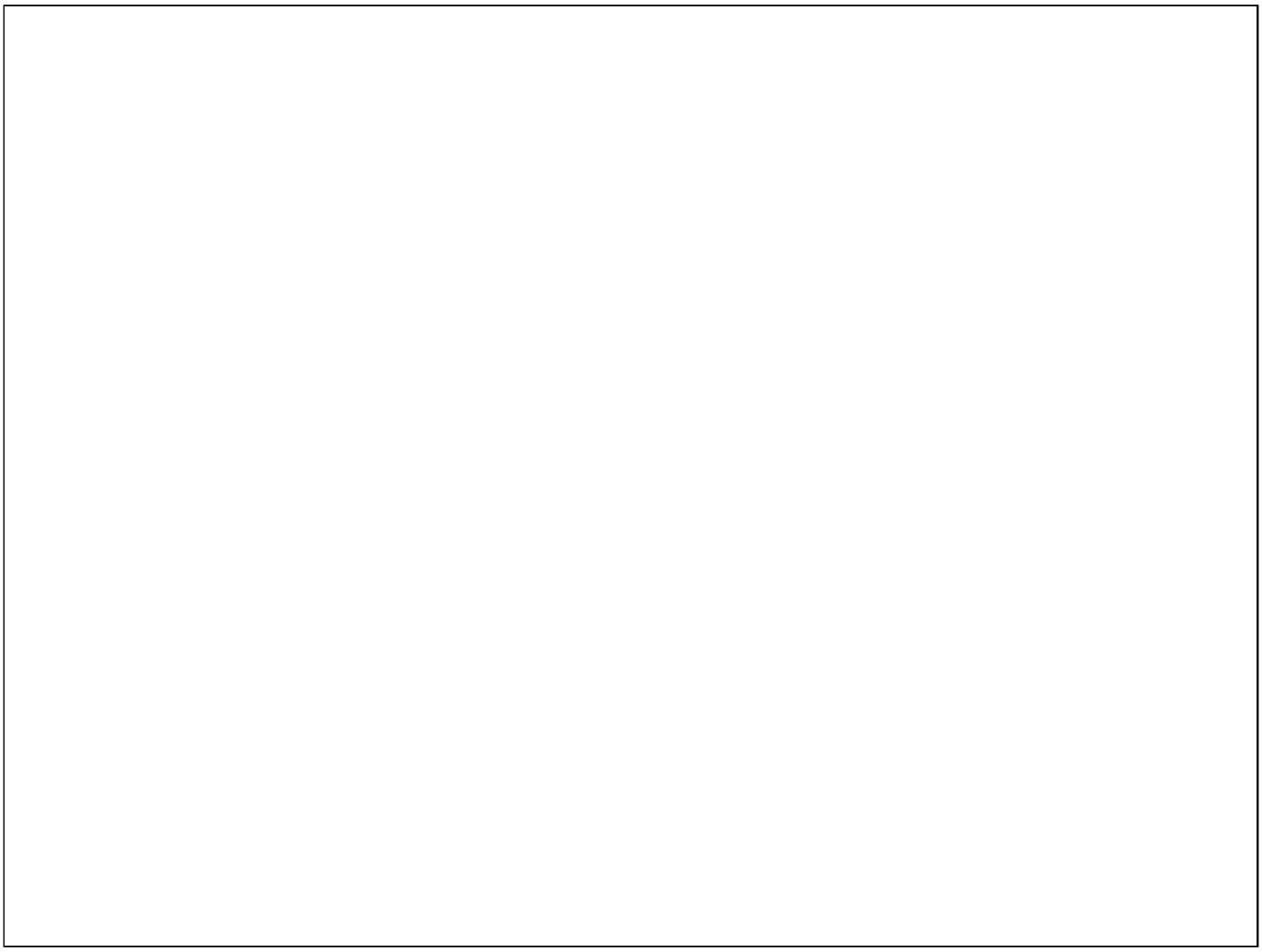
For information about *The Linux Programming Interface*, please visit http://man7.org/tlpi/.

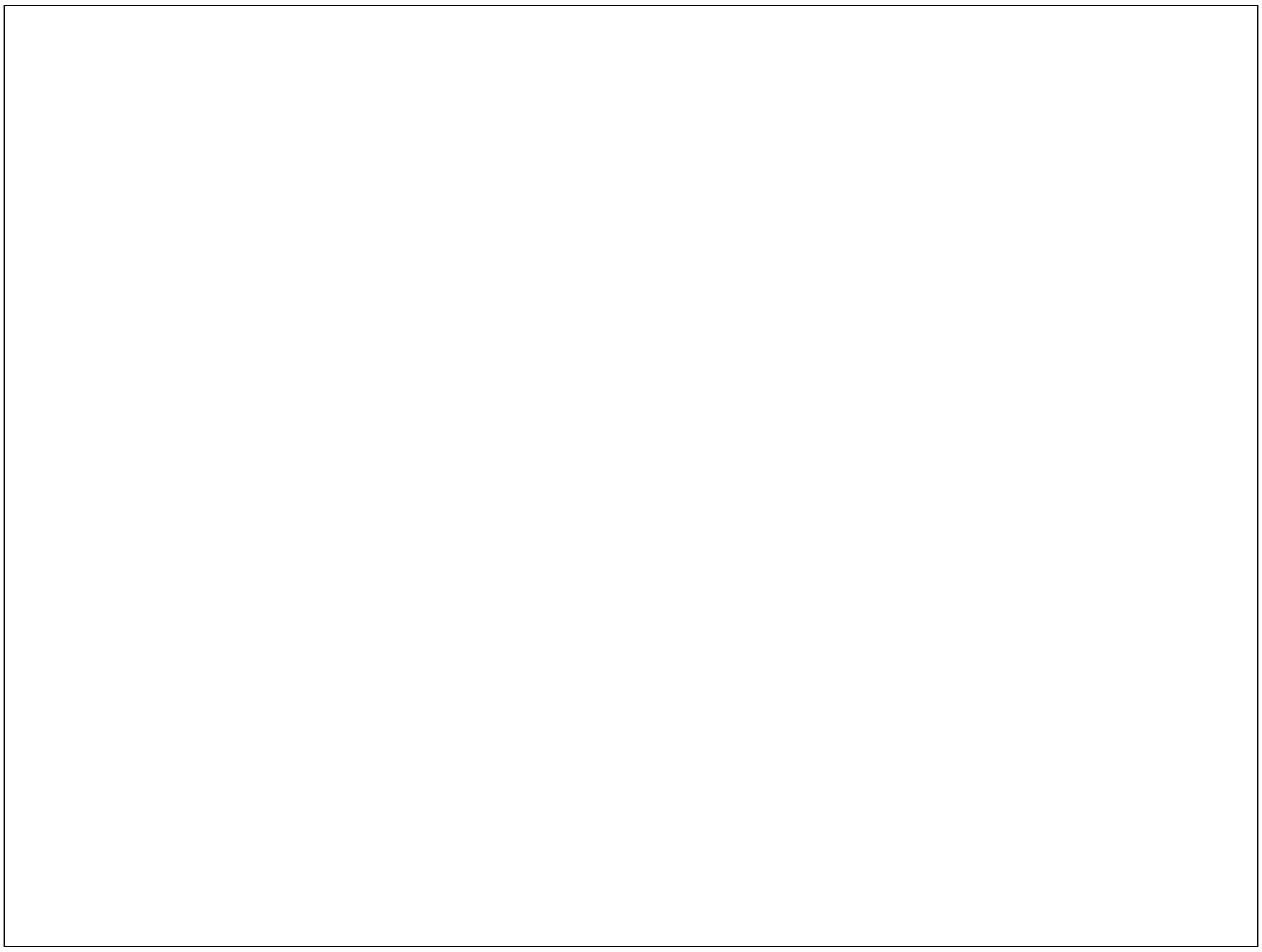
Revision: # 6f75b3d2e02f





Short table of contents





Detailed table of contents

Detailed table of contents

Detailed table of contents

Linux Security and Isolation APIs Essentials

Course Introduction

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Outline

Course prerequisites

- Prerequisites
  - (Good) reading knowledge of C
  - Can log in to Linux / UNIX and use basic commands
- Knowledge of *make(1)* is helpful
  - (Can do a short tutorial during first practical session for those new to *make*)

Course goals

- Understanding kernel mechanisms related to security and isolation:
  - Set-UID and set-GID programs
  - Capabilities
  - Namespaces
  - Cgroups (control groups)

Lab sessions

- Lots of lab sessions...
- **Pair/group work is strongly encouraged!**
  - Usually gets us through practical sessions faster
    - ⇒ so we can cover more topics
- **Read each exercise thoroughly** before starting
  - I've seen the traps that people often fall into
  - ⇒ exercise descriptions often include **important hints**
- Lab sessions are **not** instructor down time...
  - ⇒ One-on-one questions about course material or exercises

Outline

System/software requirements: kernel

- Kernel configuration; following should be "y" or "m"

```
CONFIG_AUDIT
CONFIG_CGROUPS
CONFIG_CGROUP_PIDS
CONFIG_CGROUP_FREEZER
CONFIG_CGROUP_SCHED
CONFIG_MEMCG
CONFIG_USER_NS
CONFIG_SECCOMP
CONFIG_SECCOMP_FILTER
CONFIG_CFS_BANDWIDTH
CONFIG_VETH
```

- To see what options were used to build the running kernel:

```
$ cat /proc/config.gz       # (if it is present)
$ cat /lib/modules/$(uname -r)/build/.config
```

  - On Debian derivatives:

```
$ cat /boot/config-$(uname -r)
```

System/software requirements: packages to install

- *gcc* (or your preferred C compiler)
- *make*
- *libseccomp-dev[el]*
- *libcap-dev[el]*
- *libacl1-dev* / *libacl-devel*
- *libcrypt-dev* / *libxcrypt-devel*
- *util-linux*
- *libcap-ng-utils*
- *libreadline-dev* / *readline-devel*
- *sudo* (and ensure that your login has *sudo* access)
  - See *sudo(8)*, *visudo(8)*; you will need to be in the *wheel* (or possibly, *sudo*) group
- *inotify-tools*
- *golang* (useful for a few code examples)

Outline

Course materials

- Slides / course book
- Source code tarball
  - Location sent by email
  - Unpacked source code is a Git repository; you can commit/revert changes, etc

Other resources

- Manual pages
  - Section 2: system calls
  - Section 3: library functions
  - Section 7: overviews
  - Latest version online at `http://man7.org/linux/man-pages/`
  - Latest tarball downloadable at `https://mirrors.edge.kernel.org/pub/linux/docs/man-pages/`

Outline

Common abbreviations used in slides

The following abbreviations are sometimes used in the slides:

- CWD: current working directory
- EA: extended attribute
- FD: file descriptor
- FS: filesystem
- FTM: feature test macro
- GID: group ID
  - rGID, eGID, sGID (real, effective, saved set-)
- IPC: interprocess communication
- NS: namespace
- PID: process ID
- PPID: parent process ID
- UID: user ID
  - rUID, eUID, sUID (real, effective, saved set-)

Outline

Introductions: me

- Programmer, trainer, writer
- UNIX since 1987, Linux since mid-1990s
- Active contributor to Linux
  - API review, testing, and documentation
    - API design and design review
    - Lots of testing, lots of bug reports, a few kernel patches
  - Maintainer of Linux *man-pages* project (2004-2021)
    - Documents kernel-user-space + C library APIs
    - Contributor since 2000
    - As maintainer: ≈23k commits, 196 releases
    - Author/coauthor of ≈440 manual pages
- Kiwi in .de
  - (mtk@man7.org, PGP: 4096R/3A35CE5E)
  - @mkerrisk (feel free to tweet about the course as we go...)
  - `http://linkedin.com/in/mkerrisk`

Introductions: you

In brief:

- Who are you?
  - If virtual: where are you?
- Two interesting things about you / things you like to do when you are not in front of a keyboard

Questions policy

- General policy: ask questions any time, in one of the following ways:
  - On **Slack**
  - If online, click the **"Raise hand" button**
    - I'll usually see it, **and I get to see your name as well**
  - Or out loud
    - But, wait for a quiet point
    - And if online, please announce your name, since I might not be able to see you
- In the event that questions slow us down too much, I may say: "batch your questions until next *Question penguin* slide"

Notes

Notes

Linux Security and Isolation APIs Essentials

Classical Privileged Programs

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Rev: # 6f75b3d2e02f

Outline

Process credentials (real and effective)

- Processes have credentials (user and group IDs), including:
  - **Real user ID** (rUID) and **real group ID** (rGID)
    - Tell us who process belongs to
    - Login shell gets these IDs from `/etc/passwd`
    - Can be retrieved using *getuid()* and *getgid()*
  - **Effective user ID** (eUID) and **effective group ID** (eGID)
    - Used (along with supplementary GIDs) for permission checking (e.g., file access)
    - Can be retrieved using *geteuid()* and *getegid()*
- Credentials are inherited by child of *fork()*
- For many processes, effective credentials are same as corresponding real credentials

Set-user-ID and set-group-ID programs

- Set-user-ID (set-group-ID) program is classical UNIX privilege-granting mechanism:
  - Gives process privileges of different user (group)
  - Achieved by changing process effective UID (GID)

Set-UID example (privprogs/simple_setuid.c)

```
int main(int argc, char *argv[]) {
    printf("rUID = %ld, eUID = %ld\n",
            (long) getuid(), (long) geteuid());

    if (argc > 1) {
        int fd = open(argv[1], O_RDONLY);
        if (fd >= 0)
            printf("Successfully opened %s\n", argv[1]);
        else
            perror("Open failed");
    }

    exit(EXIT_SUCCESS);
}
```

- Print process real and effective UID
- If argument was supplied, try to open that file

Set-UID example (privprogs/simple_setuid.c)

- Run program as unprivileged user, attempting to open /etc/shadow:

```
$ id
uid=1000(mtk) gid=1000(mtk) ...
$ ./simple_setuid /etc/shadow
rUID = 1000, eUID = 1000
Open failed: Permission denied
```

- Real and effective UID have same value
  - Unprivileged UID 1000
- *open()* fails; unprivileged user can't open /etc/shadow

```
$ ls -l /etc/shadow
----------. 1 root root 1450 Jan  3 14:17 /etc/shadow
```

  - On other systems, permissions may differ, but on every system, /etc/shadow is not publicly readable

Creating a set-UID program

- When executed, a set-UID (set-GID) program changes eUID (eGID) of process to be same as UID (GID) of executable
  - Technique used by several common system programs: *passwd(1)*, *mount(8)*, *su(1)*
- To create set-UID (set-GID) program:
  - Ensure executable is owned by desired UID (GID)
  - Turn on set-UID (set-GID) mode bit of executable
    - `chmod u+s` (`chmod g+s`)

Set-UID example (privprogs/simple_setuid.c)

- Let's make our program set-UID-*root*:

```
$ sudo chown root simple_setuid
$ sudo chmod u+s simple_setuid
```

- *ls* shows that this is a set-UID program:

```
$ ls -l simple_setuid
-rwsr-xr-x. 1 root mtk 27592 Jan 11 20:46 simple_setuid
```

  - "s" in user-execute permission == program is set-UID
- Again run program, attempting to open /etc/shadow:

```
$ ./simple_setuid /etc/shadow
rUID = 1000, eUID = 0
Successfully opened /etc/shadow
```

  - Process eUID was changed to be same as UID of executable
  - File was successfully opened

Privilege

- A set-UID (set-GID) program gives process the "privileges" of a different user (group)
- Could be privileges of another "normal" user (or group)
  - So, e.g., can access files owned by that user (or group)
- A set-UID-*root* program gives process privileges of *root*
  - Powerful
  - And dangerous!
    - Many pitfalls (especially in C)
    - See TLPI Ch. 38; Bishop, M. (2003) *Computer Security: Art and Science*; and other sources listed in TLPI §38.12

