Lecture Slides for Linux System Programming

Edition 0.0

Michael D. Adams

Department of Electrical and Computer Engineering
University of Victoria
Victoria, British Columbia, Canada

To obtain the most recent version of these lecture slides (with functional hyperlinks) or for additional information and resources related to these slides (including errata), please visit:

https://www.ece.uvic.ca/~mdadams/cppbook

If you like these lecture slides, **please consider posting a review** of them at:

https://play.google.com/store/search?q=ISBN:9781990707032 or
https://books.google.com/books?vid=ISBN9781990707032

youtube.com/iamcanadian1867 github.com/mdadams @mdadams16

This document was typeset with LaTeX.

ISBN 978-1-990707-03-2 (PDF)

```
   in Section 1(f) below, which, by reason of the selection and
   arrangement of their contents, constitute intellectual creations, in
   which the Work is included in its entirety in unmodified form along
   with one or more other contributions, each constituting separate and
   independent works in themselves, which together are assembled into a
   collective whole. A work that constitutes a Collection will not be
   considered an Adaptation (as defined above) for the purposes of this
   License.
c. "Distribute" means to make available to the public the original and
   copies of the Work through sale or other transfer of ownership.
d. "Licensor" means the individual, individuals, entity or entities that
   offer(s) the Work under the terms of this License.
e. "Original Author" means, in the case of a literary or artistic work,
   the individual, individuals, entity or entities who created the Work
   or if no individual or entity can be identified, the publisher; and in
   addition (i) in the case of a performance the actors, singers,
   musicians, dancers, and other persons who act, sing, deliver, declaim,
   play in, interpret or otherwise perform literary or artistic works or
   expressions of folklore; (ii) in the case of a phonogram the producer
   being the person or legal entity who first fixes the sounds of a
   performance or other sounds; and, (iii) in the case of broadcasts, the
   organization that transmits the broadcast.
f. "Work" means the literary and/or artistic work offered under the terms
   of this License including without limitation any production in the
   literary, scientific and artistic domain, whatever may be the mode or
   form of its expression including digital form, such as a book,
   pamphlet and other writing; a lecture, address, sermon or other work
   of the same nature; a dramatic or dramatico-musical work; a
   choreographic work or entertainment in dumb show; a musical
   composition with or without words; a cinematographic work to which are
   assimilated works expressed by a process analogous to cinematography;
   a work of drawing, painting, architecture, sculpture, engraving or
   lithography; a photographic work to which are assimilated works
   expressed by a process analogous to photography; a work of applied
   art; an illustration, map, plan, sketch or three-dimensional work
   relative to geography, topography, architecture or science; a
   performance; a broadcast; a phonogram; a compilation of data to the
   extent it is protected as a copyrightable work; or a work performed by
   a variety or circus performer to the extent it is not otherwise
   considered a literary or artistic work.
g. "You" means an individual or entity exercising rights under this
```

License III

```
   License who has not previously violated the terms of this License with
   respect to the Work, or who has received express permission from the
   Licensor to exercise rights under this License despite a previous
   violation.
h. "Publicly Perform" means to perform public recitations of the Work and
   to communicate to the public those public recitations, by any means or
   process, including by wire or wireless means or public digital
   performances; to make available to the public Works in such a way that
   members of the public may access these Works from a place and at a
   place individually chosen by them; to perform the Work to the public
   by any means or process and the communication to the public of the
   performances of the Work, including by public digital performance; to
   broadcast and rebroadcast the Work by any means including signs,
   sounds or images.
i. "Reproduce" means to make copies of the Work by any means including
   without limitation by sound or visual recordings and the right of
   fixation and reproducing fixations of the Work, including storage of a
   protected performance or phonogram in digital form or other electronic
   medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce,
limit, or restrict any uses free from copyright or rights arising from
limitations or exceptions that are provided for in connection with the
copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License,
Licensor hereby grants You a worldwide, royalty-free, non-exclusive,
perpetual (for the duration of the applicable copyright) license to
exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more
   Collections, and to Reproduce the Work as incorporated in the
   Collections; and,
b. to Distribute and Publicly Perform the Work including as incorporated
   in Collections.

The above rights may be exercised in all media and formats whether now
known or hereafter devised. The above rights include the right to make
such modifications as are technically necessary to exercise the rights in
other media and formats, but otherwise you have no rights to make
Adaptations. Subject to 8(f), all rights not expressly granted by Licensor
```

```
       under Section 4(b).
 e. Except as otherwise agreed in writing by the Licensor or as may be
    otherwise permitted by applicable law, if You Reproduce, Distribute or
    Publicly Perform the Work either by itself or as part of any
    Collections, You must not distort, mutilate, modify or take other
    derogatory action in relation to the Work which would be prejudicial
    to the Original Author's honor or reputation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR
OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY
KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE,
INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY,
FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF
LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS,
WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION
OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE
LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR
ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES
ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

 a. This License and the rights granted hereunder will terminate
    automatically upon any breach by You of the terms of this License.
    Individuals or entities who have received Collections from You under
    this License, however, will not have their licenses terminated
    provided such individuals or entities remain in full compliance with
    those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any
    termination of this License.
 b. Subject to the above terms and conditions, the license granted here is
    perpetual (for the duration of the applicable copyright in the Work).
    Notwithstanding the above, Licensor reserves the right to release the
    Work under different license terms or to stop distributing the Work at
    any time; provided, however that any such election will not serve to
    withdraw this License (or any other license that has been, or is
    required to be, granted under the terms of this License), and this
```

License VIII

```
Creative Commons is not a party to this License, and makes no warranty
whatsoever in connection with the Work. Creative Commons will not be
liable to You or any party on any legal theory for any damages
whatsoever, including without limitation any general, special,
incidental or consequential damages arising in connection to this
license. Notwithstanding the foregoing two (2) sentences, if Creative
Commons has expressly identified itself as the Licensor hereunder, it
shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the
Work is licensed under the CCPL, Creative Commons does not authorize
the use by either party of the trademark "Creative Commons" or any
related trademark or logo of Creative Commons without the prior
written consent of Creative Commons. Any permitted use will be in
compliance with Creative Commons' then-current trademark usage
guidelines, as may be published on its website or otherwise made
available upon request from time to time. For the avoidance of doubt,
this trademark restriction does not form part of this License.

Creative Commons may be contacted at http://creativecommons.org/.
```

Other Textbooks and Lecture Slides by the Author I

1. M. D. Adams, *Exercises for Programming in C++ (Version 2021-04-01)*, Apr. 2021, ISBN 978-0-9879197-5-5 (PDF). Available from Google Books, Google Play Books, and author's web site https://www.ece.uvic.ca/~mdadams/cppbook.
2. M. D. Adams, *Lecture Slides for Programming in C++ (Version 2021-04-01)*, Apr. 2021, ISBN 978-0-9879197-4-8 (PDF). Available from Google Books, Google Play Books, and author's web site https://www.ece.uvic.ca/~mdadams/cppbook.
3. M. D. Adams, *Multiresolution Signal and Geometry Processing: Filter Banks, Wavelets, and Subdivision (Version 2013-09-26)*, University of Victoria, Victoria, BC, Canada, Sept. 2013, ISBN 978-1-55058-507-0 (print), ISBN 978-1-55058-508-7 (PDF). Available from Google Books, Google Play Books, and author's web site https://www.ece.uvic.ca/~mdadams/waveletbook.

Other Textbooks and Lecture Slides by the Author II

4. M. D. Adams, *Lecture Slides for Multiresolution Signal and Geometry Processing (Version 2015-02-03)*, University of Victoria, Victoria, BC, Canada, Feb. 2015, ISBN 978-1-55058-535-3 (print), ISBN 978-1-55058-536-0 (PDF). Available from Google Books, Google Play Books, and author's web site https://www.ece.uvic.ca/~mdadams/waveletbook.
5. M. D. Adams, *Signals and Systems*, Edition 5.0, Dec. 2022, ISBN 978-1-990707-00-1 (PDF). Available from Google Books, Google Play Books, and author's web site https://www.ece.uvic.ca/~mdadams/sigsysbook.
6. M. D. Adams, *Lecture Slides for Signals and Systems*, Edition 5.0, Dec. 2022, ISBN 978-1-990707-02-5 (PDF). Available from Google Books, Google Play Books, and author's web site https://www.ece.uvic.ca/~mdadams/sigsysbook.

7. M. D. Adams, *Lecture Slides for the Clang Libraries*, Edition 0.0, Dec. 2022, ISBN 978-1-990707-04-9 (PDF). Available from Google Books, Google Play Books, and author's web site https://www.ece.uvic.ca/~mdadams/cppbook.

Part 0

Preface

About These Lecture Slides

- This document constitutes a set of lecture slides that covers various aspects of system programming in Linux.
- This document represents a work in progress and should be considered an ***alpha release***.
- In spite of this, it is believed that this document will be of benefit to some people. So, it is being made available in its current form.
- This document is intended to supplement the following slide deck:
  - M. D. Adams, *Lecture Slides for Programming in C++ (Version 2021-04-01)*, Apr. 2021, ISBN 978-0-9879197-4-8 (PDF). Available from Google Books, Google Play Books, and author's web site `https://www.ece.uvic.ca/~mdadams/cppbook`.