Outline

Saved set-user-ID and saved set-group-ID

- Each process has two more credentials:
  saved set-user-ID (sUID) and saved set-group-ID (sGID)
  - Designed for use with set-UID/set-GID programs
  - Can be retrieved using:
    `getresuid(&ruid, &euid, &suid)`
    `getresgid(&rgid, &egid, &sgid)`
    - APIs return real, effective, and saved set IDs

Saved set-user-ID and saved set-group-ID

- Kernel does the following when execing a program (*execve()*):
  1. Set-UID bit enabled on executable?
     ⇒ process effective UID is made same as file UID
  2. Set-GID bit enabled on executable?
     ⇒ process effective GID is made same as file GID
  3. Effective IDs are copied to corresponding saved set IDs
     - (Done regardless of whether set-UID or set-GID bit is set)
- IOW: saved set IDs record state of effective IDs at program start up

Saved set-user-ID and saved set-group-ID

- When set-UID program is executed, credentials look like this:

Real UID (unchanged by *exec()*)	**Effective UID** (copied from file owner)	**Saved set-user-ID** (copied from eff. UID at program start-up)
Unprivileged ID	Privileged ID	Privileged ID

- A process can switch its effective UID back and forth between real UID and saved set-user-ID
  - i.e., between unprivileged and privileged states
- Analogously for set-GID programs and saved set-group-ID
- What is the design mistake in initial set-up of process UIDs in above picture?
  - In other words: what is the first thing that a set-UID / set-GID program should do on start-up?
    - (Reset effective UID to same value as real UID)

Outline

Changing process credentials

General principle for all APIs that change credentials:

- **Privileged processes** can make any changes to IDs
  - Privileged process ≈ process effective user ID 0
    - More precisely: process has appropriate Linux capability (`CAP_SETUID` for UID changes, `CAP_SETGID` for GID changes)
- **Unprivileged processes** can change an ID to same value as another of its current IDs
  - e.g., unprivileged *seteuid()* can change effective UID to same value as real or saved set UID

[TLPI §9.7]

Changing process credentials

- `setresuid(ruid, euid, suid)`: change real, effective, and saved set **UIDs**
  - –1 means "no change" in corresponding UID
- `setresgid(rgid, egid, sgid)`: change real, effective, and saved set **GIDs**
  - –1 means "no change" in corresponding GID

Outline

Operate with least privilege

- Generally best to hold (elevated) privilege only when required
  - "Principle of least privilege"
  - If program is compromised while in lower privilege state, this makes attacker's life harder
- Lower privilege when not needed, and raise temporarily as required
  - i.e., switch effective ID back and forth between real and saved set ID
- If privilege will never again be needed, drop it permanently
  - i.e., set effective and saved set IDs to same value as real ID

Dropping and raising privileges

- Drop and raise privileges:

```
euid = geteuid();               /* Save eUID */
setresuid(-1, getuid(), -1);    /* Drop */

setresuid(-1, euid, -1);        /* Raise */
/* Do privileged work */
setresuid(-1, getuid(), -1);    /* Drop */
```

- Irrevocably drop privileges:

```
setresuid(-1, getuid(), getuid());
```

Linux Security and Isolation APIs Essentials

Capabilities

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Rev: # 6f75b3d2e02f

Outline

Rationale for capabilities

- Traditional UNIX privilege model divides users into two groups:
  - Normal users, subject to privilege checking based on UID and GIDs
  - Effective UID 0 (superuser) bypasses many of those checks
- Coarse granularity is a problem:
  - E.g., to give a process power to change system time, we must also give it power to bypass file permission checks
    - ⇒ No limit on possible damage if program is compromised

[TLPI §39.1]

Rationale for capabilities

- Capabilities divide power of superuser into small pieces
  - 41 capabilities, as at Linux 6.13
  - Traditional superuser == process that has full set of capabilities
- Goal: replace set-UID-*root* programs with programs that have capabilities
  - Compromise in set-UID-*root* binary ⇒ very dangerous
  - Compromise in binary with file capabilities ⇒ less dangerous

A selection of Linux capabilities

Capability	**Permits process to**
CAP_CHOWN	Make arbitrary changes to file UIDs and GIDs
CAP_DAC_OVERRIDE	Bypass file RWX permission checks
CAP_DAC_READ_SEARCH	Bypass file R and directory X permission checks
CAP_IPC_LOCK	Lock memory
CAP_FOWNER	*chmod()*, *utime()*, set ACLs on arbitrary files
CAP_KILL	Send signals to arbitrary processes
CAP_NET_ADMIN	Various network-related operations
CAP_SETFCAP	Set file capabilities
CAP_SETGID	Make arbitrary changes to process's (own) GIDs
CAP_SETPCAP	Make changes to process's (own) capabilities
CAP_SETUID	Make arbitrary changes to process's (own) UIDs
CAP_SYS_ADMIN	Perform a wide range of system admin tasks
CAP_SYS_BOOT	Reboot the system
CAP_SYS_NICE	Change process priority and scheduling policy
CAP_SYS_MODULE	Load and unload kernel modules
CAP_SYS_RESOURCE	Raise process resource limits, override some limits
CAP_SYS_TIME	Modify the system clock

More details: *capabilities(7)* manual page and TLPI §39.2

Outline

Process and file capabilities

- Processes and (binary) files can each have capabilities
- **Process capabilities** define power of process to do privileged operations
  - Traditional superuser == process that has **all** capabilities
- **File capabilities** are a mechanism to give a process capabilities when it execs the file
  - Stored in `security.capability` extended attribute
    - (File metadata; `getfattr -m - <file>`)

[TLPI §39.3]

Process and file capability sets

- Capability set: bit mask representing a group of capabilities
- Each **process**[†] has 3[‡] capability sets:
  - Permitted
  - Effective
  - Inheritable

[†] In truth, capabilities are a per-thread attribute
[‡] In truth, there are more capability sets

- An **executable file** may have 3 associated capability sets:
  - Permitted
  - Effective
  - Inheritable

Inheritable and ambient capabilities

- As a simplification, **we will largely ignore certain capability sets**:
  - Process and file **inheritable** sets
    - A feature misdesign that turned out not to be useful
    - Commonly, these sets are empty (i.e., all zeroes)
  - Process **ambient** set
    - Designed for a particular (less common) use case
    - (Serves a use case that couldn't be solved by inheritable set)

Viewing process capabilities

- `/proc/PID/status` fields (hexadecimal bit masks):

```
$ cat /proc/4091/status
...
CapInh: 0000000000000000
CapPrm: 0000000000200020
CapEff: 0000000000000000
```

  - See `<sys/capability.h>` for capability bit numbers
    - Here: `CAP_KILL` (bit 5), `CAP_SYS_ADMIN` (bit 21)
- *getpcaps(1)* (part of *libcap* package):

```
$ getpcaps 4091
Capabilities for `4091': = cap_kill,cap_sys_admin+p
```

  - More readable notation, but a little tricky to interpret
  - Here, single '=' means all sets are empty
- *capsh(1)* can be used to decode hex masks:

```
$ capsh --decode=200020
0x0000000000200020=cap_kill,cap_sys_admin
```


Modifying process capabilities

- A process can modify its capability sets by:
  - **Raising** a capability (adding it to set)
    - Synonyms: add, enable
  - **Lowering** a capability (removing it from set)
    - Synonyms: **drop**, **clear**, remove, disable
  - (APIs for changing process capabilities are *capset(2)*, *prctl(2)*, and *libcap* library; we won't look at these)
- There are various rules about changes a process can make to its capability sets

Outline

Process permitted and effective capabilities

- *Permitted* : capabilities that process *may* employ
  - "Upper bound" on effective capability set
  - Once dropped from permitted set, a capability can't be reacquired
    - (But see discussion of *execve()* later)
  - Can't drop while capability is also in effective set
- *Effective* : capabilities that are currently in effect for process
  - I.e., capabilities that are examined when checking if a process can perform a privileged operation
  - Capabilities can be dropped from effective set and reacquired
    - Operate with least privilege....
    - Reacquisition possible only if capability is in permitted set

[TLPI §39.3.3]

File permitted and effective capabilities

- *Permitted* : a set of capabilities that may be added to process's permitted set during *exec()*
- *Effective* : ⚠ a **single bit** that determines state of process's new effective set after *exec()* :
  - If set, all capabilities in process's new permitted set are also enabled in effective set
  - If not set, process's new effective set is empty
- File capabilities allow implementation of capabilities analog of set-UID-*root* program
  - Notable difference: setting effective bit off allows a program to start in **unprivileged** state
    - Set-UID/set-GID programs always start in **privileged** state

[TLPI §39.3.4]

Outline

Setting and viewing file capabilities from the shell

- *setcap(8)* sets capabilities on files
  - Requires privilege (`CAP_SETFCAP`)
  - E.g., to set `CAP_SYS_TIME` as a permitted and effective capability on an executable file:

```
$ cp /bin/date mydate
$ sudo setcap "cap_sys_time=pe" mydate
```

- *getcap(8)* displays capabilities associated with a file

```
$ getcap mydate
mydate = cap_sys_time+ep
```

- *filecap(8)* searches for files that have capabilities:

```
$ filecap                           # Report files in $PATH
$ sudo filecap -a 2> /dev/null      # Check all files on system
                          # "2>" to discard "not supported" messages
```

  - *filecap* is part of the *libcap-ng-utils* package

[TLPI §39.3.6]

cap/demo_file_caps.c

```
int main(int argc, char *argv[]) {
    cap_t caps = cap_get_proc();       /* Fetch process capabilities */
    char *str = cap_to_text(caps, NULL);
    printf("Capabilities: %s\n", str);
    ...
    if (argc > 1) {
        fd = open(argv[1], O_RDONLY);
        if (fd >= 0)
            printf("Successfully opened %s\n", argv[1]);
        else
            printf("Open failed: %s\n", strerror(errno));
    }
    exit(EXIT_SUCCESS);
}
```

- Display process capabilities
- Report result of opening file named in *argv[1]* (if present)

cap/demo_file_caps.c

```
$ id -u
1000
$ cc -o demo_file_caps demo_file_caps.c -lcap
$ ./demo_file_caps /etc/shadow
Capabilities: =
Open failed: Permission denied
$ ls -l /etc/shadow
----------. 1 root root 1974 Mar 15 08:09 /etc/shadow
```

- All steps in demos are done from unprivileged user ID 1000
- Binary has no capabilities ⇒ process gains no capabilities
- *open()* of `/etc/shadow` fails
  - Because `/etc/shadow` is readable only by privileged process
  - Process needs `CAP_DAC_READ_SEARCH` capability

cap/demo_file_caps.c

```
$ sudo setcap cap_dac_read_search=p demo_file_caps
$ ./demo_file_caps /etc/shadow
Capabilities: = cap_dac_read_search+p
Open failed: Permission denied
```

- Binary confers permitted capability to process, but capability is not effective
- Process gains capability in permitted set
- *open()* of `/etc/shadow` fails
  - Because `CAP_DAC_READ_SEARCH` is not in *effective* set

cap/demo_file_caps.c

```
$ sudo setcap cap_dac_read_search=pe demo_file_caps
$ ./demo_file_caps /etc/shadow
Capabilities: = cap_dac_read_search+ep
Successfully opened /etc/shadow
```

- Binary confers permitted capability and has effective bit on
- Process gains capability in permitted and effective sets
- *open()* of `/etc/shadow` succeeds

Outline

Notes for online practical sessions

- Small groups in **breakout rooms**
  - Write a note into Slack if you have a preferred group
- **We will go faster, if groups collaborate** on solving the exercise(s)
  - You can **share a screen** in your room
- I will circulate regularly between rooms to answer questions
- Zoom has an "**Ask for help**" button...
- **Keep an eye on the `#general` Slack channel**
  - Perhaps with further info about exercise;
  - Or a note that the exercise merges into a break
- When your room has finished, write a message in the Slack channel: **"***** Room X has finished *****"**
  - Then I have an idea of how many people have finished

Shared screen etiquette

- It may help your colleagues if you **use a larger than normal font!**
  - In many environments (e.g., *xterm*, *VS Code*), we can adjust the font size with `Control`+`Shift`+"`+`" and `Control`+"`-`"
  - Or (e.g., *emacs*) hold down `Control` key and use mouse wheel
- **Long shell prompts** make reading your shell session difficult
  - Use `PS1='$ '` or `PS1='# '`
- **Low contrast** color themes are difficult to read; change this if you can
- Turn on **line numbering** in your editor
  - In *vim* use: `:set number`
  - In *emacs* use: `M-x display-line-numbers-mode <RETURN>`
    - `M-x` means `Left-Alt`+`x`
- For collaborative editing, **relative line-numbering is evil**....
  - In *vim* use: `:set nornu`
  - In *emacs*, the following should suffice:

```
M-: (display-line-numbers-mode) <RETURN>
M-: (setq display-line-numbers 'absolute) <RETURN>
```

    - `M-:` means `Left-Alt`+`Shift`+`:`

Using *tmate* in in-person practical sessions

In order to share an X-term session with others, do the following:

- Enter the command *tmate* in an X-term, and you'll see the following:

```
$ tmate
...
Connecting to ssh.tmate.io...
Note: clear your terminal before sharing readonly access
web session read only: ...
ssh session read only: ssh S0mErAnD0m5Tr1Ng@lon1.tmate.io
web session: ...
ssh session: ssh S0mEoTheRrAnD0m5Tr1Ng@lon1.tmate.io
```

- Share last "ssh" string with colleague(s) via Slack or another channel
  - Or: "ssh session read only" string gives others read-only access
- Your colleagues should paste that string into an X-term...
- Now, you are sharing an X-term session in which anyone can type
  - Any "mate" can cut the connection to the session with the 3-character sequence `<ENTER>` ~ .
- To see above message again: `tmate show-messages`

Exercises

1. Compile and run the `cap/demo_file_caps` program, without adding any capabilities to the file, and verify that when you run the binary, the process has no capabilities:

```
$ cc -o demo_file_caps demo_file_caps.c -lcap
$ ./demo_file_caps
```

   - The string "=" means all capability sets empty.

2. Now make the binary set-UID-*root*:

```
$ sudo chown root demo_file_caps  # Change owner to root
$ sudo chmod u+s demo_file_caps   # Turn on set-UID bit
$ ls -l demo_file_caps            # Verify
-rwsr-xr-x. 1 root mtk 8624 Oct  1 13:19 demo_file_caps
```

3. Run the binary and verify that the process gains all capabilities. (The string "=ep" means "all capabilities in the permitted + effective sets".)
   - If the process does not gain all capabilities, check whether the filesystem is mounted with the `nosuid` option (`findmnt -T <dir>`). If it is, either remount the filesystem without that option or do the exercise on a filesystem that is not mounted with `nosuid` (typically, `/tmp` should work).