Typesetting Conventions

- In a definition, the term being defined is often typeset in a font **like this**.
- To emphasize particular words, the words are typeset in a font ***like this***.
- To show that particular text is associated with a hyperlink to an internal target, the text is typeset like this.
- To show that particular text is associated with a hyperlink to an external document, the text is typeset like this.
- URLs are typeset like `https://www.ece.uvic.ca/~mdadams`.

Companion Git Repository

- These lecture slides have a companion Git repository.
- Numerous code examples are available from this repository.
- This repository is hosted by GitHub.
- The URL of the main repository page on GitHub is:
  - `https://github.com/mdadams/linux_companion`
- The URL of the actual repository itself is:
  - `https://github.com/mdadams/linux_companion.git`

Part 1

Introduction

Linux

- open-source Unix-like operating system
- originally developed by Linus Torvalds in 1991
- monolithic kernel
- some additional functionality can be loaded as system running (e.g., some device drivers)
- has been ported to many hardware architectures (e.g., x86-64, ARM, RISC-V, Itanium, PowerPC, S390, and S390X)
- in 2000, Linux Foundation, non-profit technology consortium founded to standardize Linux, support its growth, and promote its commercial adoption
- some popular non-commercial Linux distributions include:
  - Ubuntu, Debian, Fedora, Gentoo, OpenSUSE
- some popular commercial Linux distributions include:
  - Red Hat Enterprise Linux (RHEL) and SUSE Linux Enterprise Server (SLES)

Single UNIX Specification (SUS)

- Single UNIX Specification (SUS) is collective name of family of standards for operating systems
- compliance with standard required to qualify for use of UNIX trademark
- core specification of SUS developed and maintained by Austin Group, which is joint working group of IEEE, ISO JTC 1 SC22, and The Open Group
- Austin Group web site: `https://www.opengroup.org/austin/`
- specifies API for system calls and library functions
- specifies utilities and shell

Portable Operating System (POSIX) Standard

- joint ISO/IEC/IEEE standard:
  - ISO/IEC/IEEE 9945:2009 — information technology — Portable Operating System Interface (POSIX) base specifications, issue 7, 2009 [3807 pages].
- intended to promote compatibility between variants of Unix and other operating systems
- defines application programming interface (API)
- specifies command-line shells and utility interfaces
- some variants of Unix have been certified as POSIX compliant, including (amongst others):
  - MacOS, HP-UX, and AIX
- many variants of Unix are mostly POSIX compliant:
  - Linux, DragonFly BSD, FreeBSD, NetBSD, OpenBSD

Linux Standard Base (LSB) Standard

- joint ISO IEC standard, consisting of several parts
- core specification:
  - ISO/IEC 23360-1:2006 — Linux Standard Base (LSB) core specification 3.1 — part 1: Generic specification, 2006 [458 pages].
- specification for AMD64 (x86-64) architecture:
  - ISO/IEC 23360-4:2006 — Linux Standard Base (LSB) core specification 3.1 — part 4: Specification for AMD64 architecture, 2006.
- joint project by several Linux distributions under organizational structure of Linux Foundation (`https://www.linuxfoundation.org`)
- defines application programming interface (API) and application binary interface (ABI)
- standardize software system structure, including filesystem hierarchy
- standard covers several architectures, including:
  - AMD64 (x86-64), IA32, IA64, PPC32, PPC64, S390, and S390X
- LSB based on POSIX specification, Single UNIX Specification (SUS), and several other open standards

Part 2

Main Topics

Section 2.1

Preliminaries

System Calls

- **system call**: mechanism by which program requests service from operating system
- for example, system calls used to:
  - open, close, read, and write files
  - create and terminate processes
- typically, system call invoked by special CPU instruction
- most system calls return integral type (e.g., **`int`** or **`long`** or type alias for such type)
- if system call fails, global variable `errno` set with reason for failure

Some errno Values

Signal	Description
EACCES	permission denied
EAGAIN	resource temporarily unavailable
EBADF	bad file descriptor
EBUSY	device or resource busy
EDQUOT	disk quota exceeded
EEXIST	file exists
EFAULT	bad address
EFBIG	file too large
EINTR	interrupted function call (due to signal)
EINVAL	invalid argument
EIO	I/O error
ELOOP	too many levels of symbolic links
ENODEV	no such device
ENOENT	no such file or directory
ENOSPC	no space left on device (e.g., file system full)
EPERM	operation not permitted

Querying System Configuration

- various parameters of system configuration can be queried using `sysconf` system call
- declaration:

```
long sysconf(int name);
```

- upon success, returns value of parameter specified by `name`
- upon failure (e.g., due to invalid value for `name`), returns -1
- some values for `name` include those shown in following table:

`name`	Description
`_SC_ARG_MAX`	maximum length of arguments to exec family of system calls
`_SC_HOST_NAME_MAX`	maximum length of host name
`_SC_LOGIN_NAME_MAX`	maximum length of login name
`_SC_OPEN_MAX`	maximum number of open files per process
`_SC_NGROUPS_MAX`	maximum number of supplementary GIDs

sysconf_1.cpp

```
#include <format>
#include <iostream>
#include <unistd.h>

int main() {
    std::cout << std::format(
      "POSIX version: {}\n"
      "maximum login name length: {}\n"
      "maximum host name length: {}\n"
      "maximum length of arguments to exec: {}\n"
      "maximum number of open files: {}\n"
      "page size in bytes: {}\n",
      sysconf(_SC_VERSION),
      sysconf(_SC_LOGIN_NAME_MAX),
      sysconf(_SC_HOST_NAME_MAX),
      sysconf(_SC_ARG_MAX),
      sysconf(_SC_OPEN_MAX),
      sysconf(_SC_PAGESIZE));
}
```

- program prints several system-configuration parameters

User Identification

- each user identified by unique **user ID (UID)**, which is nonnegative integer
- mapping of usernames to UIDs provided by system password file `/etc/passwd`
- each user can belong to one or more groups
- in particular, each user belongs to one ***primary group*** and zero or more ***supplementary groups***
- each group identified by unique **group ID (GID)**, which is nonnegative integer
- mapping of groupnames to GIDs provided by system group file `/etc/group`
- UID 0 is reserved for ***superuser*** (i.e., system administrator)
- UIDs from 0 to 99 for static use by system
- UIDs from 100 to 499 for dynamic use by system
- UIDs from 500 upwards for normal users

[Example] Password File /etc/passwd

/etc/passwd

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
nobody:x:65534:65534:Kernel Overflow User:/:/sbin/nologin
jsmith:x:1000:1000::/home/jsmith:/bin/bash
```

- `root` user has UID 0, primary GID 0, home directory `/`, and login shell `/bin/bash`
- `jsmith` user has UID 1000, primary GID 1000, home directory `/home/jsmith`, and login shell `/bin/bash`

[Example] Group File `/etc/group`

`/etc/group`

```
root:x:0:
bin:x:1:
daemon:x:2:
sys:x:3:
adm:x:4:
tty:x:5:
disk:x:6:
lp:x:7:
mem:x:8:
kmem:x:9:
wheel:x:10:jsmith
users:x:100:jsmith
jsmith:x:1000:
```

- `wheel` group has GID 0 and is supplementary group for user `jsmith`
- `jsmith` group has GID 1000 and is supplementary group for no users

Processes

- each process has unique **process ID (PID)**, which is strictly positive integer
- processes have parent-child relationships (i.e., child process created by its parent process)
- **init process** is first process started as last step in system boot
- init process is at root of parent-child tree
- **parent process ID (PPID)** is PID of parent process
- since each process must always have parent, process that is orphaned due to its parent terminating must be assigned new parent
- **reaper/subreaper process**: process that automatically adopts orphaned processes
- orphaned process always reparented to nearest still-living ancestor reaper/subreaper
- process can be made subreaper via `prctl` system call
- init process is reaper and systemd process is typically subreaper

Process Groups and Sessions

- related processes can be grouped using process group
- each process belongs to exactly one process group
- each process group named by unique **process group ID (PGID)**, which is nonnegative integer
- can signal all processes in process group at once via `kill` system call
- process groups can be grouped into sessions
- each session named by unique **session ID (SID)**, which is nonnegative integer
- often, session used to track all processes belonging to particular user-login instance

Querying PID, PPID, PGID, and SID

- can query PID with `getpid` system call, which has declaration:

```
pid_t getpid();
```

- can query PPID with `getppid` system call, which has declaration:

```
pid_t getppid();
```

- process can query its process group using `getpgrp` system call, which has declaration:

```
pid_t getpgrp();
```

- `getpid`, `getppid`, and `getpgrp` system calls cannot fail
- process can query session of another process or itself using `getsid` system call, which has declaration:

```
sid_t getsid(pid_t pid);
```

- `getsid` returns SID of process specified by `pid`
- upon failure, returns -1; otherwise returns requested SID
- if `pid` is 0, `getsid` returns SID of calling process

Process Information from Shell and `ps` Program

- `ps` command can be used to print information about processes/threads running on system
- example output from `ps` command:

```
  PID TTY          TIME CMD
  771 pts/1    00:00:01 bash
15062 pts/1    00:00:00 ps
```

- in Bash shell, `$$` gives PID of shell
- output PID, PPID, PGID, command and arguments for all processes, sorted by PGID:

```
ps -e -o pid,ppid,pgid,comm,args | sort -k 3 -n
```

[Example] Querying PID, PPID, PGID, and SID

get_process_ids.cpp