Exercises

4. Take the existing set-UID-*root* binary, add a permitted capability to it, and set the effective capability bit:

```
$ sudo setcap cap_dac_read_search=pe demo_file_caps
```

5. When you now run the binary, what capabilities does the process have?
6. Suppose you assign empty capability sets to the binary. When you execute the binary, what capabilities does the process then have?

```
$ sudo setcap = demo_file_caps
```

7. Use the following command to remove capabilities from the binary and verify that when executed, the binary once more grants all capabilities to the process:

```
$ sudo setcap -r demo_file_caps
```


Outline

Textual representation of capabilities

- Both *setcap(8)* and *getcap(8)* work with **textual representations** of capabilities
  - Syntax described in *cap_from_text(3)* manual page
- String read left to right, containing space-separated clauses
  - (The capability sets are initially considered to be empty)
- Clause: *caps-list operator flags* [*operator flags*] ...
  - *caps-list* is comma-separated list of capability names, or *all*
  - *operator* is `+`, `-`, or `=`
  - *flags* is zero or more of *p* (permitted), *e* (effective), or *i* (inheritable)
  - Clause can contain multiple [*operator flags*] parts:
    - E.g., `"cap_sys_time+p-i"` (is same as `"cap_sys_time+p cap_sys_time-i"`)

Textual representation of capabilities

Operators:

- `+` operator: raise capabilities in sets specified by *flags*
- `-` operator: lower capabilities in sets specified by *flags*
- `=` operator:
  - Raise capabilities in sets specified by *flags*; lower those capabilities in remaining sets
    - So, `"CAP_KILL=p"` is same as `"CAP_KILL+p-ie"`
  - *caps-list* can be omitted; defaults to *all*
  - *flags* can be omitted ⇒ clear capabilities from all sets ⇒ Thus: `"="` means clear all capabilities in all sets
- What does `"=p cap_kill,cap_sys_admin+e"` mean?
  - All capabilities in permitted set, plus `CAP_KILL` and `CAP_SYS_ADMIN` in effective set

Outline

Exercises

1. What capability bits are enabled by each of the following text-form capability specifications?
   - `"=p"`
   - `"="`
   - `"cap_setuid=p cap_sys_time+pie"`
   - `"=p cap_kill-p"`
   - `"cap_kill=p = cap_sys_admin+pe"`
   - `"cap_chown=i cap_kill=pe cap_setfcap,cap_chown=p"`
2. The program `cap/cap_text.c` takes a single command-line argument, which is a text-form capability string. It converts that string to an in-memory representation and then iterates through the set of all capabilities, printing out the state of each capability within the permitted, effective, and inheritable sets. It thus provides a method of verifying your interpretation of text-form capability strings. Try supplying each of the above strings as an argument to the program (**remember to enclose the entire string in quotes!**) and check the results against your answers to the previous exercise.

Outline

Transformation of process capabilities during *exec*

- During *execve()*, process's capabilities are transformed:

```
P'(perm) = F(perm) & P(bset)

P'(eff) = F(eff) ? P'(perm) : 0
```

  - *P()* / *P'()*: process capability set before/after *exec*
  - *F()*: file capability set (**of file that is being execed**)
- New permitted set for process comes from file permitted set ANDed with *capability bounding set (bset)*
  - ⚠ Note that *P(perm)* has no effect on *P'(perm)*
- New effective set is either 0 or same as new permitted set
- ⚠ Transformation **rules above are a simplification** that ignores process+file inheritable sets and process ambient set
  - In most cases, those sets are empty (i.e., 0)

Transformation of process capabilities during *exec*

- Commonly, process bounding set contains all capabilities
  - Removing capabilities from bounding set provides a way to limit capabilities that process gains during *execve()*
    - (We won't go into further detail on bounding set)
- Therefore transformation rule for process permitted set:

```
P'(perm) = F(perm) & P(bset)
```

commonly simplifies to:

```
P'(perm) = F(perm)
```

[TLPI §39.5]

Outline

Capabilities and UID transitions

- Various system calls change process credentials, subject to rules:
  - If process has `CAP_SETUID` (`CAP_SETGID`), arbitrary changes can be made to UIDs (GIDs)
  - Otherwise, can change ID only to value of another ID in same category
    - E.g., effective UID can be made same as real UID or saved set-UID

Capabilities and UID transitions

- What is effect on process capabilities for transitions to/from UID 0?
  - If rUID, eUID, or sUID was zero, and *set*uid()* renders them all nonzero:
    - Permitted, effective, and ambient sets are cleared
  - If eUID changes from zero to nonzero value:
    - Effective capability set is cleared
  - If eUID changes from nonzero value to 0:
    - Permitted set is copied to effective set
    - (Transition possible even if `CAP_SETUID` is not in process's effective set, so long as either rUID or sUID is 0)
- This behavior maps traditional privilege semantics of set-UID-*root* programs onto capabilities model

[TLPI §39.6]

Outline

Exercises

The `cap/setuid_root_cap_dumb.c` program can be used to verify the effect of UID transitions on process capabilities. This program uses various *set*uid()* calls to change the process's UIDs between zero and nonzero values, and prints out the state of the process's capabilities after each step.

1. Read the code of the *main()* function to understand what the program is doing (ignore the use of a command-line argument that triggers the use of `SECBIT_NO_SETUID_FIXUP`), and then compile it:

```
$ 'PS1='$ '
$ cc -o setuid_root_cap_dumb setuid_root_cap_dumb.c -lcap  # Or use make(1)
```

2. Make the program set-UID-*root*; assign a file permitted capability and enable the file effective capability bit:

```
$ sudo chown root setuid_root_cap_dumb
$ sudo chmod u+s setuid_root_cap_dumb            # Turn on set-UID bit
$ sudo setcap cap_setpcap=pe setuid_root_cap_dumb
```

3. Run the program and explain the results:

```
$ ./setuid_root_cap_dumb
```

Linux Security and Isolation APIs Essentials

Namespaces

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Rev: # 6f75b3d2e02f

Outline

Namespaces

- A namespace (NS) "wraps" some global system resource to provide resource isolation
- Linux supports multiple NS types
  - UTS, mount, network, ..., each governing different resources
- For each NS type:
  - Multiple **instances** of NS may exist on a system
    - At system boot, there is one instance of each NS type–the so-called **initial namespace instance**
  - Each process resides in one NS instance
  - To processes inside NS instance, it appears that only they can see/modify corresponding global resource
    - Processes are unaware of other instances of resource
- When new process is created via *fork()*, it resides in same set of NSs as parent

Outline

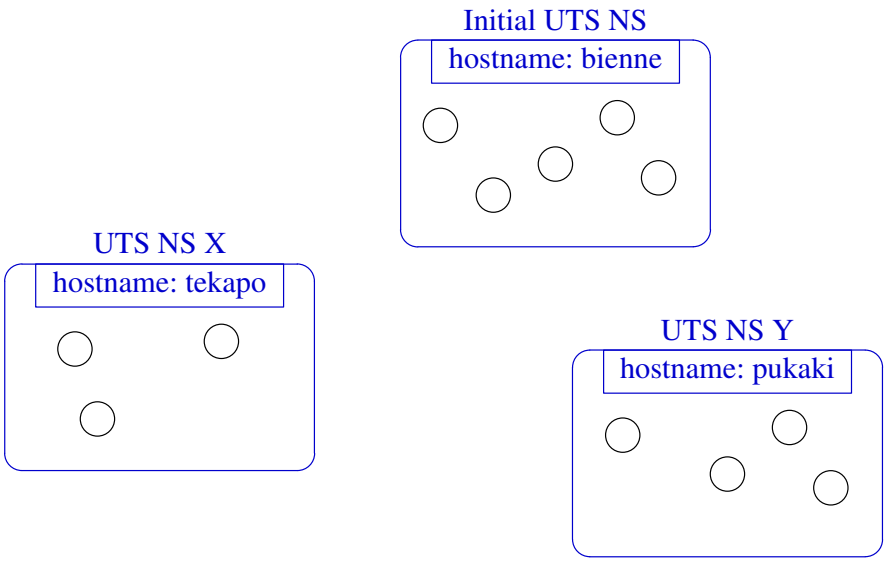
UTS namespaces

- UTS NSs are simple, and so provide an easy example
- Isolate two system identifiers returned by *uname(2)*
  - *nodename*: system hostname (set by *sethostname(2)*)
  - *domainname*: NIS domain name (set by *setdomainname(2)*)
- Container configuration scripts might tailor their actions based on these IDs
  - E.g., nodename could be used with DHCP, to obtain IP address for container
- "UTS" comes from *struct utsname* argument of *uname(2)*
  - Structure name derives from "UNIX Timesharing System"

UTS namespaces

- Running system may have multiple UTS NS instances
- Processes within single instance access (get/set) same *nodename* and *domainname*
- Each NS instance has its own *nodename* and *domainname*
  - Changes to *nodename* and *domainname* in one NS instance are invisible to other instances

UTS namespace instances



Each UTS NS contains a set of processes (the circles) which see/modify same hostname (and domain name, not shown)

Outline

Some "magic" symlinks

- Each process has some symlink files in `/proc/PID/ns`

```
/proc/PID/ns/cgroup     # Cgroup NS instance
/proc/PID/ns/ipc        # IPC NS instance
/proc/PID/ns/mnt        # Mount NS instance
/proc/PID/ns/net        # Network NS instance
/proc/PID/ns/pid        # PID NS instance
/proc/PID/ns/time       # Time NS instance
/proc/PID/ns/user       # User NS instance
/proc/PID/ns/uts        # UTS NS instance
```

  - One symlink for each of the NS types

Some "magic" symlinks

- Target of symlink tells us which NS instance process is in:

```
$ readlink /proc/$$/ns/uts
uts:[4026531838]
```

  - Content has form: *ns-type* : [*magic-inode-#*]
    - (*inode-#* comes from internally mounted NS filesystem)
- Various uses for these symlinks, including:
  - **If processes show same symlink target, they are in same NS**

The *unshare(1)* and *nsenter(1)* commands

There are shell commands for working with namespaces...

- *unshare(1)* creates new NSs and executes a command in those NSs:

```
unshare [options] [command [arg...]]
```

  - *command* defaults to *sh*
- *nsenter(1)* steps into already existing NS(s) and executes a command:

```
nsenter [options] [command [arg...]]
```

  - *command* defaults to *sh*

The *unshare(1)* and *nsenter(1)* commands

unshare(1) and *nsenter(1)* have options for specifying NS types:

```
unshare [options] [command [arguments]]
  -C      Create new cgroup NS
  -i      Create new IPC NS
  -m      Create new mount NS
  -n      Create new network NS
  -p      Create new PID NS
  -T      Create new time NS
  -u      Create new UTS NS
  -U      Create new user NS
```

```
nsenter [options] [command [arguments]]
  -t PID  PID of process whose NSs should be entered
  -C      Enter cgroup NS of target process
  -i      Enter IPC NS of target process
  -m      Enter mount NS of target process
  -n      Enter network NS of target process
  -p      Enter PID NS of target process
  -T      Enter time NS of target process
  -u      Enter UTS NS of target process
  -U      Enter user NS of target process
  -a      Enter all NSs of target process
```


Outline

Demo

- Start two terminal windows (*sh1*, *sh2*) in initial UTS NS

```
sh1$ hostname      # Show hostname in initial UTS NS
bienne
```

```
sh2$ hostname
bienne
```

- In *sh2*, create new UTS NS, and change hostname

```
$ SUDO_PS1='sh2# ' sudo unshare -u bash --norc
sh2# hostname langwied      # Change hostname
sh2# hostname               # Verify change
langwied
```

  - *sudo(8)* because we need privilege (`CAP_SYS_ADMIN`) to create a UTS NS
    - We set `SUDO_PS1` so shell has a distinctive prompt. Setting this environment variable causes *sudo(8)* to set `PS1` for the command that it executes. (`PS1` defines the prompt displayed by the shell.) The *bash --norc* option prevents the execution of shell start-up scripts that might modify `PS1`.

Demo

- In *sh1*, verify that hostname is unchanged:

```
sh1$ hostname
bienne
```

- Compare `/proc/PID/ns/uts` symlinks in two shells

```
sh1$ readlink /proc/$$/ns/uts
uts:[4026531838]
```

```
sh2# readlink /proc/$$/ns/uts
uts:[4026532855]
```

  - The two shells are in different UTS NSs

Demo

- Discover the PID of *sh2*:

```
sh2# echo $$
5912
```

- From *sh1*, use *nsenter(1)* to create a new shell that is in same NS as *sh2*:

```
sh1$ SUDO_PS1='sh3# ' sudo nsenter -t 5912 -u
sh3# hostname
langwied
sh3# readlink /proc/$$/ns/uts
uts:[4026532855]
```

  - Comparing the symlink values, we can see that this shell (`sh3#`) is in the second (`sh2#`) UTS NS

Outline

Namespace APIs

- Programs can use various system calls to work with NSs:
  - *clone(2)*: create new (child) process in new NS(s)
  - *unshare(2)*: create new NS(s) and move caller into it/them
    - Used by *unshare(1)* command
  - *setns(2)*: move calling process to another (existing) NS instance
    - Used by *nsenter(1)* command
  - (We revisit these APIs in detail later)

The Linux namespaces

- Linux supports following NS types:

Mount	`CLONE_NEWNS`	2002 (v2.4.19)
UTS	`CLONE_NEWUTS`	2006 (v2.6.19)
IPC	`CLONE_NEWIPC`	2006 (v2.6.19)
PID	`CLONE_NEWPID`	2008 (v2.6.24)
Network	`CLONE_NEWNET`	2009 (≈v2.6.29)
User	`CLONE_NEWUSER`	2013 (v3.8)
Cgroup	`CLONE_NEWCGROUP`	2016 (v4.6)
Time	`CLONE_NEWTIME`	2020 (v5.6)

- Above list includes corresponding *clone()* flag and year + kernel version for "milestone" release
- **Note**: we *won't* cover all NS types in this course

Privilege requirements for creating namespaces

- Creating **user** NS instances requires no privileges
- Creating instances of **other** (nonuser) NS types requires privilege
  - `CAP_SYS_ADMIN`

Combining namespace types

- It's possible to use individual NS types
  - E.g., mount NSs (first NS type) were invented to solve specific use cases
- But, often, several NS types are combined for an application
  - E.g., the use of PID, IPC, or cgroup NSs typically requires corresponding use of mount NSs
    - Because certain filesystems are commonly mounted for PID, IPC, and cgroup NSs
- In container-style frameworks, most or all NS types are used in concert

Sources of further information

- See my LWN.net article series *Namespaces in operation*
  - `https://lwn.net/Articles/531114/`
  - Many example programs and shell sessions...
  - Source code tarball for course includes all of that code, with a few important updates
  - A few details have subsequently changed (see my post-publication comments at end of some articles)
- *namespaces(7)*, *user_namespaces(7)*, *pid_namespaces(7)*, *mount_namespaces(7)*, *cgroup_namespaces(7)*, *uts_namespaces(7)*, *network_namespaces(7)*, *time_namespaces(7)*, *ipc_namespaces(7)*
- "Linux containers in 500 lines of code"
  - `https://blog.lizzie.io/linux-containers-in-500-loc.html`
  - (But note: uses cgroups v1)

Outline

Mount namespaces (CLONE_NEWNS)

- First namespace type (merged into mainline in 2002)
- Isolation of set of mounts seen by process(es)
  - A mount is a tuple that includes:
    - Mount source (e.g., device)
    - Pathname (mount point)
    - ID of parent mount
- Mount NSs allow processes to see distinct sets of mounts
  - Process's view of filesystem (FS) tree is defined by (hierarchically related) set of mounts
  - $\Rightarrow$ processes in different mount NSs see different FS trees

Mount namespaces (CLONE_NEWNS)

- Created via *clone()* or *unshare()* with CLONE_NEWNS flag
  - NS == "new namespace": no one foresaw that there might be further NS types...
  - **New NS inherits copy of mount list** from NS of creating process

Mount namespaces: use cases

- Per-process, private filesystem trees
  - (See also *pam_namespace(8)*)
- Mount new `/proc` FS without side effects
  - Useful when creating PID NS
  - (There are analogous use cases for IPC and cgroup NSs)
- Jailing in the manner of *chroot*, but more flexible and secure
  - Can set process up with different root directory, and subset of available filesystems

Mount namespaces demo

- In first terminal window (in initial mount NS), create a directory to be used as root of small tree of mounts:

```
$ mkdir /tmp/x
```

- Mount a *tmpfs* filesystem at that location, and create further directories that will be used as (child) mount points:

```
$ sudo mount -t tmpfs none /tmp/x
$ mkdir /tmp/x/{aaa,bbb}
```