```
#include <format>
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

int main() {
    std::cout << std::format(
      "PID: {}; "
      "parent PID: {}; "
      "process group ID: {}; "
      "session ID: {}\n",
      getpid(), getppid(), getpgrp(), getsid(0));
}
```

- program prints its PID, PPID, process group ID, and session ID

Use of Process Groups By Shell

- shell must be capable of running many programs as child processes, possibly with many running concurrently
- single command line often requires running several programs concurrently
- for convenience in managing processes that shell runs, typically process groups used to collect together related processes
- consider several different ways of running `get_process_ids` program from previous slide:

```
./get_process_ids
./get_process_ids; ./get_process_ids
./get_process_ids 3>&1 1>&2- 2>&3- | ./get_process_ids
(./get_process_ids; ./get_process_ids)
./get_process_ids 2> >( ./get_process_ids)
```

Real and Effective Users and Groups

- each process deemed belong to particular user and group, referred to as **real user** and **real group**, respectively
- real user and real group identified by **real UID (RUID)** and **real GID (RGID)**, respectively
- for purposes of authorization checking (e.g., file permission checks), however, process treated as if it belongs to particular user and group, referred to as **effective user** and **effective group**, respectively
- effective user and effective group identified by **effective UID (EUID)** and **effective GID (EGID)**, respectively
- normally, real and effective users are same and real and effective groups are same, but they need not be
- effective user/group typically set differently from real user/group in situations where one wants to have more or less privileges than normal

User and Group IDs for New Processes

- child process inherits from parent process:
  - RUID and RGID
  - EUID and EGID
  - supplementary groups
- login process sets user/group IDs of login shell as follows:
  - RUID and RGID set according to information in system password file
  - supplementary groups set according to information in system group file
  - EUID set to RUID
  - EGID set to RGID

Querying User and Group Information from Command Line

- `id` command can be used to query user and group information
- example output for `id` command:

```
uid=1000(jsmith) gid=1000(jsmith) groups=1000(jsmith)
    ↪ ,10(wheel),100(users)
```

Querying Real and Effective UIDs and GIDs

- process can get its real UID using `getuid` system call, which has declaration:

```
uid_t getuid();
```

- process can get its real GID using `getgid` system call, which has declaration:

```
gid_t getgid();
```

- process can get its effective UID using `geteuid` system call, which has declaration:

```
uid_t geteuid();
```

- process can get its effective GID using `getegid` system call, which has declaration:

```
gid_t getegid();
```

- `getuid`, `getgid`, `geteuid`, and `getegid` system calls can never fail

Querying Supplementary GIDs

- process can get its supplementary GIDs using `getgroups` system call, which has declaration:

```
int getgroups(int size, gid_t* list);
```

- upon success, returns supplementary GIDs of calling process in array pointed to by `list`
- `size` is maximum number of elements that array pointed to by `list` can accommodate
- if `size` is not large enough to hold all supplementary GIDs for process, error results
- upon success, returns number of GIDs placed in array
- upon failure, returns -1
- can query maximum number of supplementary groups to which any process can belong via `sysconf` system call

get_user_group_ids.cpp

```
#include <format>
#include <iostream>
#include <vector>
#include <sys/types.h>
#include <unistd.h>

int main() {
    std::cout << std::format(
      "real UID: {}\n"
      "real GID: {}\n"
      "effective UID: {}\n"
      "effective GID: {}\n",
      getuid(), getgid(), geteuid(), getegid());
    auto max_groups = sysconf(_SC_NGROUPS_MAX);
    std::vector<gid_t> groups(max_groups);
    if (int num_groups = getgroups(groups.size(), groups.data());
      num_groups > 0) {
        groups.resize(num_groups);
        std::cout << "supplementary GIDs:";
        for (auto gid : groups) {std::cout << std::format(" {}", gid);}
        std::cout << '\n';
    }
}
```

- program prints real and effective UIDs and GIDs for process as well as supplementary GIDs

Environment

- each process has its own environment, which is collection of variables called environment variables
- some programs use values of certain environment variables to control their behavior
- `PATH` environment variable controls where shells and some library functions looks for executable programs
- normally, environment propagated from parent process to child
- `printenv` command prints environment variables and their values
- example of `printenv` output:

```
PWD=/home/jdoe
LOGNAME=jdoe
HOME=/home/jdoe
USER=jdoe
SHLVL=1
PATH=/usr/local/sbin:/usr/local/bin:/usr/local/games:/usr/sbin:/
    ↪ usr/bin:/usr/games:/sbin:/bin
```

Section 2.2

File I/O

File Systems

- file system views name of file and file contents/metadata as distinct
- file contents plus file metadata constitute what is called **inode**
- in addition to possible data for file, inode also has metadata, including:
  - file type
  - file mode (permissions and a few other attributes of file)
  - link count (number of names referencing inode)
  - UID of user who owns file
  - GID of group that owns file
  - size of file in bytes
  - last access time (atime)
  - file creation time (btime)
  - last modification time (mtime)
  - last status change time (ctime)

File Systems (Continued)

- file types:
  - regular file
  - directory
  - symbolic link
  - block device (e.g., disk)
  - character device (e.g., terminal)
  - socket (i.e., UNIX-domain socket)
  - FIFO (named pipe)

File Permissions

- three categories of user permissions:
  - user permissions: apply when process UID matches UID of file owner
  - group permissions: apply when user permissions do not apply; and GID of process matches GID of file owner
  - other permissions: apply when user and group permissions do not apply
- for file:
  - read permission required to read contents of file
  - write permission required to modify contents of file
  - execute permission required to run program stored in file
- for directory:
  - read permission required to inspect contents of directory
  - write permission required to add, remove, or rename file/directory in directory
  - execute permission is required to access (for any purpose) subdirectories of directory

Set-UID and Set-GID Bits

- meaning of each of set-UID and set-GID bits depends on file type (i.e., regular file versus directory)
- for (regular) file:
  - if set-UID bit is set and user execute bit is set, program to be run with EUID set to UID of file owner
  - if set-GID bit and group execute bit are set, program to be run with EGID set to GID of group owner of file
  - if set-GID bit is set and group execute bit is not set, file uses mandatory file/record locking
- for directory:
  - set-UID bit usually ignored by most Unix and Linux systems
  - if set-GID bit is set, files created in directory inherit their GID from directory not from EGID of creating process and any created directory will also have set-GID bit set
- set-UID and set-GID bits can be employed to allow users to run programs with escalated privileges
- for example, `sudo` program has set-UID bit set

Sticky Bits

- meaning of sticky bit depends on file type (i.e., regular file versus directory)
- for directory: if sticky bit set, only file’s owner, directory’s owner, or root can rename or delete file in directory
- for (regular) file: sticky bit ignored by Linux and most other modern Unix variants
- in practice, sticky bit often set on system temporary directories (such as `/tmp` or `/var/tmp`) so that one user cannot delete temporary files of another user

Querying and Setting File Permissions From Command Line

- can query ownership and permissions of file using `stat` and `ls` commands
- example output for `ls -l /bin/mkdir`:

```
-rwxr-xr-x 1 root root 182816 May 3 2019 /bin/mkdir
```

- example output for `stat /bin/mkdir`:

```
  File: /bin/mkdir
  Size: 182816    Blocks: 360        IO Block: 4096 regular file
Device: 31h/49d Inode: 1094    Links: 1
Access: (0755/-rwxr-xr-x) Uid: ( 0/   root)  Gid: (   0/    root)
Access: 2019-11-13 10:50:26.254653816 -0800
Modify: 2019-05-03 07:41:27.000000000 -0700
Change: 2019-09-25 07:48:09.381732326 -0700
 Birth: 2019-05-14 18:02:48.406664793 -0700
```

- can set permissions of file (including set-UID, set-GID, and sticky bits) using `chmod` command
- can change user/group ownership of file using `chown` command

Files

- files often identified through identifier known as **file descriptor**, which is nonnegative integer
- file descriptors only have meaning in context of single process
- many operations involving files take file descriptors as parameters
- by convention, three file descriptors have special meaning:

Value	Description
0	standard input
1	standard output
2	standard error

- system has upper bound on number of file descriptors that can be in use by single process at any given time (which can be queried by `sysconf` system call)

Opening File

- can open file using `open`, `creat`, and `openat` system calls
- most commonly used function is `open`
- declaration:

```
int open(const char* pathname, int oflags, mode_t mode);
```

- `pathname` is pathname of file/directory to open
- `oflags` is flags used to control open operations (see following slides for list of flags)
- `mode` is mode bits for file being created when `O_CREAT` or `O_TMPFILE` flags specified (see following slides for list of modes)
- `creat(fd, mode)` equivalent to `open(fd, O_CREAT | O_WRONLY | O_TRUNC, mode)`
- `openat` takes additional parameter relative to `open` which provides file descriptor corresponding to directory to be used for interpreting relative pathnames

Open Flags

Flag	Description
O_APPEND	always append to file
O_ASYNC	enable signal-driven I/O
O_CLOEXEC	enable close-on-exec flag for file descriptor
O_CREAT	create file if it does not exist
O_DIRECT	bypass cache
O_DIRECTORY	fail if not directory
O_DSYNC	write operations use synchronized I/O data integrity completion
O_EXCL	ensure that call creates file
O_LARGEFILE	allow large files to be opened
O_NOATIME	do not update file last access time when file read
O_NOCTTY	do not become controlling terminal
O_NOFOLLOW	do not follow symbolic links
O_NONBLOCK	enable nonblocking I/O
O_NDELAY	same as O_NONBLOCK
O_PATH	open path only
O_SYNC	write operations use synchronized I/O file integrity completion
O_TMPFILE	create unnamed temporary regular file
O_TRUNC	truncate file

Open Modes

Flag	Description
S_IRWXU	user has read, write, and execute permission
S_IRUSR	user has read permission
S_IWUSR	user has write permission
S_IXUSR	user has execute permission
S_IRWXG	group has read, write, and execute permission
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	others have read, write, and execute permission
S_IROTH	others have read permission
S_IWOTH	others have write permission
S_IXOTH	others have execute permission
S_ISUID	set-UID bit
S_ISGID	set-GID bit
S_ISVTX	sticky bit

Closing File

- file is closed with `close` system call
- declaration:

```
int close(int fd);
```

- upon success, returns 0
- upon failure (e.g., due to invalid file descriptor), returns -1

[Example] Opening and Closing Files

file_3.cpp

```
#include <iostream>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd = open("/etc/passwd", O_RDONLY);
    if (fd < 0) {
        std::cerr << "open failed\n";
        return 1;
    }
    // ... (use fd)
    close(fd); // note: we might forget to close
}
```

- code structure like that above not recommended since prone to resource leaks (i.e., leaking open file descriptors)
- better to use RAII class to hold file descriptor

`unique_handle` Class Template

- `unique_handle` class template holds opaque handle to resource (such as open file, capability information, etc.) following unique-ownership model
- similar to `std::unique_ptr` except can hold any resource, not only pointer to memory
- declaration:

```
template <class Policy> class unique_handle;
```

- `unique_handle` object can hold non-null or null handle
- non-null handle is object that refers to some resource that must be freed
- null handle is dummy handle that does not refer to any resource
- template parameter `Policy` is class specifying:
  - type of handle
  - function for freeing resource associated with non-null handle
  - function for testing if handle is null
  - function that returns null handle
- if, when destructor invoked, non-null handle held by object, underlying resource freed

unique_handle.hpp

```
#include <utility>

template<typename Policy>
class unique_handle {
public:
    using handle_type = typename Policy::handle_type;
    unique_handle() : h_(Policy::get_null()) {}
    unique_handle(handle_type handle) : h_(handle) {}
    unique_handle(unique_handle&& other) noexcept
      {h_ = other.h_; other.h_ = Policy::get_null();}
    unique_handle& operator=(unique_handle&& other) noexcept
      {reset(); h_ = other.h_; other.h_ = Policy::get_null(); return *this;}
    unique_handle(const unique_handle&) = delete;
    unique_handle& operator=(unique_handle&) = delete;
    ~unique_handle() {reset();}
    handle_type get() const {return h_;}
    explicit operator bool() const {return !Policy::is_null(h_);}
    void reset(handle_type new_handle = Policy::get_null()) {
        handle_type old_handle = h_;
        h_ = new_handle;
        if(!Policy::is_null(old_handle)) {Policy::free(old_handle);}
    }
    void swap(unique_handle& other)
      {using std::swap; swap(h_, other.h_);}
private:
    handle_type h_;
};
```

unique_fd Class

unique_fd.hpp

```
#include "unique_handle.hpp"

struct fd_uh_policy {
    using handle_type = int;
    static void free(handle_type h) {close(h);}
    static bool is_null(handle_type h) {return h < 0;}
    static handle_type get_null() {return -1;}
};

using unique_fd = unique_handle<fd_uh_policy>;
```

- unique_fd class used to manage file descriptors following unique-ownership model
- unique_fd class utilizes unique_handle class template introduced earlier
- if open file descriptor associated with unique_fd object when destructor invoked, destructor automatically closes file descriptor

unique_fd Class: Usage Example

unique_fd_1.cpp

```
#include <string>
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include "unique_fd.hpp"

int do_work() {
    unique_fd fd(open("/dev/null", O_WRONLY));
    if (!fd) {return 1;}
    std::string text("Hello, World!\n");
    if (write(fd.get(), text.data(), text.size()) != text.size()) {
        // NOTE: no need to close file descriptor here
        return 2;
    }
    // NOTE: no need to close file descriptor here
    return 0;
} // NOTE: destruction of fd will close file descriptor (if open)

int main() {return do_work();}
```

- no open file descriptors leaked in code shown above despite fact that code does not ***explicitly*** close any file descriptors
- close of open file descriptor implicitly performed by destructor of unique_fd class

[Example] Opening and Closing Files Revised: Summary

- use class template to rewrite earlier example application in manner that greatly reduces chance of leaking file-descriptors
- explicit close of file descriptor no longer needed (unless early close of file desired)
- object that owns file descriptor closes file descriptor upon destruction

[Example] Opening and Closing Files Revisited: Code

file_1.cpp

```
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include "unique_fd.hpp"

int main() {
    unique_fd fd(open("/etc/passwd", O_RDONLY));
    if (!fd) {
        std::cerr << "open failed\n";
        return 1;
    }
    // ... (use fd)
    /* when destructor for fd invoked, any open file descriptor
       associated with fd is closed */
}
```

Reading from File

- data can be read from file using `read` system call
- declaration:

```
ssize_t read(int fd, void *buf, size_t count);
```

- `fd`: file descriptor specifying file from which to read
- `buf`: pointer to start of buffer in which to place data to be read
- `count`: number of bytes of data to read
- data is read starting from current position in file
- upon failure, -1 is returned
- upon success, number of bytes read is returned, which ***may be less than*** `count` (e.g., due to end of file, fewer bytes being available when reading from pipe or terminal, or ***interrupted system call***)
- if read takes place at end of file, 0 is returned

Writing to File

- data can be written to file using `write` system call
- declaration:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- `fd`: file descriptor specifying file to which to write
- `buf`: pointer to buffer holding data to be written
- `count`: number of bytes of data to written
- data is written starting from current position in file
- upon failure, -1 is returned
- upon success, number of bytes written is returned, which ***may be less than*** `count` (e.g., due no space left on disk or ***interrupted system call***)

[Example] Copying Files: Summary

- program copies file from source to destination
- pathname for source and destination files specified as command-line arguments
- code retries interrupted read/write system calls
- note that not accounting for interrupted system calls is common source of bugs

copy_file_1.cpp

```
#include <iostream>
#include <vector>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include "unique_fd.hpp"

ssize_t read_with_retry(int fd, void* buf, ssize_t count) {
    ssize_t n;
    while ((n = read(fd, buf, count)) < 0 && errno == EINTR) {}
    return n;
}

ssize_t write_with_retry(int fd, const void* buf, ssize_t count) {
    ssize_t n;
    while ((n = write(fd, buf, count)) < 0 && errno == EINTR) {}
    return n;
}

ssize_t write_all(int fd, const void* buf, ssize_t count) {
    const char* start = static_cast<const char*>(buf);
    ssize_t remaining = count;
    do {
        ssize_t n = write_with_retry(fd, start, remaining);
        if (n <= 0) {return -1;}
        remaining -= n;
        start += n;
    } while (remaining > 0);
    return count;
}
```

copy_file_1.cpp (Continued)

```
int copy(int source_fd, int destination_fd) {
    std::vector<char> buffer(64 * 1024);
    for (;;) {
        ssize_t n = read_with_retry(source_fd, buffer.data(), buffer.size());
        if (n < 0) {return -1;}
        else if (!n) {break;}
        if (write_all(destination_fd, buffer.data(), n) != n) {return -1;}
    }
    return 0;
}

int main(int argc, char** argv) {
    if (argc < 3) {std::cerr << "invalid usage\n";}
    unique_fd source_fd(open(argv[1], O_RDONLY));
    if (!source_fd)
        {std::cerr << "cannot open source file\n"; return 1;}
    unique_fd destination_fd(open(argv[2], O_CREAT | O_TRUNC | O_WRONLY,
      S_IRWXU));
    if (!destination_fd)
        {std::cerr << "cannot open destination file\n"; return 1;}
    if (copy(source_fd.get(), destination_fd.get()))
        {std::cerr << "copy failed\n"; return 1;}
}
```

Seeking in File

- current position in file can be changed with `lseek` system call
- declaration:

```
off_t lseek(int fd, off_t offset, int whence);
```

- adjusts current location in file using `offset` and `whence` as follows:

whence	Meaning
`SEEK_SET`	offset set to `offset` bytes
`SEEK_CUR`	offset set to current location plus `offset` bytes
`SEEK_END`	offset set of size of file plus `offset` bytes

- upon success, returns current location measured in bytes from start of file
- upon failure, returns -1
- some types of entities to which file descriptor might refer do not allow seek operation (e.g., pipe, FIFO, or socket)

[Example] Reading and Writing File

file_2.cpp