- In a second terminal, create a new mount NS (NS 2), and create a new mount (`/tmp/x/bbb`) in that NS:

```
$ SUDO_PS1='ns2# ' sudo unshare --mount bash --norc
ns2# mount -t tmpfs none /tmp/x/bbb
```

Mount namespaces demo

- Verify the subtree of mounts in NS 2:

```
ns2# findmnt -a -o target -R /tmp/x
TARGET
/tmp/x
`-/tmp/x/bbb
```

- In first terminal (initial NS), create a mount (`/tmp/x/aaa`), and verify that mount `/tmp/x/bbb` is not present:

```
$ sudo mount -t tmpfs none /tmp/x/aaa
$ findmnt -a -o target -R /tmp/x
TARGET
/tmp/x
`-/tmp/x/aaa
```

- Show that `/tmp/x/aaa` mount is not present in NS 2:

```
$ findmnt -a -o target -R /tmp/x
TARGET
/tmp/x
`-/tmp/x/bbb
```


Shared subtrees and mount propagation

For time reasons, we will omit some important features:

- **Shared subtrees and mount propagation types**
- Allow (controlled, partial) reversal of isolation provided by mount NSs
  - IOW: initial mount NS implementation provided **too much** isolation for various use cases
  - Permit mount/unmount events in one mount NS to automatically propagate to other mount NSs
    - Classic example use case: mount optical disk in one NS, and have mount appear in all NSs
- See *mount_namespaces(7)*

Outline

PID namespaces (`CLONE_NEWPID`)

- Isolate process ID number space
  - ⇒ processes in different PID NSs can have same PID
- Benefits:
  - Allow processes inside containers to maintain same PIDs when container is migrated to different host
    - "Container live migration", implemented using CRIU ("Checkpoint restore in userspace"); `https://lisas.de/~adrian/container-live-migration-article.pdf`, `https://www.youtube.com/watch?v=FwbZuRMdO94`
  - Allows per-container *init* process (PID 1) that manages container initialization and reaping of orphaned children

PID namespace hierarchies

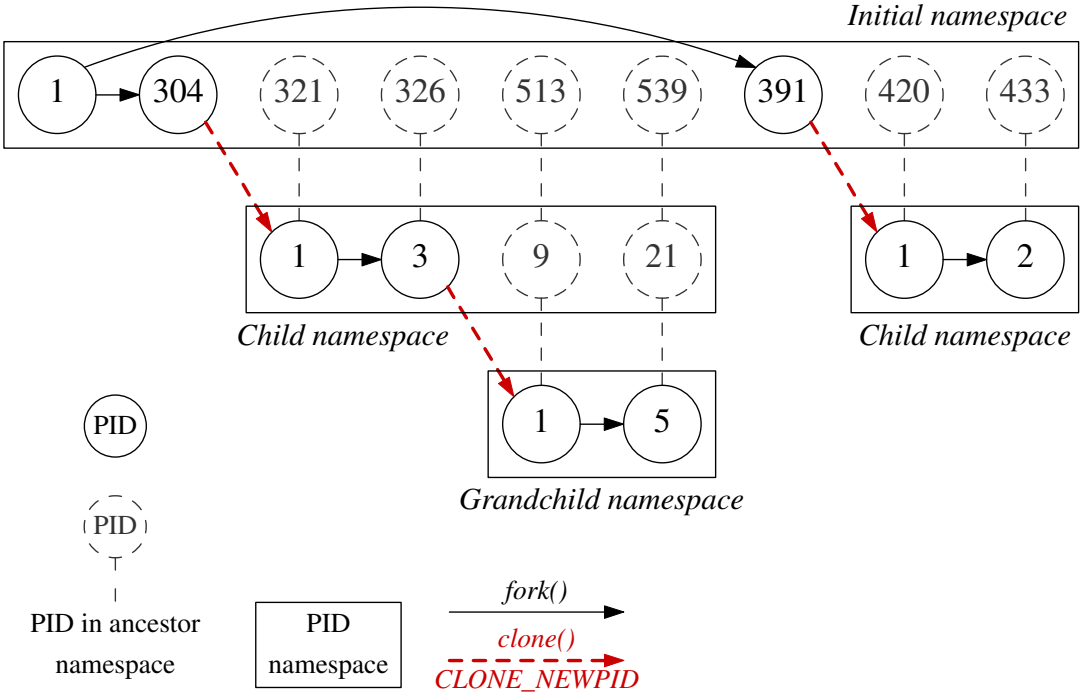
- Unlike (most) other NS types, PID NSs form a hierarchy
  - Each PID NS has a parent, going back to initial PID NS
  - **Parent** of PID NS is PID NS of caller of *clone()* or *unshare()*

PID namespace hierarchies

- A process is a member of its immediate PID NS, but is also visible in each ancestor PID NS
- Process will (typically) have different PID in each PID NS in which it is visible!
- A process in **initial PID NS can "see" all processes** in all PID NSs
  - See == employ syscalls on, send signals to, ...
- A processes in a lower NS won't be able to "see" any processes that are members only of ancestor NSs
  - Can see only peers in same NS + members of descendant NSs

A PID namespace hierarchy

A process is also visible in all ancestor PID namespaces



PID namespaces and PIDs

- *getpid()* returns caller's PID **inside caller's PID NS**
- When making syscalls and using `/proc` in outer NSs, process in a descendant NS is referred to by its PID in **caller's** NS
- A caller's parent might be in a different PID NS
  - *getppid()* returns 0!
- Via `/proc/PID/status`, we can see process's IDs in PID NSs of which it is a member
  - `NStgid`: thread group ID (PID!) in successively nested PID NSs, starting (at left) from NS of reading process
    - `NSpid`: thread(!) ID in successively nested PID namespaces
  - See *proc(5)* and `namespaces/pid_namespaces.go`

PID namespaces and /proc/PID

- /proc/PID directories contain info about processes corresponding to a PID NS
  - Allows us to introspect system
  - Without /proc, many systems tools will fail to work
    - *ps*, *top*, etc.
  - Some library functions also rely on /proc
    - E.g., *fexecve(3)*
  - ⇒ create new mount NS at same time, and remount /proc
- To mount /proc:

```
mount -t proc proc /proc
```

  - Or use *mount(2)*:

```
mount("proc", "/proc", "proc", 0, NULL)
```


PID namespaces and /proc/PID

- Mount and PID namespaces are orthogonal
- In new PID NS, we'll see /proc/PID of parent NS until we stack a new mount on /proc
  - **But note**: /proc/self always provides process with info about itself, regardless of whether /proc corresponds to process's PID NS

PID namespaces and *init*

First process inside new PID NS is special:

- Gets PID 1 (inside the NS)
- Fulfills role of *init*
  - Performs "system" initialization
  - Becomes parent of orphaned children
- If killed/terminated, all other processes in NS are terminated (`SIGKILL`), and NS is torn down
  - And it is no longer possible to *fork()* new processes into NS (after *unshare()* or *setns()*)
- (All of the above perfectly supports model of containers as virtual systems)

PID namespaces demo

- Create a PID NS and mount a `/proc` filesystem for that NS:

```
$ sudo unshare --pid --fork --mount-proc dash
```

- Inside PID NS, display PID of shell, and start a *sleep* process and display its PID:

```
# echo $$
1
# sleep 1000 &
# pidof sleep                   # 'pidof' used PID 3
2
```

- Take a look in `/proc`:

```
# ls -1 /proc
1                               # dash
2                               # sleep
4                               # ls
acpi
...
```

  - PIDs outside NS are not visible

PID namespaces demo

- From another terminal window (in initial PID NS), display PID of *dash* and *sleep*:

```
$ pidof dash
22645
$ pidof sleep
22677
```

  - Processes are visible outside NS, but with different PIDs!
- If we kill *init* process of a PID NS, all other processes in NS are also killed:

```
$ sudo kill -9 22645        # Kill PID 1 in inner NS
$ sudo kill -9 22677        # Is 'sleep' process still present?
bash: kill: (22677) - No such process
```


Notes

Notes

Notes

Linux Security and Isolation APIs Essentials

Namespaces APIs

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Rev: # 6f75b3d2e02f

Outline

Overview of namespaces API

- System calls:
  - *clone()*: create new NS(s) (while creating new process)
  - *unshare()*: create new NS(s) and move caller into it/them
    - Analogous shell command: *unshare(1)*
  - *clone()* and *unshare()* can employ one (or more) of flags: `CLONE_NEWCGROUP`, `CLONE_NEWIPC`, `CLONE_NEWNET`, `CLONE_NEWNS`, `CLONE_NEWPID`, `CLONE_NEWTIME` (*unshare* only), `CLONE_NEWUSER`, `CLONE_NEWUTS`
  - Creating new NS instance requires `CAP_SYS_ADMIN`
    - Except user NSs, which require no capabilities
  - *setns()*: move caller to another (existing) NS instance
    - Analogous shell command: *nsenter(1)*
- `/proc` files
  - `/proc/PID/ns/*` files (+ other NS-specific files)

Outline

The *clone()* system call

```
#include <sched.h>
int clone(int (*child_func)(void *), void *stack,
          int flags, void *arg);
```

- Creates new child process (like *fork()*)
- Much lower-level API that gives control of many facets of process/thread creation
  - Used to implement *pthread_create()*
  - Can be used to implement *fork()* (glibc does this)
- Above prototype is actually for glibc *clone()* wrapper function
  - Underlying syscall has somewhat different arguments

The *clone()* system call

```
#include <sched.h>
int clone(int (*child_func)(void *), void *stack,
          int flags, void *arg);
```

- Returns PID of new process as function result
- New process begins execution by calling "start" function *child_func*, of form:

```
int child_func(void *arg) {
    ...
}
```

- *arg* is argument to be given in call to *child_func*

The *clone()* system call

```
#include <sched.h>
int clone(int (*child_func)(void *), void *stack,
          int flags, void *arg);
```

- *flags* consists of flag bits ORed with signal number
  - Signal is delivered to caller when child terminates (like traditional `SIGCHLD`)
- 20+ flag bits spanning many different pieces of functionality
  - Use one or more of `CLONE_NEW*` flags to place new process in newly created namespace(s)
- *stack* points to top of region to be used for child's (downwardly growing) stack

Create a (new process and) new namespace with *clone()*

```
demo_uts_namespaces <hostname>
```

- Uses *clone()* to create child process in new UTS namespace
- Child changes hostname in new UTS namespace
- Parent and child fetch (*uname(2)*) and display hostname

namespaces/demo_uts_namespaces.c

```
int main(int argc, char *argv[]) {
    struct utsname uts;
    char *stack = mmap(..., STACK_SIZE, ...);
    ...
    pid_t child_pid = clone(childFunc, stack + STACK_SIZE,
                            CLONE_NEWUTS | SIGCHLD, argv[1]);
    munmap(stack, STACK_SIZE);
    sleep(1);
    uname(&uts);
    printf("uts.nodename in parent: %s\n", uts.nodename);
    waitpid(child_pid, NULL, 0);      /* Wait for child */
}
```

- *clone()* creates new child process
- CLONE_NEWUTS creates new UTS NS
  - New process is placed in that NS
- Sleep, so child has time to change and display hostname
- Fetch and display hostname of parent's UTS NS

namespaces/demo_uts_namespaces.c

```
static int childFunc(void *arg) {
    sethostname(arg, strlen(arg));

    struct utsname uts;
    uname(&uts);
    printf("uts.nodename in child:  %s\n", uts.nodename);
    sleep(1000);
    return 0;           /* Terminates child */
}
```

- "Start" function executed by child created by *clone()*
- Change hostname in child's UTS NS
- Fetch and display hostname of child's UTS NS
- Sleep for a while, so child and NS continue to exist
- Child terminates when "start" function returns

namespaces/demo_uts_namespaces.c

Running the program demonstrates that the parent and child are in separate UTS namespaces:

```
$ uname -n      # Show hostname in initial UTS namespace
bienne
$ sudo ./demo_uts_namespaces tekapo
PID of child created by clone() is 14958
uts.nodename in child:  tekapo
uts.nodename in parent: bienne
```

- Privilege is needed to create the new UTS NS

Linux Security and Isolation APIs Essentials

User Namespaces

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Rev: # 6f75b3d2e02f

Outline

Introduction

- Milestone release: Linux 3.8 (Feb 2013)
  - User NSs can now be created by unprivileged users...
- Allow per-namespace mappings of UIDs and GIDs
  - I.e., process's UIDs and GIDs inside NS may be different from IDs outside NS
- Interesting use case: process has nonzero UID outside NS, and UID of 0 inside NS
  - $\Rightarrow$ Process has *root* privileges *for operations inside user NS*
    - We will learn what this means...

Relationships between user namespaces

- User NSs have a hierarchical relationship:
  - A user NS can have 0 or more child user NSs
  - Each user NS has parent NS, going back to initial user NS
    - Initial user NS == sole user NS that exists at boot time
  - Parent of a user NS == user NS of process that created this user NS using *clone()* or *unshare()*
- Parental relationship determines some rules about how capabilities work in NSs (later...)

Outline

Creating and joining a user NS

- New user NS is created with `CLONE_NEWUSER` flag
  - *clone()* ⇒ child is made a member of new user NS
  - *unshare()* ⇒ caller is made a member of new user NS
- Can join an existing user NS using *setns()*
  - Process must have `CAP_SYS_ADMIN` capability in target NS
    - (The capability requirement will become clearer later)

User namespaces and capabilities

- A process gains a full set of permitted and effective capabilities in the new/target user NS when:
  - It is the child of *clone()* that creates a new user NS
  - It creates and joins a new user NS using *unshare()*
  - It joins an existing user NS using *setns()*
- But, process has no capabilities in parent/previous user NS
  - ⚠ Even if it was *root* in that NS!

Example: namespaces/demo_userns.c

```
./demo_userns
```

- (Very) simple user NS demonstration program
- Uses *clone()* to create child in new user NS
- Child displays its UID, GID, and capabilities

Example: namespaces/demo_userns.c

```
#define STACK_SIZE (1024 * 1024)

int main(int argc, char *argv[]) {
    char *stack = mmap(..., STACK_SIZE);    /* Allocate memory for
                                               child's stack */
    pid_t pid = clone(childFunc, stack + STACK_SIZE,
                      CLONE_NEWUSER | SIGCHLD, argv[1]);
    printf("PID of child: %ld\n", (long) pid);

    munmap(stack, STACK_SIZE);              /* Deallocate stack */

    waitpid(pid, NULL, 0);
    exit(EXIT_SUCCESS);
}
```

- Use *clone()* to create a child in a new user NS
  - Child will execute *childFunc()*, with argument *argv[1]*
- Printing PID of child is useful for some demos...
- Wait for child to terminate

Example: namespaces/demo_userns.c

```
static int childFunc(void *arg) {
    for (;;) {
        printf("eUID = %ld; eGID = %ld; ",
                (long) geteuid(), (long) getegid());

        cap_t caps = cap_get_proc();
        char *str = cap_to_text(caps, NULL);
        printf("capabilities: %s\n", str);
        cap_free(caps);
        cap_free(str);

        if (arg == NULL)
            break;
        sleep(5);
    }
    return 0;
}
```

- Display PID, effective UID + GID, and capabilities
- If *arg* (*argv[1]*) was NULL, break out of loop
- Otherwise, redisplay IDs and capabilities every 5 seconds

Example: namespaces/demo_userns.c

```
$ id -u      # Display effective UID of shell process
1000
$ id -g      # Display effective GID of shell process
1000
$ ./demo_userns
eUID = 65534; eGID = 65534; capabilities: =ep
```

Upon running the program, we'll see something like the above

- Program was run from unprivileged user account
- =ep means child process has a full set of permitted and effective capabilities

Example: namespaces/demo_userns.c

```
$ id -u      # Display effective UID of shell process
1000
$ id -g      # Display effective GID of shell process
1000
$ ./demo_userns
eUID = 65534; eGID = 65534; capabilities: =ep
```

Displayed UID and GID are "strange"

- System calls such as *geteuid()* and *getegid()* always return credentials as they appear inside user NS where caller resides
- But, no mapping has yet been defined to map IDs outside user NS to IDs inside NS
- ⇒ when a UID is unmapped, system calls return value in `/proc/sys/kernel/overflowuid`
  - Unmapped GIDs ⇒ `/proc/sys/kernel/overflowgid`
  - Default value, 65534, chosen to be same as NFS `nobody` ID

Outline

UID and GID mappings

- One of first steps after creating a user NS is to define UID and GID mapping for NS
- Mappings for a user NS are defined by writing to 2 files: `/proc/PID/uid_map` and `/proc/PID/gid_map`
  - Each process in user NS has these files; writing to files of *any* process in the user NS suffices
  - Initially, these files are empty

UID and GID mappings

- Records written to/read from `uid_map` and `gid_map` have this form:

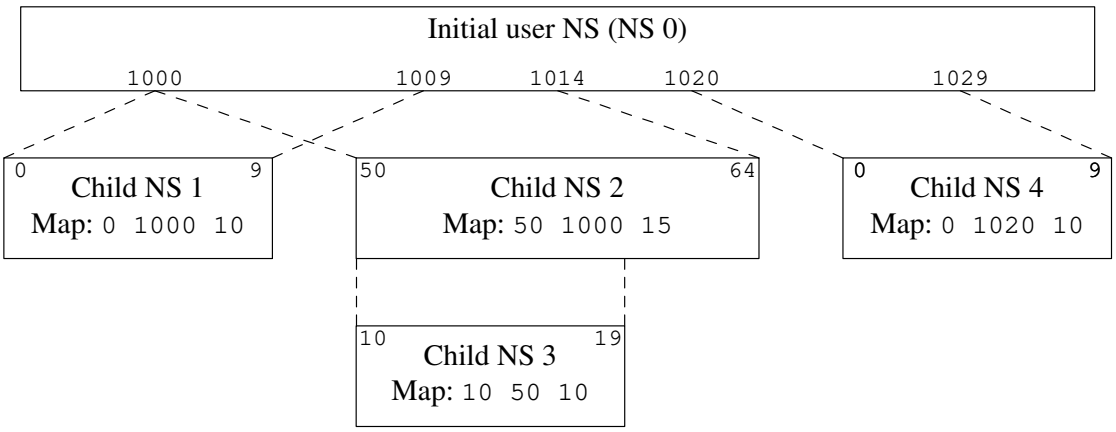
```
ID-inside-ns   ID-outside-ns   length
```

  - *ID-inside-ns* and *length* define range of IDs inside user NS that are to be mapped
  - *ID-outside-ns* defines start of corresponding mapped range in "outside" user NS
- E.g., following says that IDs 0...9 inside user NS map to IDs 1000...1009 in outside user NS

```
0     1000     10
```

- ⚠ To properly understand *ID-outside-ns*, we must first look at a picture

Understanding UID and GID maps

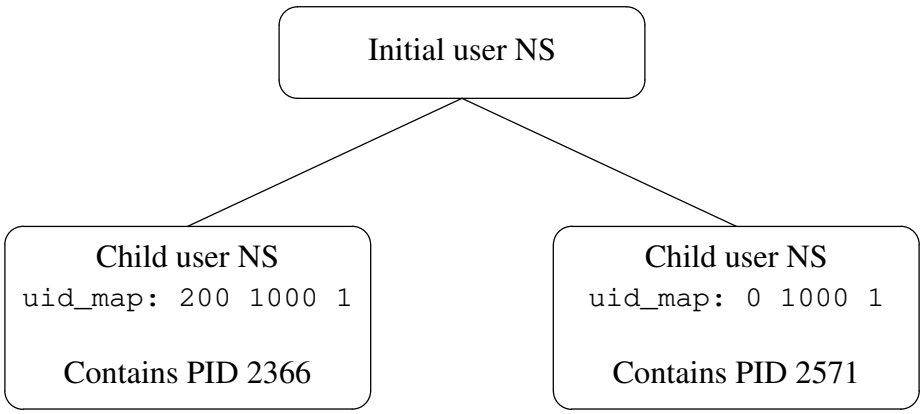


- "What does ID X in namespace Y map to in namespace Z?" means "what is the equivalent ID (if any) in namespace Z?"
- What does ID 5 in NS 1 map to in the initial NS (NS 0)?
- What does ID 5 in NS 1 map to in NS 2 and NS 3?
- What does ID 15 in NS 3 map to in NS 2 and NS 1?
- What does the UID 0 in NS 4 map to in NS 1?

Interpretation of *ID-outside-ns*

- ⚠ Interpretation of *ID-outside-ns* depends on whether process opening `uid_map`/`gid_map` is in same NS as *PID*
  - NB: contents of `uid_map`/`gid_map` are generated on the fly by the kernel, and can be different in different processes
- If "opener" and *PID* are in **same user NS**:
  - *ID-outside-ns* interpreted as **ID in parent user NS** of *PID*
  - Common case: process is writing its own mapping file
- If "opener" and *PID* are in **different user NSs**:
  - *ID-outside-ns* interpreted as **ID in opener's user NS**
  - Equivalent to previous case, if "opener" is (parent) process that created user NS using *clone()*
- ⚠ Only *ID-outside-ns* is interpreted; *ID-inside-ns* and *length* are always treated literally

Quiz: reading /proc/PID/uid_map



- If PID 2366 reads `/proc/2571/uid_map`, what should it see?
  - `0 200 1`
- If PID 2571 reads `/proc/2366/uid_map`, what should it see?
  - `200 0 1`

Example: updating a mapping file

- Let's run `demo_userns` with an argument, so it loops:

```
$ id -u      # Display user ID of shell
1000
$ id -G      # Display group IDs of shell
1000 10
$ ./demo_userns x
PID of child: 2810
eUID = 65534; eGID = 65534; capabilities: =ep
```

- Then we switch to another terminal window (i.e., a shell process in parent user NS), and write a UID mapping:

```
echo '0 1000 1' > /proc/2810/uid_map
```

- Returning to window where we ran `demo_userns`, we see:

```
eUID = 0; eGID = 65534; capabilities: =ep
```

Example: updating a mapping file

- But, if we go back to second terminal window, to create a GID mapping, we encounter a problem:

```
$ echo '0 1000 1' > /proc/2810/gid_map
bash: echo: write error: Operation not permitted
```

- There are **(many) rules** governing updates to mapping files
  - Inside the new user NS, user is getting full capabilities
  - **It is critical that capabilities can't leak**
    - I.e., user must not get more privileges than they would otherwise have **outside the NS**

Validity requirements for updating mapping files

If any of these rules are violated, *write()* fails with `EINVAL`:

- There is a limit on the number of lines that may be written
  - Since Linux 4.15 (2017): up to 340 lines
  - Linux 4.14 and earlier: up to 5 lines
- Each line contains 3 valid numbers, with *length* > 0, and a newline terminator
- The ID ranges specified by the lines may not overlap
  - (Because that would make IDs ambiguous)

Permission rules for updating mapping files

If any of these "permission" rules are violated when updating `uid_map` and `gid_map` files, *write()* fails with `EPERM`:

- Each map may be **updated only once**
- Writer must be in target user NS or in parent user NS
- The mapped IDs must have a mapping in parent user NS
- Writer must have following **capability in target user NS**
  - `CAP_SETUID` for `uid_map`
  - `CAP_SETGID` for `gid_map`

Permission rules for updating mapping files

As well as preceding rules, one of the following also applies:

- **Either**: writer has `CAP_SETUID` (for `uid_map`) or `CAP_SETGID` (for `gid_map`) **capability in parent user NS**:
  - ⇒ no further restrictions apply (more than one line may be written, and arbitrary UIDs/GIDs may be mapped)
- **Or**: otherwise, all of the following restrictions apply:
  - **Only a single line** may be written to `uid_map` (`gid_map`)
  - That line **maps only the writer's eUID** (eGID)
    - Usual case: we are writing a mapping for eUID/eGID of process that created the NS
  - eUID of writer must match eUID of creator of NS
    - (e**U**ID restriction also applies for `gid_map`)
  - For `gid_map` only: corresponding `/proc/PID/setgroups` file must have been previously updated with string "deny"
    - (Fix for a security bug in earlier kernels)

Example: updating a mapping file

- Going back to our earlier example:

```
$ echo '0 1000 1' > /proc/2810/gid_map
bash: echo: write error: Operation not permitted
$ echo 'deny' > /proc/2810/setgroups
$ echo '0 1000 1' > /proc/2810/gid_map
$ cat /proc/2810/gid_map
         0       1000          1
```

  - After writing "deny" to `/proc/PID/setgroups` file, we can update `gid_map`

- Upon returning to window running `demo_userns`, we see:

```
eUID = 0; eGID = 0; capabilities: =ep
```


Outline

Exercises

1. Try replicating the steps shown earlier on your system:
   - **Use the *id(1)* command to discover your UID and GID; you will need this information for a later step**.
   - Run the `namespaces/demo_userns.c` program with an argument (any string), so it loops. Verify that the child process has all capabilities.
   - Inspect (*readlink(1)*) the `/proc/PID/ns/user` symlink for the `demo_userns` child process and compare it with the `/proc/PID/ns/user` symlink for a shell running in the initial user namespace (for the latter, simply open a new shell window on your desktop). You should find that the two processes are in different user namespaces.
   - From a shell in the initial user NS, define UID and GID maps for the `demo_userns` child process (i.e., **for the UID and GID that you discovered in the first step**). Map the *ID-outside-ns* value for both IDs to IDs of your choice in the inner NS.
     - This step will involve writing to the `uid_map`, `setgroups`, and `gid_map` files in the `/proc/PID` directory.
   - Verify that the UID and GID displayed by the looping `demo_userns` program have changed.

   [Further exercises follow on the next slide]

Exercises

2. What are the contents of the UID and GID maps of a process in the initial user namespace?

```
$ cat /proc/1/uid_map
```

3. The script `namespaces/show_non_init_uid_maps.sh` shows the processes on the system that have a UID map that is different from the *init* process (PID 1). Included in the output of this script are the capabilities of each processes. Run this script to see examples of such processes. As well as noting the UID maps that these processes have, observe the capabilities of these processes.

Outline

Combining user namespaces with other namespaces

- Creating other (non-user) NSs requires `CAP_SYS_ADMIN`
- Creating user NSs requires no capabilities
  - And process in new user NS gets full capabilities
- $\Rightarrow$ We can create a user NS, and then create other NS types inside that user NS
  - I.e., two *clone()* or *unshare()* calls
- Actually, we can achieve desired result in one call; e.g.:

```
clone(child_func, stackptr, CLONE_NEWUSER | CLONE_NEWUTS, arg);
// or
unshare(CLONE_NEWUSER | CLONE_NEWUTS);
```

  - Kernel **creates user NS first**, then other NS types
    - And the other NSs are owned by the user NS

Notes

Notes

Linux Security and Isolation APIs Essentials

User Namespaces and Capabilities

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Rev: # 6f75b3d2e02f

Outline

What are the rules that determine the capabilities that a process has in a given user namespace?

User namespace hierarchies

- User NSs exist in a hierarchy
  - Each user NS has a parent, going back to initial user NS
- Parental relationship is established when user NS is created:
  - *clone()*: parent of new user NS is NS of caller of *clone()*
  - *unshare()*: parent of new user NS is caller's previous NS
- Parental relationship is significant because it plays a part in determining capabilities a process has in user NS

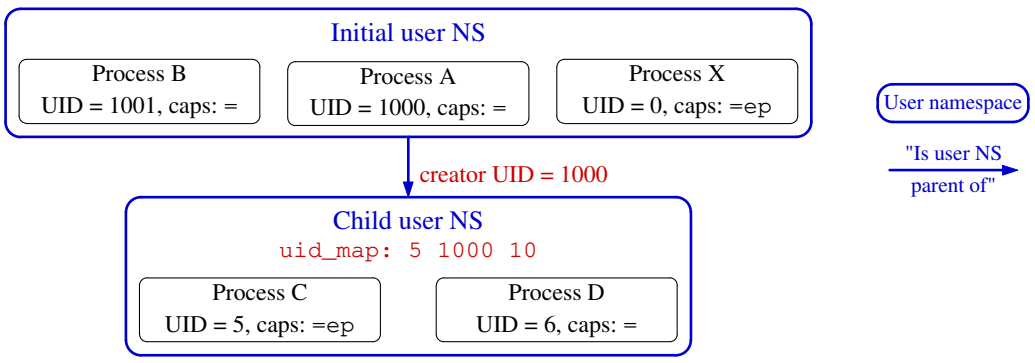
User namespaces and capabilities

- Whether a process has an effective capability inside a "target" user NS depends on several factors:
  - Whether the capability is present in process's effective set
  - Which user NS the process is a member of
  - The process's effective UID
  - The effective UID of process that created target user NS
  - The parental relationship between process's user NS and target user NS
- See also `namespaces/ns_capable.c`
  - (A program that encapsulates the rules described next)

Capability rules for user namespaces

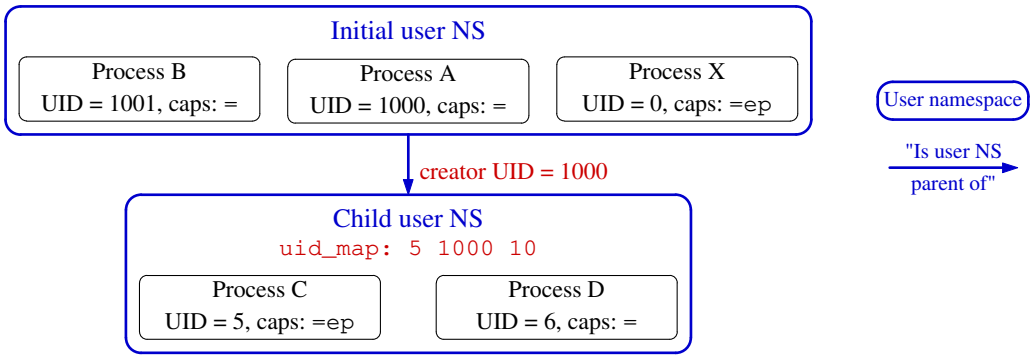
1. A process has a capability in a user NS if:
   - it is a **member of the user NS**, and
   - **capability is present in its effective set**
   - Note: this rule doesn't grant that capability in parent NS
2. A process that has a capability in a user NS **has the capability in all descendant user NSs** as well
   - I.e., members of user NS are not isolated from effects of privileged process in parent/ancestor user NS
3. A **process in a parent user NS that has same eUID as eUID of creator of user NS** has all capabilities in the NS
   - At creation time, **kernel records eUID of creator** as "owner" of user NS
   - By virtue of previous rule, process also has capabilities in all descendant user NSs

Quiz (who can signal a process in a child user NS?)