```
#include <iostream>
#include <string>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include "unique_fd.hpp"

int main() {
    std::string buf{"Hello, World!\n"};
    unique_fd fd(open("/tmp/foo", O_CREAT | O_TRUNC |
      O_WRONLY, S_IRWXU));
    if (!fd) {std::cout << "open failed\n";}
    if (lseek(fd.get(), 1, SEEK_SET) < 0) {std::cerr << "lseek failed\n";}
    if (write(fd.get(), &buf[1], buf.size() - 1) != buf.size() - 1)
      {std::cerr << "write failed\n";}
    if (lseek(fd.get(), 0, SEEK_SET) < 0) {std::cerr << "lseek failed\n";}
    if (write(fd.get(), &buf[0], 1) != 1) {std::cerr << "write failed\n";};
}
```

- program writes `"Hello, World!\n"` to file nonsequentially in two parts, with help of seek operation

Vectored I/O

- **vectored I/O**: ***single*** I/O operation transfers data between data stream and ***multiple*** buffers in memory
- **scatter read**: reads data from single data stream and sequentially writes this data to multiple buffers in memory
- **gather write**: sequentially reads data from multiple buffers in memory and writes this data to single data stream
- advantages of vectored I/O:
  1. efficiency: fewer I/O operations required, since ***single*** read/write operation can transfer data between file and ***multiple*** buffers in memory
  2. atomicity: no risk of interleaving with I/O operations from other processes/threads
  3. convenience: eliminates need to write code to copy data into and out of single contiguous buffer

Specifying Multiple Buffers for Vectored I/O

- `iovec` used to specify single buffer for vectored I/O
- array of `iovec` objects used to specify multiple buffers
- declaration:

```
struct iovec {
    void* iov_base; // start address of buffer
    size_t iov_len; // size of buffer in bytes
};
```

Scatter Read from File

- scatter read (i.e., read into multiple buffers) can be performed by `readv` system call
- declaration:

```
ssize_t readv(int fd, const iovec *iov, int iovcnt);
```

- `fd`: file descriptor specifying file from which to read
- `iov`: pointer to array of I/O vectors, where each I/O vector is description of buffer
- `iovcnt`: number of elements in I/O vector array
- `iovec` struct describes single buffer by specifying start address of buffer and its size in bytes
- data is read starting from current position in file
- upon failure, -1 is returned
- upon success, number of bytes read is returned, which ***may be less than total number of bytes requested to be transferred*** (for similar reasons as in case of `read` system call)

Gather Write from File

- gather write (i.e., write from multiple buffers) can be performed by `writev` system call
- declaration:

```
ssize_t writev(int fd, const iovec *iov, int iovcnt);
```

- `fd`: file descriptor specifying file to which to write
- `iov`: pointer to array of I/O vectors, where each I/O vector is description of buffer
- `iovcnt`: number of elements in I/O vector array
- `iovec` struct describes single buffer by specifying start address of buffer and its size in bytes
- data is written starting from current position in file
- upon failure, -1 is returned
- upon success, number of bytes written is returned, which ***may be less than total number of bytes requested to be transferred*** (for similar reasons as in case of `write` system call)

[Example] Vectored I/O

vectored_io_1_a.cpp

```
#include <array>
#include <cstring>
#include <iostream>
#include <string>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

int main() {
    std::string hello("Hello!");
    char bonjour[] = "Bonjour!";
    char newline = '\n';
    std::array<iovec, 4> iov{{
        {hello.data(), hello.size()}, {&newline, 1},
        {bonjour, std::strlen(bonjour)}, {&newline, 1},
    }};
    /* note: successfull write operation may not necessarily
      write all of data */
    if (writev(1, &iov[0], iov.size()) < 0)
        {std::cerr << "write failed\n";}
}
```

- program writes `"Hello!\nBonjour!\n"` to standard output using single write operation, where data to be written split across multiple buffers

[Example] Vectored I/O: Summary

- code in previous example did not correctly handle case of `writev` being interrupted (when output partially written)
- on next slide, we consider code that correctly handles this case

[Example] Vectored I/O: Code

vectored_io_1_b.cpp

```
#include <array>
#include <cstring>
#include <iostream>
#include <string>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

ssize_t writev_all(int fd, iovec* iov, size_t count) {
    ssize_t written = 0;
    for (;;) {
        ssize_t n;
        while ((n = writev(fd, iov, count)) < 0 && errno == EINTR) {}
        if (n < 0) {return -1;}
        else if (!n) {return written;}
        written += n;
        for (; count > 0 && n >= iov->iov_len; ++iov, --count) {n -= iov->iov_len;}
        if (!count) {return written;}
        iov->iov_base = static_cast<char*>(iov->iov_base) + n;
        iov->iov_len -= n;
    }
}

int main() {
    std::string hello("Hello!");
    char bonjour[] = "Bonjour!";
    char newline = '\n';
    std::array<iovec, 4> iov{{
        {hello.data(), hello.size()}, {&newline, 1},
        {bonjour, std::strlen(bonjour)}, {&newline, 1},
    }};
    if (writev_all(1, &iov[0], iov.size()) < 0) {std::cerr << "write failed\n";}
}
```

Duplicating File Descriptors

- can duplicate file descriptor with another file descriptor
- `dup` duplicates file descriptor using lowest-numbered unused file descriptor as new descriptor; declaration:

```
int dup(int oldfd);
```

- `dup2` duplicates file descriptor at specified file descriptor, closing it first if open; declaration:

```
int dup2(int oldfd, int newfd);
```

- `dup3` provides functionality that is essentially superset of `dup2`; declaration:

```
int dup3(int oldfd, int newfd, int flags);
```

- `dup3` similar as `dup2`, except `flags` can be used to specify close-on-exec flag (and `oldfd` and `newfd` cannot be equal)
- duplicating file descriptor typically used after fork but before exec in order to handle I/O redirection

[Example] I/O Redirection: Summary

- program runs another specified program as child process
- child process has standard input and standard output redirected from/to specified files
- program has following command-line arguments (in order):
  1. pathname of file to associate with standard input of program to be run
  2. pathname of file to associate with standard output of program to be run
  3. pathname of program to be run as child process
  4. zero or more additional arguments to be passed as command-line arguments to program to be run
- for example, to run command "`/bin/ls -al /`" with standard input read from `/dev/null` and standard output written to `/tmp/output`, use command:

```
io_redirection_1 /dev/null /tmp/output /bin/ls -al /
```

[Example] I/O Redirection: Code

io_redirection_1.cpp

```
#include <cassert>
#include <format>
#include <iostream>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char** argv) {
    if (argc < 4) {std::cerr << "invalid usage\n"; std::exit(1);}
    int stdin_fd = open(argv[1], O_RDONLY);
    if (stdin_fd < 0) {std::cerr << "cannot open input\n"; return 1;}
    int stdout_fd = open(argv[2], O_CREAT | O_TRUNC | O_WRONLY, S_IRUSR | S_IWUSR);
    if (stdin_fd < 0) {std::cerr << "cannot open output\n"; return 1;}
    if (pid_t child_pid = fork(); child_pid > 0) {
        int status;
        if (wait(&status) < 0) {std::cerr << "wait failed\n";}
        std::cout << std::format("child exit status {}\n",
          (WIFEXITED(status) ? WEXITSTATUS(status) : -1));
    } else if (child_pid == 0) {
        close(0);
        dup2(stdin_fd, 0);
        close(1);
        dup2(stdout_fd, 1);
        if (execve(argv[3], &argv[3], environ) < 0)
          {std::cerr << "exec failed\n"; std::exit(1);}
    } else {std::cerr << "fork failed\n"; std::exit(1);}
}
```

Pipes

- **pipe** is form of inter-process communication mechanism
- can think of it like pipe in plumbing sense
- process at each end of pipe
- data flows in one direction through pipe from process at sending end to process at receiving end
- named pipe is associated with FIFO file in file system
- unnamed pipe is not associated with any file in file system (i.e., is essentially FIFO buffer internal to operating system)

Creating Pipes

- unnamed pipe can be created with `pipe` system call
- `pipe` system call returns pair of file descriptors that refer to sending and receiving ends of pipe
- declaration:

```
int pipe(int pipefd[2]);
```

- `pipefd`: pointer to array of two file descriptors
- `pipefd[0]` is receiving end of pipe (i.e., end from which data read)
- `pipefd[1]` is sending end of pipe (i.e., end to which data written)

[Example] I/O Redirection With Pipeline: Summary

- program takes two command line arguments
- each of first and second argument is pathname of program to be run
- use fork and exec to create processes that run specified two programs
- standard output of first program redirected to standard input of second program
- pipe system call to used generate unnamed pipe

pipe_1.cpp

```
#include <iostream>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

template <class... Ts> void panic(const Ts&... args)
  {(std::cerr << ... << args) << '\n'; std::exit(255);}
```

[Example] I/O Redirection With Pipeline: Code (2)

`pipe_1.cpp` (Continued)