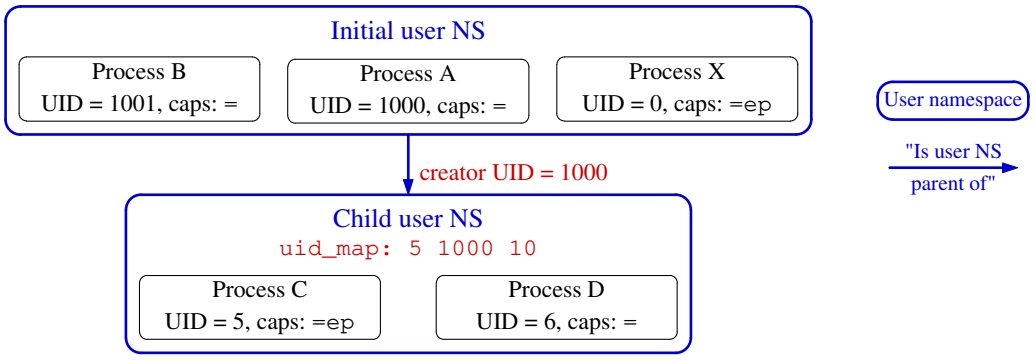
- Child user NS was created by a process with UID 1000
  - That process (which presumably was not A) had capabilities that allowed it to create a user NS with UID map with *length* > 1
- Process X has all capabilities in initial user NS
- Assume process A and process B have no capabilities in initial user NS
- Assume C was first process in child NS and has all capabilities in NS
- Process D has no capabilities

Quiz (who can signal a process in a child user NS?)



- Sending a signal requires UID match or `CAP_KILL` capability
- To which of B, C, D can process A send a signal?
- Can B send a signal to D? Can D send a signal to B?
- Can process X send a signal to processes C and D?
- Can process C send a signal to A? To B?
- Can C send a signal to D?

Quiz (who can signal a process in a child user NS?)



- A can't signal B, but can signal C (matching credentials) and D (because A has capabilities in D's NS)
- B can signal D (matching credentials); likewise, D can signal B
- X can signal C and D (because it has capabilities in parent user NS)
- C can signal A (credential match), but not B
- C can signal D, because it has capabilities in its NS

Outline

Exercises

1. As an unprivileged user, start two *sleep* processes, one as the unprivileged user and the other as UID 0:

```
$ id -u
1000
$ sleep 1000 &
$ sudo sleep 2000
```

As superuser, in another terminal window use *unshare* to create a user namespace (*–U*) with root mappings (*–r*) and run a shell in that namespace:

```
$ SUDO_PS1="ns2# " sudo unshare -U -r bash --norc
```

- (Root mappings == process's UID and GID in parent NS map to 0 in child NS)
- Setting the `SUDO_PS1` environment variable causes *sudo(8)* to set the `PS1` environment variable for the command that it executes. (`PS1` defines the prompt displayed by the shell.) The *bash --norc* option prevents the execution of shell start-up scripts that might change `PS1`.

[Exercises continue on next slide]

Exercises

Verify that the shell has a full set of capabilities and a UID map "`0 0 1`" (i.e., UID 0 in the parent namespace maps to UID 0 in the child user namespace):

```
ns2# grep -E 'Cap(Prm|Eff)' /proc/$$/status
ns2# cat /proc/$$/uid_map
```

From this shell, try to kill each of the *sleep* processes started above:

```
ns2# ps -o 'pid uid cmd' -C sleep   # Discover 'sleep' PIDs
...
ns2# kill -9 <PID-1>
ns2# kill -9 <PID-2>
```

Which of the *kill* commands succeeds? Why?

Outline

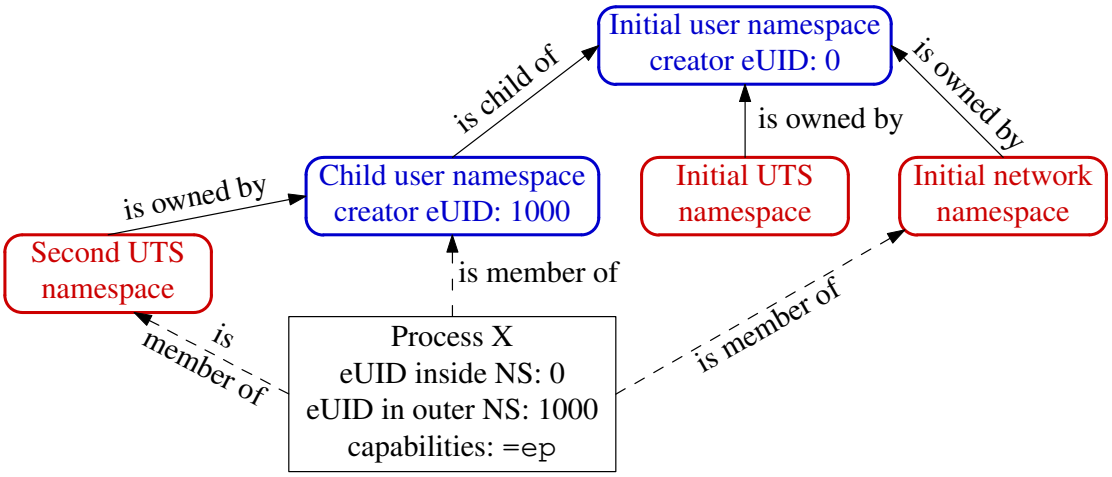
User namespaces and capabilities

- Kernel grants initial process in new user NS a full set of capabilities
- But, those capabilities are available **only for operations on objects governed by the new user NS**

User namespaces and capabilities

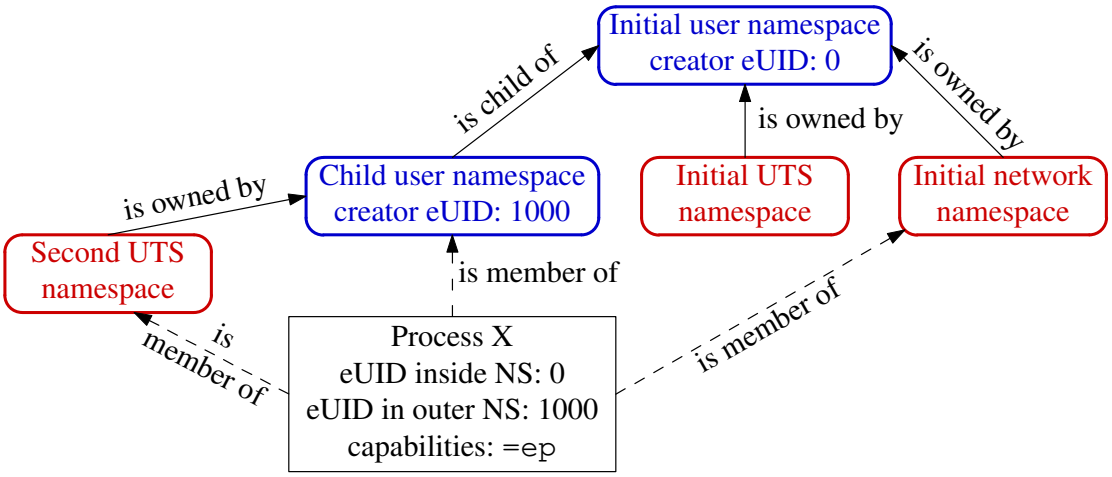
- **Kernel associates each non-user NS instance with a specific user NS instance**
  - Each non-user NS is "owned" by a user NS
  - When creating a new non-user NS, user NS of the creating process becomes the owner of the new NS
- Suppose a process operates on global resources governed by a (non-user) NS:
  - Privilege checks are done according to process's capabilities in user NS that owns the NS
- $\Rightarrow$ User NSs can deliver full capabilities inside a user NS without allowing capabilities in outer user NS(s)
  - (Barring kernel bugs)

User namespaces and capabilities–an example



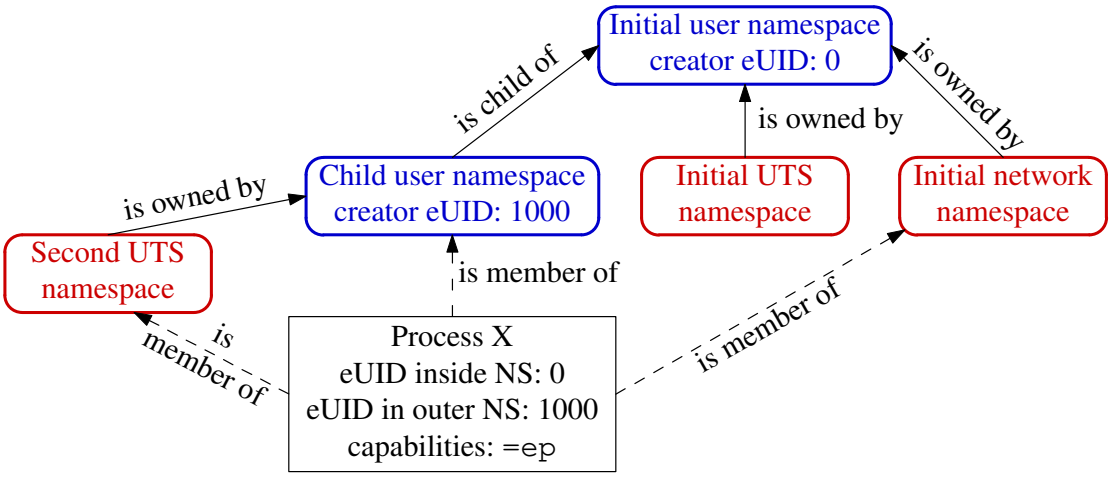
- Example scenario; X was created with: `unshare -Ur -u <prog>`
  - X is in a new user NS, created with root mappings
  - X is in a new UTS NS, which is owned by new user NS
  - X is in initial instance of all other NS types (e.g., network NS)

User namespaces and capabilities–an example



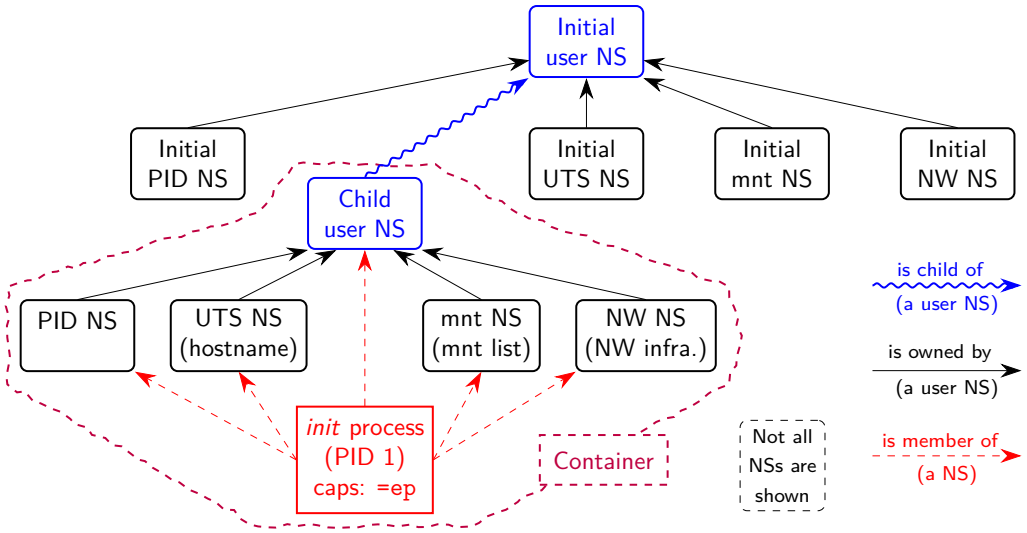
- Suppose X tries to change host name (`CAP_SYS_ADMIN`)
  - E.g., `hostname bienne`
- X is in second **UTS** NS
- Privileges checked according to X's capabilities in user NS that owns that UTS NS ⇒ succeeds (X has capabilities in user NS)

User namespaces and capabilities–an example



- Suppose X tries to bring network device up/down (`CAP_NET_ADMIN`)
  - E.g., `ip link set dev lo down`
- X is in initial **network** NS
- Privileges checked according to X's capabilities in user NS that owns network NS ⇒ attempt fails (no capabilities in initial user NS)

Containers and namespaces



- "Superuser" process in a container has **root power over resources governed by non-user NSs owned by container's user NS**
- And does **not** have privilege in outside user NS
  - (E.g., can't change mounts seen by processes outside container)

Demo: effect of capabilities in a user NS

- Create a shell in new user and UTS NSs:

```
$ unshare -Ur -u bash
# getpcaps $$
929: =ep                # Shell has all capabilities in its user NS
```

- Since this shell has all capabilities in user NS that owns its UTS NS, we can change hostname:

```
# hostname
bienne
# hostname langwied
# hostname
langwied
```

- But, this shell is in a network NS owned by **initial** user NS, and so can't turn a NW device down:

```
# ip link set dev lo down
RTNETLINK answers: Operation not permitted
```


What about resources not governed by namespaces?

- Some privileged operations relate to resources/features not (yet) governed by any namespace
  - E.g., load kernel modules, raise process nice values
- Having all capabilities in a (noninitial) user NS doesn't grant power to perform operations on features not currently governed by any NS
  - E.g., load/unload kernel modules, raise process nice values
  - IOW: to perform these operations, process must have the relevant capability in the **initial** user NS

Outline

Homework exercises

1. Using two terminal windows, and suitable *unshare* and *nsenter* commands, construct a scenario where, in addition to the initial user namespace, there is also a child user namespace and a grandchild user namespace. In this scenario, the grandchild user namespace has a member process (running, say, *sleep(1)*), but the child namespace does not have (i.e., no longer has) a member process. Even though the child namespace has no member processes, it is nevertheless pinned into existence by virtue of being the parent of the grandchild namespace.

   Once you have set up the scenario, verify the hierarchical relationship of the user namespaces and that the child user namespace has no member processes, using *either* of the following commands:

```
$ sudo lsns -t user --tree=owner -p $(pidof sleep)
$ cd lsp/namespaces; sudo go run namespaces_of.go --namespaces=user
```

   - In the output of *lsns*, you should see the value 0 for `NPROCS` (the number of processes in the namespace).

Linux Security and Isolation APIs Essentials

Control Groups (cgroups): Introduction

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Rev: # 6f75b3d2e02f

Outline

Goals

- We'll focus on:
  - General principles of operation; goals of cgroups
  - The `cgroup2` filesystem
  - Interacting with `cgroup2` filesystem using shell commands
    - By 2021, all major distros switched to cgroups v2, so we'll ignore cgroups v1
- We'll look **briefly** at some of the controllers

Resources

- Kernel documentation files
  - V2: `Documentation/admin-guide/cgroup-v2.rst`
  - V1: `Documentation/admin-guide/cgroup-v1/*.rst`
    - Before Linux 5.3: `Documentation/cgroup-v1/*.txt`
- *cgroups(7)* manual page
- Chris Down, *7 years of cgroup v2*, `https://www.youtube.com/watch?v=LX6fMlIYZcg`
- Neil Brown's (2014) LWN.net series on cgroups: `https://lwn.net/Articles/604609/`
  - Thought-provoking ideas on the meaning of grouping & hierarchy
- `https://lwn.net/Articles/484254/` – Tejun Heo's initial thoughts about redesigning cgroups (Feb 2012)
  - See also `https://lwn.net/Articles/484251/`, *Fixing Control Groups*, Jon Corbet, Feb 2012
- Other articles at `https://lwn.net/Kernel/Index/#Control_groups`

Outline

What are control groups?

- Two principal components:
  - A **mechanism for hierarchically grouping** processes
  - A set of **controllers** (kernel components) that manage, control, or monitor processes in cgroups
- Interface is via a pseudo-filesystem
- Cgroup manipulation takes form of filesystem operations, which might be done:
  - Via shell commands
  - Programmatically
  - Via management daemon (e.g., *systemd*)
  - Via your container framework's tools (e.g., LXC, Docker)

What do cgroups allow us to do?

- Limit resource usage of group
  - E.g., limit % of CPU available to group; limit amount of memory that group can use
- Resource accounting
  - Measure resources used by processes in group
- Limit device access
- Pin processes to CPU cores
- Shape network traffic
- Freeze a group
  - Freeze, restore, and checkpoint a group
- And more...