```
int main(int argc, char** argv) {
    if (argc != 3) {panic("invalid usage");}
    pid_t pids[2];
    int pipe_fds[2];
    if (pipe(pipe_fds)) {panic("pipe failed");}
    pids[0] = fork();
    if (pids[0] == 0) {
        if (close(pipe_fds[0])) {panic("close failed");}
        if (pipe_fds[1] != 1) {
            if (dup2(pipe_fds[1], 1) != 1) {panic("dup2 failed");}
            if (close(pipe_fds[1])) {panic("close failed");}
        }
        char *args[] = {argv[1], nullptr};
        if (execve(argv[1], args, environ)) {panic("exec failed");}
    } else if (pids[0] < 0) {panic("fork failed");}
    if (close(pipe_fds[1])) {panic("close failed");}
    pids[1] = fork();
    if (pids[1] == 0) {
        if (pipe_fds[0] != 0) {
            if (dup2(pipe_fds[0], 0) != 0) {panic("dup2 failed");}
            if (close(pipe_fds[0])) {panic("close failed");}
        }
        char *args[] = {argv[2], nullptr};
        if (execve(argv[2], args, environ)) {panic("exec failed");}
        return 1;
    } else if (pids[1] < 0) {panic("fork failed");}
    if (close(pipe_fds[0])) {panic("closed failed");}
    int status;
    if (waitpid(pids[0], &status, 0) < 0) {std::cerr << "wait failed\n";}
    if (waitpid(pids[1], &status, 0) < 0) {std::cerr << "wait failed\n";}
    return WIFEXITED(status) ? WEXITSTATUS(status) : -1;
}
```

Section 2.3

Sockets

Local and Network Communication

- any communication requires two endpoints
- messages or data that originates at one endpoint is transferred to other endpoint
- endpoints may be on same machine (i.e., local communication) or on different machines (i.e., network communication)
- protocol provides mechanism for communicating
- protocol has several key defining characteristics:
  - is it connection-oriented or connectionless?
  - does it provide reliable data transmission (i.e., can data be lost)?
  - does it provide sequencing (i.e., data always arrives in order sent)?
  - is it datagram-based or stream-based (i.e., is data partitioned into messages or just single stream of bytes)?
- connection-oriented analogous to telephone call; only need to specify recipient when first establishing connection
- connectionless analogous to letter in postal mail; must specify recipient each time data to be transferred

Sockets

- socket is end point for communication
- communication may be either local (i.e., endpoints on same machine) or across network (i.e., endpoints on different machines)
- underlying communication mechanism used by socket depends on its:
  - protocol domain (i.e., family of protocols used by socket)
  - type of communication functionality (e.g., connection-oriented versus connectionless and reliable versus unreliable)
  - specific protocol in domain, if more than one supported particular type of functionality
- Unix-domain sockets can only be used for local communication and are similar to message queues or pipes
- IP-domain sockets can be used for network or local communication and employ TCP and IP

Some Address Families

Family	Description
`AF_UNIX`	local communication
`AF_INET`	IPv4 Internet protocols (e.g., TCP, IP)
`AF_INET6`	IPv6 Internet protocols (e.g., TCP, IP)

Socket Types

Type	Description
SOCK_STREAM	provides sequenced, reliable, two-way connection-based byte streams
SOCK_DGRAM	supports datagrams (i.e., connectionless, unreliable messages of fixed maximum length)
SOCK_SEQPACKET	provides sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length
SOCK_RAW	provides raw network protocol access
SOCK_RDM	provides reliable datagram layer that does not guarantee ordering

Creating a Socket

- socket can be created with `socket` system call, which has declaration

```
int socket(int domain, int type, int protocol);
```

- returns file descriptor for created socket
- socket uses protocol from protocol family `domain` that has functionality specified by `type`
- in cases where protocol family has more than one protocol that supports functionality specified by `type`, `protocol` used to identify which one of these protocols to use

Socket Pair

- can create pair of connected sockets
- socket pair similar to pipe with following main differences:
  - socket pair constitutes bidirectional communication channel, whereas pipe is unidirectional
  - sockets can be stream or datagram oriented, whereas pipes always stream oriented
  - can send credentials and rights through socket pair using ancillary messages like `SCM_RIGHTS` and `SCM_CREDENTIALS`, which is not possible with pipes

Creating a Socket Pair

- socket pair created with `socketpair` system call
- declaration:

```
int socketpair(int domain, int type, int protocol, int
↪ sv[2]);
```

- `domain` domain of connection (e.g., `AF_UNIX`, `AF_INET`, and `AF_INET6`)
- `type` type of connection (e.g., `SOCK_STREAM` and `SOCK_DGRAM`)
- `protocol` protocol for connection for when more than one protocol is supported
- `sv`: pointer to array of two file descriptors
- herein, only consider case where `domain` is `AF_UNIX` and `type` is `SOCK_STREAM`

Message Header for sendmsg/recvmsg

- msghdr data structure:

```
struct msghdr {
    void        *msg_name;      // optional address
    socklen_t    msg_namelen;   // size of address
    struct iovec *msg_iov;      // scatter/gather array
    size_t       msg_iovlen;    // # elements in msg_iov
    void        *msg_control;   // ancillary data, see below
    size_t       msg_controllen; // ancillary data buffer len
    int          msg_flags;     // flags on received message
};
```

- message has data part and control part
- msg_iov and msg_iovlen specify data part using vectored I/O buffer
- msg_control and msg_controllen specify control part of message
- each of data part and control part can have zero length (if not used)
- advisable to use length of at least one (otherwise, may be difficult to distinguish message with no data from end of file)

Sending Message Over Socket

- can send message over socket using `sendmsg` system call
- declaration:

```
ssize_t sendmsg(int sockfd, const msghdr *msg, int flags);
```

- `sockfd`: file descriptor of socket to which to send message
- `msg`: pointer to `msghdr` data structure for message to send
- `flags`: flags that control behavior of send operation
- upon success, number of bytes sent is returned
- upon failure, -1 is returned

Receiving Message Over Socket

- can receive message over socket using `recvmsg` system call
- declaration:

```
ssize_t recvmsg(int sockfd, msghdr *msg, int flags);
```

- `sockfd`: file descriptor of socket from which to receive message
- `msg`: pointer to `msghdr` data structure for message to be received
- `flags`: flags that control behavior of receive operation
- upon success, number of bytes received is returned
- upon failure, -1 is returned

Passing File Descriptors Via Sockets

- sometimes need arises to pass descriptors between processes with arbitrary relationships
- can pass file descriptors between processes using Unix-domain sockets and `sendmsg` and `recvmsg` system calls

[Example] Passing File Descriptors Via Sockets: Summary

- program creates unnamed Unix-domain socket pair that is used to receive file descriptor from child process created by forking
- child process opens file `/etc/passwd` and then passes file descriptor to parent process through socket connection
- parent then uses file descriptor to output contents of associated file to standard output

passing_descriptors_1.cpp

```
#include <iostream>
#include <stdexcept>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>

int copy(int source_fd, int destination_fd) {
    char buf[512];
    for (;;) {
        int n = read(source_fd, buf, sizeof(buf));
        if (n < 0) {return -1;}
        else if (n == 0) {break;}
        if (write(destination_fd, buf, n) != n) {return -1;}
    }
    return 0;
}
```

[Example] Passing File Descriptors Via Sockets: Code (2)

`passing_descriptors_1.cpp` (Continued)

```
int send_fd(int sd, int fd) {
    struct msghdr msg = {0};
    char cbuf[CMSG_SPACE(sizeof(int))]{};
    char dbuf[1]{};
    struct iovec iov;
    iov.iov_base = dbuf;
    iov.iov_len = sizeof(dbuf);
    msg.msg_iov = &iov;
    msg.msg_iovlen = 1;
    msg.msg_control = cbuf;
    msg.msg_controllen = sizeof(cbuf);
    msg.msg_name = nullptr;
    msg.msg_namelen = 0;
    struct cmsghdr* cmsg = CMSG_FIRSTHDR(&msg);
    cmsg->cmsg_level = SOL_SOCKET;
    cmsg->cmsg_type = SCM_RIGHTS;
    cmsg->cmsg_len = CMSG_LEN(sizeof(int));
    memcpy(CMSG_DATA(cmsg), &fd, sizeof(int));
    return (sendmsg(sd, &msg, 0) < 0) ? -1 : 0;
}
```

passing_descriptors_1.cpp (Continued)

```
int receive_fd(int sd) {
    struct msghdr msg = {0};
    char dbuf[1];
    struct iovec iov;
    iov.iov_base = dbuf;
    iov.iov_len = sizeof(dbuf);
    msg.msg_iov = &iov;
    msg.msg_iovlen = 1;
    char cbuf[256];
    msg.msg_control = cbuf;
    msg.msg_controllen = sizeof(cbuf);
    msg.msg_name = nullptr;
    msg.msg_namelen = 0;
    ssize_t n = recvmsg(sd, &msg, 0);
    if (n < 0) {return -1;}
    struct cmsghdr* cmptr = CMSG_FIRSTHDR(&msg);
    if (!cmptr) {return -2;}
    if (cmptr->cmsg_len != CMSG_LEN(sizeof(int))) {return -10;}
    if (cmptr->cmsg_level != SOL_SOCKET) {return -3;}
    if (cmptr->cmsg_type != SCM_RIGHTS) {return -4;}
    int fd;
    memcpy(&fd, CMSG_DATA(cmptr), sizeof(int));
    return fd;
}
```

passing_descriptors_1.cpp (Continued)

```
void child(int sd) {
    int fd = open("/etc/passwd", O_RDONLY);
    if (fd < 0) {throw std::runtime_error("[child] open failed");}
    if (send_fd(sd, fd))
      {throw std::runtime_error("[child] sending FD failed");}
}

void parent(int sd) {
    int fd = receive_fd(sd);
    if (fd < 0) {throw std::runtime_error("recv_fd failed");}
    if (copy(fd, 1)) {throw std::runtime_error("copy failed");}
}

int main(int argc, char** argv) try {
    int sock_fds[2];
    if (socketpair(AF_UNIX, SOCK_STREAM, 0, sock_fds)) {
        throw std::runtime_error("socketpair failed");}
    pid_t pid = fork();
    if (pid > 0) {
        close(sock_fds[1]);
        parent(sock_fds[0]);
    } else if (pid == 0) {
        close(sock_fds[0]);
        child(sock_fds[1]);
    } else {throw std::runtime_error("fork failed");}
} catch (std::exception& e) {std::cerr << e.what() << '\n';}
```

Binding a Socket to an Address

- socket can be bound to address with `bind` system call, which has declaration