Terminology

- **Control group**: a group of processes that are bound together for purpose of resource management
- **(Resource) controller**: kernel component that controls or monitors processes in a cgroup
  - E.g., `memory` controller limits memory usage; `cpu` controller limits CPU usage
  - Also known as **subsystem**
    - (But that term is rather ambiguous because so generic)
- Cgroups are arranged in a **hierarchy**
  - Each cgroup can have zero or more child cgroups
  - Child cgroups **inherit** control settings from parent

Filesystem interface

- Cgroup filesystem **directory structure defines cgroups + cgroup hierarchy**
  - I.e., use *mkdir(2)* / *rmdir(2)* (or equivalent shell commands) to create cgroups
- Each **subdirectory contains automagically created files**
  - Some files are used to **manage the cgroup** itself
  - Other files are **controller-specific**
- Files in cgroup are used to:
  - **Define/display membership** of cgroup
  - **Control behavior** of processes in cgroup
  - **Expose information** about processes in cgroup (e.g., resource usage stats)

The cgroup2 filesystem

- On boot, *systemd* mounts v2 hierarchy at `/sys/fs/cgroup`

```
# mount -t cgroup2 none /sys/fs/cgroup
```

- The cgroups v2 mount is sometimes known as the "**unified hierarchy**"
  - Because all controllers are associated with a single hierarchy
  - By contrast, in v1 there were multiple hierarchies

Outline

Example: the pids controller

- pids ("process number") controller allows us to limit number of PIDs in cgroup (prevent *fork()* bombs!)
- Create new cgroup, and place shell's PID in that cgroup:

```
# mkdir /sys/fs/cgroup/mygrp
# echo $$
17273
# echo $$ > /sys/fs/cgroup/mygrp/cgroup.procs
```

  - cgroup.procs defines/displays PIDs in cgroup
  - (Note '#' prompt ⇒ all commands done as superuser)
- Moving a PID into a group automatically removes it from group of which it was formerly a member
  - I.e., a process is always a member of exactly one group in the hierarchy

Example: the pids controller

- Can read cgroup.procs to see PIDs in group:

```
# cat /sys/fs/cgroup/mygrp/cgroup.procs
17273
20591
```

  - Where did PID 20591 come from?
  - PID 20591 is *cat* command, created as a child of shell
    - Child process inherits cgroup membership from parent
- pids.current shows how many processes are in group:

```
# cat /sys/fs/cgroup/mygrp/pids.current
2
```

  - Two processes: shell + *cat*

Example: the pids controller

- We can limit number of PIDs in group using pids.max file:

```
# echo 5 > /sys/fs/cgroup/mygrp/pids.max
# for a in $(seq 1 5); do sleep 60 & done
[1] 21283
[2] 21284
[3] 21285
[4] 21286
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: Resource temporarily unavailable
```

  - (The shell retries a few times and then gives up)
  - pids.max defines/exposes limit on number of PIDs in cgroup

- From a **different** shell, examine pids.current:

```
$ cat /sys/fs/cgroup/mygrp/pids.current
5
```

  - Not possible from first shell (can't create more processes)

Outline

Creating cgroups

- Initially, all processes on system are members of **root cgroup**
- New cgroups are **created** by creating subdirectories under cgroup mount point:

```
# mkdir /sys/fs/cgroup/mygrp
```

- Relationships between cgroups are reflected by creating nested (arbitrarily deep) subdirectory structure

Destroying cgroups

An **empty cgroup** can be **destroyed** by removing directory

- **Empty** == last process in cgroup terminates or migrates to another cgroup **and** last child cgroup is removed
- Not necessary (or possible) to delete attribute files inside cgroup directory before deleting it

Placing a process in a cgroup

- To move a **process** to a cgroup, we write its PID to `cgroup.procs` file in corresponding subdirectory

```
# echo $$ > /sys/fs/cgroup/mygrp/cgroup.procs
```

  - In multithreaded process, moves all threads to cgroup
- ⚠ Can write only one PID at a time
  - Otherwise, *write()* fails with `EINVAL`

Viewing cgroup membership

- **To see PIDs in cgroup**, read `cgroup.procs` file
  - PIDs are newline-separated
  - Zombie processes do not appear in list
- ⚠ List is **not guaranteed to be sorted or free of duplicates**
  - PID might be moved out and back into cgroup or recycled while reading list

Cgroup membership details

- A **process can be member of just one cgroup**
  - That association defines attributes / parameters that apply to the process
- Adding a process to a different cgroup automatically removes it from previous cgroup
- On *fork()*, **child inherits cgroup membership(s)** of parent
  - Afterward, cgroup membership(s) of parent and child can be independently changed

/proc/PID/cgroup file

- `/proc/PID/cgroup` shows cgroup memberships of PID

```
0::/grp1
```

- On a system booted in v2-only mode, there is just one line in this file (`0::...`)

Outline

Notes for online practical sessions

- Small groups in **breakout rooms**
  - Write a note into Slack if you have a preferred group
- **We will go faster, if groups collaborate** on solving the exercise(s)
  - You can **share a screen** in your room
- I will circulate regularly between rooms to answer questions
- Zoom has an "**Ask for help**" button...
- **Keep an eye on the `#general` Slack channel**
  - Perhaps with further info about exercise;
  - Or a note that the exercise merges into a break
- When your room has finished, write a message in the Slack channel: **"***** Room X has finished *****"**
  - Then I have an idea of how many people have finished

Shared screen etiquette

- It may help your colleagues if you **use a larger than normal font!**
  - In many environments (e.g., *xterm*, *VS Code*), we can adjust the font size with `Control`+`Shift`+"`+`" and `Control`+"`-`"
  - Or (e.g., *emacs*) hold down `Control` key and use mouse wheel
- **Long shell prompts** make reading your shell session difficult
  - Use `PS1='$ '` or `PS1='# '`
- **Low contrast** color themes are difficult to read; change this if you can
- Turn on **line numbering** in your editor
  - In *vim* use: `:set number`
  - In *emacs* use: `M-x display-line-numbers-mode <RETURN>`
    - `M-x` means `Left-Alt`+`x`
- For collaborative editing, **relative line-numbering is evil**....
  - In *vim* use: `:set nornu`
  - In *emacs*, the following should suffice:

```
M-: (display-line-numbers-mode) <RETURN>
M-: (setq display-line-numbers 'absolute) <RETURN>
```

    - `M-:` means `Left-Alt`+`Shift`+`:`

Using *tmate* in in-person practical sessions

In order to share an X-term session with others, do the following:

- Enter the command *tmate* in an X-term, and you'll see the following:

```
$ tmate
...
Connecting to ssh.tmate.io...
Note: clear your terminal before sharing readonly access
web session read only: ...
ssh session read only: ssh S0mErAnD0m5Tr1Ng@lon1.tmate.io
web session: ...
ssh session: ssh S0mEoTheRrAnD0m5Tr1Ng@lon1.tmate.io
```

- Share last "ssh" string with colleague(s) via Slack or another channel
  - Or: "ssh session read only" string gives others read-only access
- Your colleagues should paste that string into an X-term...
- Now, you are sharing an X-term session in which anyone can type
  - Any "mate" can cut the connection to the session with the 3-character sequence `<ENTER>` ~ .
- To see above message again: `tmate show-messages`

Exercises

1. In this exercise, we create a cgroup, place a process in the cgroup, and then migrate that process to a different cgroup.
   - Create two subdirectories, `m1` and `m2`, in the cgroup root directory (`/sys/fs/cgroup`).
   - Execute the following command, and note the PID assigned to the resulting process:

     ```
     # sleep 300 &
     ```

   - Write the PID of the process created in the previous step into the file `m1/cgroup.procs`, and verify by reading the file contents.
   - Now write the PID of the process into the file `m2/cgroup.procs`.
   - Is the PID still visible in the file `m1/cgroup.procs`? Explain.
   - Try removing cgroup `m1` using the command `rm -rf m1`. Why doesn't this work?
   - If it is still running, kill the *sleep* process and then remove the cgroups `m1` and `m2` using the *rmdir* command.

Outline

Enabling and disabling controllers

- Each cgroup v2 directory contains two files:
  - `cgroup.controllers`: lists controllers that are **available** in this cgroup
  - `cgroup.subtree_control`: used to list/modify set of controllers that are **enabled** in this cgroup
    - Always a subset of `cgroup.controllers`
- Together, these files allow different controllers to be managed to **different levels of granularity** in v2 hierarchy

Available controllers: `cgroup.controllers`

```
$ cat /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids rdma misc
```

- `cgroup.controllers` lists the controllers that are available in a cgroup
- Certain "automatic" controllers are always available in every cgroup, and are not listed in `cgroup.controllers`
  - `devices`, `freezer`, `network`, `perf_event`

Available controllers: cgroup.controllers

```
$ cat /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids rdma misc
```

- A **controller may not be available** because:
  - Controller is **not enabled in parent cgroup**
    - (Does not apply for "automatic" controllers)
  - Controller was disabled at boot time
    - Using the boot option `cgroup_disable=name[,...]`

Enabling controllers: cgroup.subtree_control

- `cgroup.subtree_control` is used to show or modify the set of controllers that are enabled in a cgroup:

```
# cd /sys/fs/cgroup/
# cat cgroup.subtree_control
cpu io memory pids
```

  - Set of controllers enabled in root cgroup will depend on distro and *systemd* configuration and version
- Contents of `cgroup.subtree_control` are always a subset of `cgroup.controllers`
  - I.e., can't enable controller that is not available in a cgroup
- Controllers are enabled/disabled by writing to this file:

```
# echo '+cpuset' > cgroup.subtree_control   # Enable a controller
# cat cgroup.subtree_control
cpuset cpu io memory pids
# echo '-cpuset' > cgroup.subtree_control   # Disable a controller
# cat cgroup.subtree_control
cpu io memory pids
```

Enabling controllers: cgroup.subtree_control

- Enabling a controller in cgroup.subtree_control:
  - Allows resource to be **controlled in child cgroups**
  - **Causes controller-specific attribute files to appear in each child directory**
- Attribute files in child cgroups are **used by process managing parent cgroup** to manage resource allocation into child cgroups

cgroup.subtree_control example

- Review situation in root cgroup:

```
# cd /sys/fs/cgroup/
# cat cgroup.controllers
cpuset cpu io memory hugetlb pids misc
# cat cgroup.subtree_control
cpu io memory pids
```

- Create a small subhierarchy:

```
# mkdir -p grp_x/grp_y
```

- Controllers available in grp_x are those that were enabled at level above; no controllers are enabled in grp_x:

```
# cat grp_x/cgroup.controllers
cpu io memory pids
# cat grp_x/cgroup.subtree_control              # Empty...
```

- Consequently, no controllers are available in grp_y:

```
# cat grp_x/grp_y/cgroup.controllers            # Empty...
```

cgroup.subtree_control example

- List cpu.* files in grp_y:

```
# cd /sys/fs/cgroup/grp_x
# ls grp_y/cpu.*
grp_y/cpu.pressure  grp_y/cpu.stat
```

  - (These two files show CPU-related statistics and are present in every cgroup)

- Enabling cpu controller in parent cgroup (grp_x) causes controller interface files to appear in child (grp_y) cgroup:

```
# echo '+cpu' > cgroup.subtree_control
# ls grp_y/cpu.*
grp_y/cpu.idle          grp_y/cpu.max.burst  grp_y/cpu.stat
grp_y/cpu.weight.nice   grp_y/cpu.max        grp_y/cpu.pressure
grp_y/cpu.weight
```


cgroup.subtree_control example

- After enabling controller in parent cgroup, we can limit resources in child cgroup...
- Set hard CPU limit of 50% in child cgroup (grp_y):

```
# echo '50000 100000' > grp_y/cpu.max
```

- In another window, we start a program that burns CPU time and displays statistics; and we move it into grp_y:

```
# echo 6445 > grp_y/cgroup.procs    # 6445 is PID of burner process
```

- In the other terminal, we see:

```
$ ./cpu_burner
[6445]  %CPU = 99.86
[6445]  %CPU = 99.83
...
[6445]  %CPU = 83.52
[6445]  %CPU = 50.00
[6445]  %CPU = 50.00
...
```

Managing controllers to differing levels of granularity

- A controller is **available in child** cgroup only if it is **enabled in parent** cgroup:

```
# cat cgroup.controllers
cpuset cpu io memory hugetlb pids
# cat cgroup.subtree_control
cpu memory pids
# cat grp1/cgroup.controllers
cpu memory pids
```

  - `cpuset`, `io`, and `hugetlb` are not available in `grp1`
- In `grp1`, none of the available controllers is initially enabled, so no controllers are available at next level:

```
# cat grp1/cgroup.controllers
cpu memory pids
# cat grp1/cgroup.subtree_control       # Empty
# mkdir grp1/{grp10,grp11}              # Make grandchild cgroups
# cat grp1/grp2/cgroup.controllers      # Empty
```


Managing controllers to differing levels of granularity

- If we enable `cpu` in `grp1`, it becomes available at next level

```
# echo '+cpu' > grp1/cgroup.subtree_control
# cat grp1/grp10/cgroup.controllers
cpu
```

  - And `cpu` interface files appear in `grp1/{grp10,grp11}`
- Here, `cpu` is being managed at finer granularity than `memory`
  - We can make distinct `cpu` allocation decisions for processes in `grp10` vs processes in `grp11`
  - But we can't make distinct `memory` allocation decisions
    - `grp10` and `grp11` will share `memory` allocation from `grp1`
- We **did this by design** (so we can manage different resources to different levels of granularity):
  - We want distinct CPU allocations in `grp10` and `grp11`
  - We want `grp10` and `grp11` to share a memory allocation

Top-down constraints

- Child cgroups are always subject to any resource constraints established in ancestor cgroups
  - ⇒ Descendant cgroups can't relax constraints imposed by ancestor cgroups
- If a controller is disabled in a cgroup (i.e., not present in `cgroup.subtree_control`), it cannot be enabled in any descendants of the cgroup

No internal tasks rule

- Cgroups v2 enforces a rule often expressed as: "a cgroup can't have both child cgroups and member processes"
  - I.e., only leaf nodes can have member processes
  - The "no internal tasks" rule
- But the rule more precisely is:
  - A cgroup can't both:
    - distribute a resource to child cgroups (i.e., enable controllers in `cgroup.subtree_control`), **and**
    - have member processes

Outline

Exercises

1. This exercise demonstrates that resource constraints apply in a top-down fashion, using the cgroups v2 `pids` controller.
   - To simplify the following steps, change your current directory to the cgroup root directory (i.e., the location where the `cgroup2` filesystem is mounted; on recent *systemd*-based systems, this will be `/sys/fs/cgroup`, or possibly `/sys/fs/cgroup/unified`).
   - Create a child and grandchild directory in the cgroup filesystem and enable the PIDs controller in the root directory and the first subdirectory:

```
# mkdir xxx
# mkdir xxx/yyy
# echo '+pids' > cgroup.subtree_control
# echo '+pids' > xxx/cgroup.subtree_control
```

[Exercise continues on next page...]

Exercises

- Set an upper limit of 10 tasks in the child cgroup, and an upper limit of 20 tasks in the grandchild cgroup:

```
# echo '10' > xxx/pids.max
# echo '20' > xxx/yyy/pids.max
```

- In another terminal, use the supplied `cgroups/fork_bomb.c` program.

```
fork_bomb <num-children> [<child-sleep>]
# Default:      0               300
```

  Run the program with the following command line, which (after the user presses *Enter*) will cause the program to create 30 children that sleep for (the default) 300 seconds:

```
$ ./fork_bomb 30
```

[Exercise continues on next page...]

Exercises

- The parent process in the `fork_bomb` program prints its PID. Return to the first terminal and place the parent process in the grandchild `pids` cgroup:

```
# echo parent-PID  > xxx/yyy/cgroup.procs
```

- In the second terminal window, press *Enter*, so that the parent process now creates the child processes. How many children does it successfully create?