```
int bind(int sockfd, const struct sockaddr *addr,
↪ socklen_t addrlen);
```

- `sockaddr` type is struct that starts with `sa_family_t` specifying address family
- number of remaining bytes and their contents depend on address family
- bind operation used to associate address with socket communication endpoint

Listening for Incoming Connections on a Socket

- can cause incoming connections for address associated with socket to be queued for later acceptance with `listen` system call, which has declaration

```
int listen(int sockfd, int backlog);
```

- can only be used for connection-oriented protocols (since there is no notion of connection in connectionless protocol)

Accepting an Incoming Connection on Socket

- can accept incoming connection with `accept` system call, which has declaration

```
int accept(int sockfd, struct sockaddr *addr, socklen_t
↪ *addrlen);
```

- can only be used for connection-oriented protocols (since there is no notion of connection in connectionless protocol)

Unix-Domain Sockets

- Unix-domain sockets provide means to communicate local to machine
- used to efficiently communicate between processes on same machine
- Unix-domain sockets have address family `AF_UNIX` (also `AF_LOCAL`)
- Unix-domain sockets can be unnamed, bound to pathname in file system, or bound to abstract name (Linux only)
- three socket types:
  1. stream sockets (`SOCK_STREAM`), which are connection oriented and do not preserve message boundaries
  2. datagram sockets (`SOCK_DGRAM`), which are connectionless and preserve message boundaries
  3. sequenced-packet sockets (`SOCK_SEQPACKET`), which are connection oriented and preserve message boundaries
- all socket types provide reliable in-order delivery (including datagram)
- Unix-domain sockets support passing of file descriptors and process credentials (and SELinux security contexts) to other processes using ancillary data
- `SOCK_SEQPACKET` preserves message boundaries when used with `sendmsg`/`recvmsg`

[Example] Datagram Server/Client: Summary

- two programs: server and client
- server loops accepting requests from client until special shutdown message received
- server creates socket and then binds it to agreed-upon address in order to receive messages sent to this address
- server then loops receiving messages sent to agreed-upon address
- bind operation creates socket-type file in filesystem
- server: `socket` ⟶ `bind` ⟶ `recvfrom`
- client creates socket and then uses it to send messages to above agreed-upon address
- client: `socket` ⟶ `sendto`
- since Unix-domain sockets always provide reliable in-order transmission of data, do not have to worry about complications caused by loss/reordering of data

[Example] Datagram Server/Client: Server Code

dgram_server.cpp

```
#include <cstring>
#include <format>
#include <iostream>
#include <errno.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/un.h>
#include <unistd.h>

int main(int argc, char** argv) {
    std::string pathname("/tmp/socket_demo");
    if (argc >= 2) {pathname = argv[1];}
    int fd = socket(AF_UNIX, SOCK_DGRAM, 0);
    struct sockaddr_un addr;
    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, pathname.c_str(), sizeof(addr.sun_path) - 1);
    if (bind(fd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
      {std::cerr << "bind failed\n"; return 1;}
    constexpr int bufsize = 256;
    char buf[bufsize];
    for (;;) {
        int ret;
        socklen_t addr_len = sizeof(sockaddr_un);
        if ((ret = recvfrom(fd, buf, bufsize, 0, (struct sockaddr*) &addr,
          &addr_len)) < 0) {std::cerr << "recvfrom failed\n"; return 1;}
        buf[ret] = '\0';
        if (ret == 0) {break;}
        if (!strcmp(buf, "end")) {break;}
        std::cout << std::format("{} {} {}\n", addr_len, ret, buf);
    }
    if (unlink(pathname.c_str()) == -1 && errno != ENOENT)
      {std::cerr << "unlink failed\n"; return 1;}
}
```

dgram_client.cpp

```
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <string>
#include <errno.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/un.h>
#include <unistd.h>

int main(int argc, char** argv) {
    std::string pathname("/tmp/socket_demo");
    if (argc >= 2) {pathname = argv[1];}
    int fd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (fd < 0) {std::cerr << "socket failed\n"; return 1;}
    std::string message;
    while (std::cin >> message) {
        struct sockaddr_un addr;
        memset(&addr, 0, sizeof(struct sockaddr_un));
        addr.sun_family = AF_UNIX;
        strncpy(addr.sun_path, pathname.c_str(), sizeof(addr.sun_path) - 1);
        int ret;
        if ((ret = sendto(fd, message.c_str(), message.size(), 0,
          (struct sockaddr*) &addr, sizeof(struct sockaddr_un))) < 0)
          {std::cerr << "sendto failed\n"; return 1;}
    }
    close(fd);
}
```

[Example] Stream Server/Client: Summary

- two programs: server and client
- server loops accepting requests from client until zero-length message received or error occurs
- client loops reading word from standard input and then sending word to server
- server creates socket, binds it to agreed-upon address, asks that incoming connections be queued, and then loops accepting connections from queue
- bind operation creates socket-type file in filesystem
- server: `socket` $\longrightarrow$ `bind` $\longrightarrow$ `listen` $\longrightarrow$ `accept` $\longrightarrow$ `recv`
- client connects to server using above agreed-upon address
- client creates socket, connects socket to agreed-upon address, and sends message over established connection
- client: `socket` $\longrightarrow$ `connect` $\longrightarrow$ `send`

[Example] Stream Server/Client: Server Code

stream_server.cpp

```
#include <cstring>
#include <format>
#include <iostream>
#include <errno.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/un.h>
#include <unistd.h>

int main(int argc, char** argv) {
    std::string pathname("/tmp/socket_demo");
    if (argc >= 2) {pathname = argv[1];}
    int sfd = socket(AF_UNIX, SOCK_SEQPACKET, 0);
    struct sockaddr_un addr;
    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, pathname.c_str(), sizeof(addr.sun_path) - 1);
    if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
      {std::cerr << "bind failed\n"; return 1;}
    constexpr int bufsize = 256;
    char buf[bufsize];
    if (listen(sfd, 1)) {std::cerr << "listen failed\n"; return 1;}
    socklen_t addr_len = sizeof(sockaddr_un);
    int fd = accept(sfd, (struct sockaddr*) &addr, &addr_len);
    for (;;) {
        int ret;
        if ((ret = recv(fd, buf, bufsize, 0)) < 0)
          {std::cerr << "recv failed\n"; return 1;}
        buf[ret] = '\0';
        if (ret == 0) {break;}
        std::cout << std::format("{} {} {}\n", addr_len, ret, buf);
    }
    close(fd);
    if (unlink(pathname.c_str()) == -1 && errno != ENOENT)
      {std::cerr << "unlink failed\n"; return 1;}
}
```

[Example] Stream Server/Client: Client Code

stream_client.cpp

```
#include <iostream>
#include <string>
#include <cstring>
#include <errno.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/un.h>
#include <unistd.h>

int main(int argc, char** argv) {
    std::string pathname("/tmp/socket_demo");
    if (argc >= 2) {pathname = argv[1];}
    int fd = socket(AF_UNIX, SOCK_SEQPACKET, 0);
    if (fd < 0) {std::cerr << "socket failed\n"; return 1;}
    struct sockaddr_un addr;
    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, pathname.c_str(), sizeof(addr.sun_path) - 1);
    if (connect(fd, (struct sockaddr*) &addr, sizeof(struct sockaddr_un)) < 0)
      {std::cerr << "connect failed\n"; return 1;}
    std::string message;
    while (std::cin >> message) {
        int ret;
        if ((ret = send(fd, message.c_str(), message.size(), 0)) < 0 ||
          ret != message.size())
          {std::cerr << "send failed\n"; return 1;}
    }
    close(fd);
}
```

Section 2.4

Signals

Signals

- signals are very limited form of inter-process communication (IPC)
- signal is asynchronous notification sent to process or specific thread to notify that event occurred
- signals similar to interrupts, main difference being that interrupts mediated by processor and handled by kernel, whereas signals mediated by kernel and handled by processes
- process/thread can register signal handler (i.e., function that handles signals)
- can register signal handler for each signal
- if signal received, but no handler, process terminated
- `SIGKILL` signal cannot be caught and results in process being terminated
- system calls that can potentially block calling thread are typically interruptible by signal
- system call that fails as result of being interrupted indicates `EINTR` as reason for failure

Some Common Signals

Signal	Description
`SIGINT`	interrupt (e.g., ctrl-c)
`SIGQUIT`	quit (e.g., ctrl-backslash)
`SIGABORT`	abort (e.g., `std::abort`
`SIGKILL`	kill (cannot be caught)
`SIGTERM`	terminate
`SIGSEGV`	segmentation violation (e.g., invalid address)
`SIGBUS`	bus error (e.g., incorrectly aligned memory access)
`SIGFPE`	floating-point exception
`SIGSTOP`	stop (i.e., temporarily suspend execution)

Sending Signal to Process

- can send signal to another process via `kill` system call
- declaration:

```
int kill(pid_t pid, int sig);
```

- sends signal `sig` to one or more processes
- if `pid` greater than 0, signal sent to process with PID `pid`
- if `pid` is -1, signal sent to every process for which calling process has permission to send signals (except process with PID 1)
- if `pid` is less than -1, signal sent to all processes in process group with PGID `-pid`
- can only send signals to processes associated with same user; otherwise special privileges required

[Example] Forcing Termination of Child Process: Summary

- program creates new child process via fork then sleeps for several seconds
- child process produces output for full duration of execution
- when parent process wakes from sleep, terminates child process with `SIGKILL` signal

[Example] Forcing Termination of Child Process: Code

kill_1.cpp