Linux Security and Isolation APIs Essentials

Control Groups (cgroups): Other Controllers

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Rev: # 6f75b3d2e02f

Outline

Cgroups v2 controllers

- Initial release of cgroups v2 (Linux 4.5), did not include equivalents of all v1 controllers
- Remaining controllers were added later, with last appearing in Linux 5.6
- Documentation/admin-guide/cgroup-v2.rst documents v2 controllers

Summary of cgroups controllers

The following table summarizes some info about controllers that are provided in cgroups v1 and v2, including kernel versions where the controllers first appeared

V1 controller	Linux	V2 equivalent	Linux
cpu	2.6.24 (& 3.2)	cpu +	4.15
cpuacct	2.6.24	cpu +	4.15
cpuset	2.6.24	cpuset +	5.0
memory	2.6.25	memory	4.5
devices	2.6.26	devices *	4.15
freezer	2.6.26	freezer *	5.2
net_cls	2.6.29	network *	4.5
net_prio	3.3	network *	4.5
blkio	2.6.33	io	4.5
perf_event	2.6.39	perf_event * +	4.11
hugetlb	3.6	hugetlb	5.6
pids	4.3	pids +	4.5
rdma	4.3	rdma	4.11
n/a	-	misc	5.13

- (*) V2 "automatic" controllers (always available, not listed in `cgroup.controllers`)
- (+) V2 threaded controllers

Cgroups v2 controllers

- Each of the controllers is selectable via a **kernel configuration option**
  - And there is an overall option, `CONFIG_CGROUPS`
- For each controller, there are controller-specific files in each cgroup directory
  - Names are prefixed with controller-specific string
    - E.g., `cpu.weight`, `memory.max`, `pids.current`
- In following slides we look at a couple of example controllers

Outline

The cpu controller

cpu: control and accounting of CPU usage

- cpu.stat provides statistics on CPU used by cgroup

```
# cat mygrp/cpu.stat
usage_usec 345928360
user_usec 195880335
system_usec 150048024
...
```

  - Values (expressed in μs) include total CPU (kernel+user) time, and time broken down info kernel and user mode
  - Values are totals of time consumed by processes while they reside in cgroup
  - Statistics include CPU consumed in descendant cgroups

The cpu controller

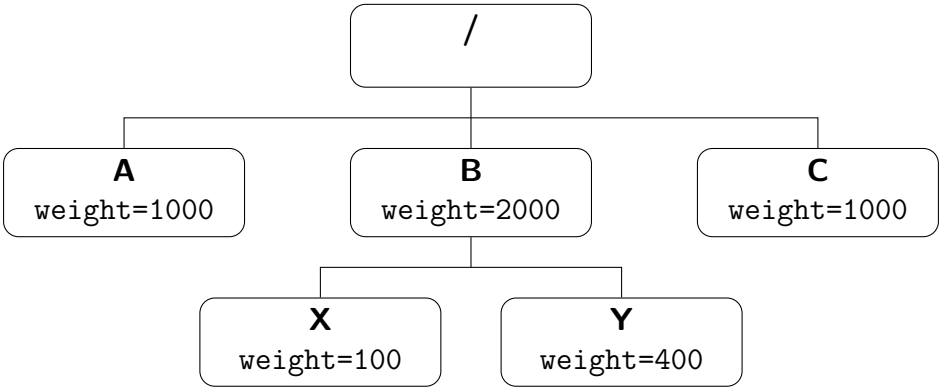
- `cpu` controller provides two modes to control distribution of CPU cycles to cgroups:
  - **Proportional-weight** mode
  - **Absolute-bandwidth** mode
- Default is proportional-weight mode
  - Absolute-bandwidth mode is used if *quota* limit is set in `cpu.max`

cpu controller: proportional-weight mode

`cpu` **proportional-weight** mode:

- `cpu.weight` file defines proportion of CPU given to cgroup
  - Default is 100; permitted range is 1..10000
  - Proportion of CPU given to cgroup defined by quotient:
    (`cpu.weight` / [sum of all `cpu.weight` at same level])

cpu controller: proportional-weight mode



- Processes in B get $\frac{2000}{1000+2000+1000} = \frac{1}{2}$ of CPU time
- Processes in A and C each get $\frac{1000}{1000+2000+1000} = \frac{1}{4}$ of CPU time
- Processes in X get $\frac{2000}{1000+2000+1000} \cdot \frac{100}{100+400} = \frac{1}{2} \cdot \frac{1}{5} = \frac{1}{10}$ of CPU time
- Processes in Y get $\frac{2000}{1000+2000+1000} \cdot \frac{400}{100+400} = \frac{1}{2} \cdot \frac{4}{5} = \frac{4}{10}$ of CPU time

cpu controller: proportional-weight mode

`cpu` **proportional-weight** mode:

- Constraints have **effect only if there is competition** for CPU
  - No effect until [# CPU-bound processes] > [# CPUs]
    - For experiments, use *taskset(1)* to constrain multiple processes to same CPU
- Constraints propagate proportionally into child cgroups
  - I.e., child cgroups further subdivide proportion given to parent cgroup

cpu controller: absolute-bandwidth mode

`cpu` **absolute-bandwidth** mode:

- Used to set absolute limit on CPU that can be consumed per defined period
- Limit is defined by writing two values to `cpu.max`:

```
echo '<quota> <period>' > cpu.max
```

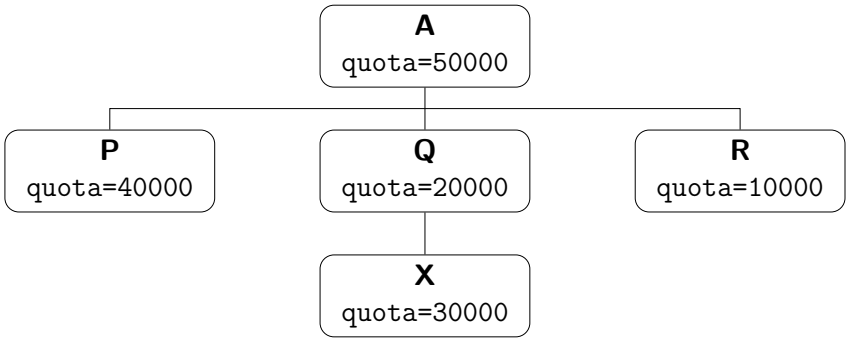
  - *period*: measurement period for CFS scheduler (microsecs; range: [1000..1'000'000]; default: 100'000)
    - Larger period means CPU is allocated in longer bursts (i.e., 1000/2000 is not same as say 50'000/100'000)
  - *quota*: allowed run-time within period (range: ≥ 1000)
    - *quota/period* **expresses fraction of one CPU**; can be > 1
    - If cgroup exhausts its quota within a given period, it is throttled until the next period
    - Default: `max` == no limit/inherit quota from parent

cpu controller: absolute-bandwidth mode

`cpu` **absolute-bandwidth** mode:

- Quota is enforced even if no other competitors for CPU
- Parent quota is a **cap** for child quota

cpu controller: absolute-bandwidth mode



- Assume that *period* is 100'000 in all cgroups
- Processes under A will get maximum of 50% of (one) CPU
- Processes under Q will get maximum of 20% of CPU
- Processes under X will get maximum of 20% of CPU (capped by Q)
- Note that sibling cgroups under A are oversubscribed (they won't get 70% of CPU)

Outline

The freezer controller

freezer: freeze (suspend) and thaw (resume) processes in a cgroup

- Cgroup is frozen/thawed by writing 1/0 to cgroup.freeze
  - Operations propagate to descendant cgroups
  - cgroup.freeze is not present in root cgroup
- Useful for container migration and checkpoint/restore
  - And, e.g., *docker pause*
- Gets around some limitations of using SIGSTOP/SIGCONT for this purpose
  - SIGSTOP is observable by waiting parent or ptracer
  - SIGCONT can be caught by application!
  - Observability of these signals can cause behavior changes in applications

Outline

Exercises

1. The `cpu` controller implements bandwidth-based throttling of CPU usage. Throttling is specified by writing a pair of numbers to `cpu.max`:

```
# echo '<quota> <period>' > cpu.max
```

   - *period*: the period used for allocating CPU bandwidth (μsec; default 100'000).
   - *quota*: the portion of the period available to this cgroup (μsec; default "`max`", meaning no limit).

   Perform the following experiments:

   - Check the `cgroup.subtree_control` file in the root cgroup to see if the `cpu` controller is enabled, and if it is not, enable it.
   - Create two sibling CPU cgroups, named `fast` and `slow`. In the `fast` cgroup, set a *quota* of 30'000 and a *period* of 100'000:

```
# echo '30000 100000' > fast/cpu.max
```

     In the `slow` cgroup, set *quota* to 10'000 and *period* to 100'000.

Exercises

- Run two instances of the `timers/cpu_burner.c` program, which consumes CPU time. The program prints a message every second that includes the percentage of CPU time it received during that second. (i.e., *CPU-time* / *elapsed-time*). Place the two instances in the different CPU cgroups, and observe the effect on the rate of execution of the two programs. What happens if you adjust the *quota* to 50'000 in the `slow` cgroup?
- Suspend the two `cpu_burner` processes using control-Z and then check how much CPU time has been consumed in each cgroup by examining the `usage_usec` field in the file `cpu.stat` in each directory. This field shows CPU usage in microseconds, which can be converted to seconds using commands such as the following:

```
$ awk '/usage_usec/ {print $2 / 1000000}' < slow/cpu.stat
$ awk '/usage_usec/ {print $2 / 1000000}' < fast/cpu.stat
```

- If you move the process in the `slow` cgroup to the `fast` cgroup, does this change the `usage_usec` value in either of the `cpu.stat` files?

Exercises

2. The `freezer` controller can be used to suspend and resume execution of all of the processes in a cgroup hierarchy. (Note that the `freezer` controller is one of the "automatic" controllers; it is always available, and doesn't need to be enabled in `cgroup.subtree_control`.)

   Create a cgroup hierarchy containing two child cgroups (thus three cgroups in total) as follows:

```
# mkdir /sys/fs/cgroup/mfz
# mkdir /sys/fs/cgroup/mfz/sub1
# mkdir /sys/fs/cgroup/mfz/sub2
```

   Then run four separate instances of the `timers/cpu_burner.c` program (in four separate terminal windows), and place two of the resulting processes in the `mfz/sub1` cgroup, and one process in each of `mfz` and `mfz/sub2`. Arrange your screen so that you can see all four terminal windows simultaneously. **Observe what happens to these processes as each of the following commands are executed**.

   Freeze the processes in the `mfz/sub1` cgroup:

```
# echo 1 > /sys/fs/cgroup/mfz/sub1/cgroup.freeze
```


Exercises

   Freeze all of the processes in all cgroups under the `mfz` subtree:

```
# echo 1 > /sys/fs/cgroup/mfz/cgroup.freeze
```

   Thaw the `mfz` subtree (which processes resume execution?):

```
# echo 0 > /sys/fs/cgroup/mfz/cgroup.freeze
```

   Once more freeze the entire `mfz` subtree, and then try thawing just the processes in the `mfz/sub1` cgroup:

```
# echo 1 > /sys/fs/cgroup/mfz/cgroup.freeze
# echo 0 > /sys/fs/cgroup/mfz/sub1/cgroup.freeze
```

   Do the processes in the `mfz/sub1` cgroup resume execution? Why not? For a clue, view the state of this cgroup using the following command:

```
# grep frozen /sys/fs/cgroup/mfz/sub1/cgroup.events
```

   Try moving one of the processes in the frozen `mfz` cgroup into the root cgroup. What happens?

   Use the `kill -KILL` command to send a `SIGKILL` signal to a process in a frozen cgroup? Is the process killed immediately? (A design bug in cgroups v1 meant that the process was not killed immediately in this scenario.)

Notes

Notes

Linux Security and Isolation APIs Essentials

Wrapup

Michael Kerrisk, man7.org © 2025

January 2025

mtk@man7.org

Outline

Rev: # 6f75b3d2e02f

Outline

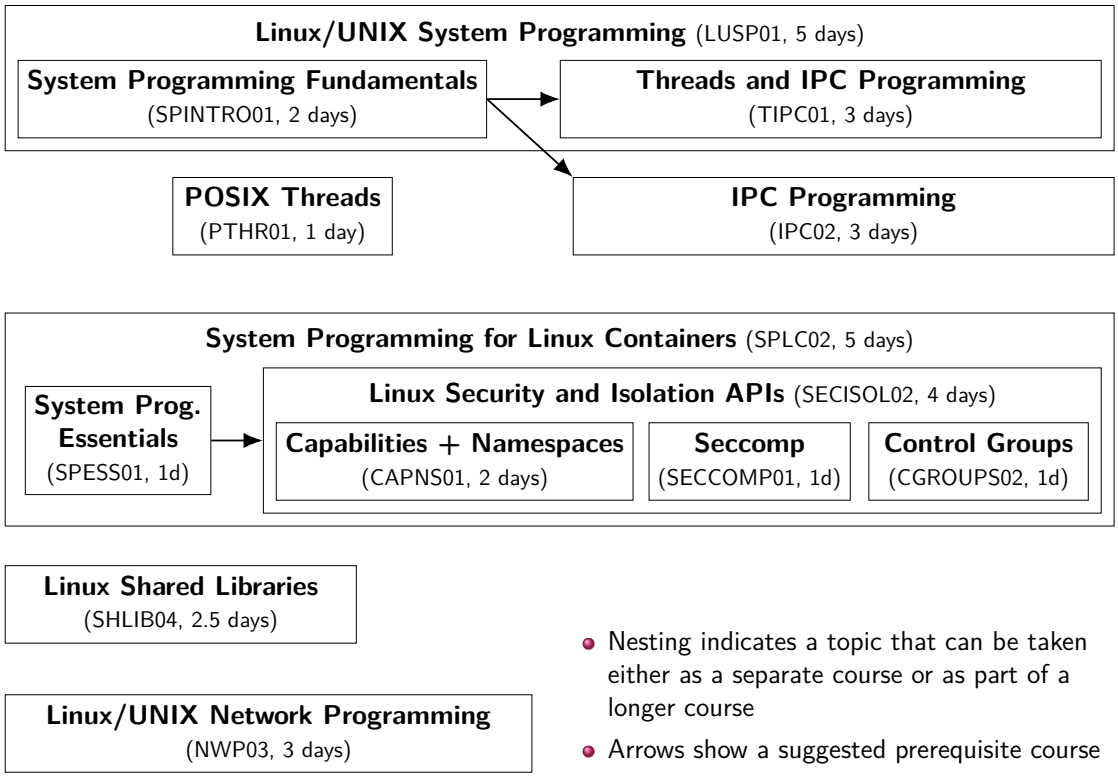
Course materials

- I'm the (sole) producer of the course book and example programs
- Course materials are continuously revised
- Send corrections and suggestions for improvements to mtk@man7.org

Marketing

- Independent trainer, consultant, and writer
  - Author of *The Linux Programming Interface*
- Reputation / word-of-mouth are important for my business...
- Let people know about these courses!
  - Linux/UNIX system programming
  - Linux security and isolation APIs
  - Building and using shared libraries
  - System programming for Linux containers
  - Linux/UNIX network programming
  - Subsets/combinations of the above; **see next slide**
  - Further courses to be announced: `http://man7.org/training/`

Course overview (see `https://man7.org/training`)



- Nesting indicates a topic that can be taken either as a separate course or as part of a longer course
- Arrows show a suggested prerequisite course

Thanks!

mtk@man7.org @mkerrisk linkedin.com/in/mkerrisk

PGP fingerprint: 4096R/3A35CE5E

`http://man7.org/training/`

Notes