```
#include <cassert>
#include <format>
#include <iostream>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>

int main() {
    pid_t parent_pid = getpid();
    if (pid_t child_pid = fork(); child_pid > 0) {
        // parent
        sleep(5);
        kill(child_pid, SIGKILL);
        int status;
        if (waitpid(child_pid, &status, 0) > 0) {
            if (WIFSIGNALED(status)) {
                std::cout << std::format("child terminated by signal {}\n",
                  WTERMSIG(status));
            }
        }
    } else if (child_pid == 0) {
        // child
        while (true) {std::cout << '.' << std::flush; sleep(1);}
    } else {
        std::cout << "fork failed\n";
    }
}
```

Signal Handling

- can register handler for signal with `signal` function
- declaration:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

- registers function `handler` as handler for signal `signum`
- upon success, returns previous signal handler value
- upon error, returns `SIG_ERR`
- due to potential for race conditions (e.g., data races), signal handler very limited in how it can access global state
- signal handler can safely access (lock-free) atomic global variable

Suspending Thread for Period of Time

- can temporarily suspend execution of thread for specified period of time using `nanosleep` system call
- declaration:

```
int nanosleep(const timespec *req, timespec *rem);
```

- suspends execution of calling thread until at least time specified by `*req` has elapsed
- upon success, returns 0
- upon failure, returns -1
- if call interrupted by signal handler, returns -1 and sets `errno` to `EINTR`; also, if `rem` is nonnull, writes time remaining into `*rem`
- time duration specified by `timespec` type:

```
struct timespec {
    time_t tv_sec; // seconds
    long tv_nsec; // nanoseconds
};
```

[Example] Signal Handling: Summary

- program registers signal handler for `SIGINT` signal
- signal handler increments global (atomic) counter
- program enters infinite loop
- repeatedly sleeps for fixed interval using `nanosleep`
- upon awaking from sleep, prints value of counter and then resets it
- if sleep interrupted due to signal, sleep operation restarted with sleep duration adjusted to compensate for time already spent sleeping

[Example] Signal Handling: Code

signal_1.cpp

```
#include <atomic>
#include <format>
#include <iostream>
#include <signal.h>
#include <time.h>
#include <unistd.h>

std::atomic<unsigned long> counter;

void handler(int sig_no) {++counter;}

int sleep_with_retry(int seconds) {
    timespec t = {.tv_sec = seconds, .tv_nsec = 0};
    int ret;
    while ((ret = nanosleep(&t, &t)) < 0 && errno == EINTR) {}
    if (ret < 0) {return -1;}
    return 0;
}

int main() {
    if (signal(SIGINT, handler) == SIG_ERR)
        {std::cerr << "signal failed\n"; return 1;}
    for (unsigned long i = 0;; ++i) {
        counter = 0;
        if (int ret = sleep_with_retry(5); ret < 0) {return 1;}
        std::cout << std::format("\nsignal count: {}",
          static_cast<unsigned long>(counter));
    }
}
```

Remarks on Signal Handlers

- for `SIGTRAP`, saved instruction pointer refers to instruction following one that generated exception
- for `SIGSEGV`, `SIGBUS`, `SIGILL`, saved instruction pointer refers to instruction that generated exception

[Example] Breakpoint: Summary

- example of using signal handler for `SIGTRAP` signals in order to handle breakpoint instructions
- registers signal handler for `SIGTRAP` signal
- deliberately places breakpoint instructions in code
- then forces breakpoint instructions to be executed (which result in `SIGTRAP` signals being raised)
- signal handler increments global counters to track number of signals received and how many were due to encountering breakpoint instructions
- information about counters are printed at various points during code execution
- assumes x86-64 architecture

[Example] Breakpoint: Code (1)

breakpoint_lib.hpp

```
#include <sys/ucontext.h>

extern "C" void breakpoint(int);

inline int breakpoint_type(void* ip) {
    auto p = static_cast<unsigned char*>(ip);
    if (p[-1] == 0xcc) {return 0;}
    else if (p[-1] == 0x03 && p[-2] == 0xcd) {return 1;}
    else {return -1;}
}

inline void* get_ip(void* context) {
    ucontext_t* ucontext = static_cast<ucontext_t*>(context);
    return reinterpret_cast<void*>(ucontext->uc_mcontext.gregs[REG_RIP]);
}
```

breakpoint_x86.s

```
    .text
    .globl breakpoint
# void breakpoint(int type)
breakpoint:
    test %edi, %edi
    jnz .L0
    int3 # 1-byte breakpoint opcode (0xcc)
    jmp .L1
.L0: # 2-byte breakpoint opcode
    .byte 0xcd, 0x03
.L1:
    ret
```

[Example] Breakpoint: Code (2)

breakpoint_main.cpp

```
#include <atomic>
#include <format>
#include <iostream>
#include <signal.h>
#include <unistd.h>
#include "breakpoint_lib.hpp"

std::atomic<int> sigtrap_count(0);
std::atomic<int> break_count(0);

void sigtrap_handler(int sig_no, siginfo_t* siginfo, void* context) {
    if (breakpoint_type(get_ip(context)) >= 0) {++break_count;}
    ++sigtrap_count;
}

void print_stats(const char* s) {
    std::cout << std::format("{}: {}/{}\n", s, static_cast<int>(break_count),
      static_cast<int>(sigtrap_count));
}

int main(int argc, char** argv) {
    struct sigaction sa;
    sa.sa_sigaction = &sigtrap_handler;
    sigfillset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    if (sigaction(SIGTRAP, &sa, 0)) {abort();}
    print_stats("initial values");
    raise(SIGTRAP); print_stats("after sending SIGTRAP");
    breakpoint(0); print_stats("after 1-byte breakpoint");
    breakpoint(1); print_stats("after 2-byte breakpoint");
}
```

[Example] SIGILL: Summary

- example of using signal handler for `SIGILL` signals in order to handle illegal instructions
- registers signal handler for `SIGILL` signal
- deliberately places illegal instructions (using several distinct opcodes) in code
- then forces illegal instructions to be executed (which result in `SIGILL` signals being raised)
- signal handler increments counter and adjusts instruction pointer to skip over remainder of illegal opcode
- value of counter printed at several points during program execution
- assumes x86-64 architecture

[Example] SIGILL: Code (1)

sigill_lib.hpp

```
#include <sys/ucontext.h>

extern "C" void illegal_instruction(int);

inline void* get_ip(void* context) {
    ucontext_t* ucontext = static_cast<ucontext_t*>(context);
    return reinterpret_cast<void*>(ucontext->uc_mcontext.gregs[REG_RIP]);
}

inline void set_ip(void* context, void* ip) {
    ucontext_t* ucontext = static_cast<ucontext_t*>(context);
    ucontext->uc_mcontext.gregs[REG_RIP] = reinterpret_cast<unsigned long>(ip);
}

inline int get_ins_length(void* ip) {
    auto p = static_cast<unsigned char*>(ip);
    if (p[0] == 0xf && p[1] == 0xb) {return 2;}
    else if (p[0] == 0x48 && p[1] == 0x0f) {
        if ((p[2] == 0xff && p[3] == 0xc0) || (p[2] == 0xb9 && p[3] == 0xc0))
          {return 4;}
    }
    return -1;
}
```

sigill_x86.s

```
    .text
    .globl illegal_instruction
# void illegal_instruction(int type)
illegal_instruction:
    test %eax, %eax
    jnz .L1
    ud2 # illegal instruction (2-byte opcode: 0x0f, 0x0b)
    jmp .L_done
.L1:
    cmp $1, %eax
    jnz .L2
    ud0 %rax, %rax # illegal instruction (4-byte opcode: 0x48, 0x0f, 0xff, 0xc0)
    jmp .L_done
.L2:
    cmp $2, %eax
    jnz .L_done
    ud1 %rax, %rax # illegal instruction (4-byte opcode: 0x48, 0x0f, 0xb9, 0xc0)
.L_done:
    ret
```

sigill_main.cpp

```
#include <atomic>
#include <format>
#include <cassert>
#include <iostream>
#include <signal.h>
#include <sys/ucontext.h>
#include <unistd.h>
#include "sigill_lib.hpp"

std::atomic<unsigned int> sigill_count(0);

void sigill_handler(int sig_no, siginfo_t* siginfo, void* context) {
    void* ip = get_ip(context);
    int length = get_ins_length(ip);
    if (length < 0) {abort();}
    set_ip(context, static_cast<unsigned char*>(ip) + length);
    ++sigill_count;
}

int main() {
    struct sigaction sa;
    sa.sa_sigaction = &sigill_handler;
    sigfillset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    if (sigaction(SIGILL, &sa, 0)) {abort();}
    for (int i = 0; i < 3; ++i) {
        illegal_instruction(i);
        std::cout << std::format("{}\n",
          static_cast<unsigned int>(sigill_count));
    }
}
```

[Example] SIGSEGV: Summary

- example of using signal handler for `SIGSEGV` signals in order to safely access memory locations that may be invalid
- registers signal handler for `SIGSEGV` signal
- safe memory access function works in conjunction with signal handler to allow program to check if access to particular address is valid
- program walks sequentially through pages in memory trying to access them safely
- when address is found that can be successfully accessed, program terminates
- information about failed/successful accesses are output
- assumes x86-64 architecture

sigsegv_lib.hpp

```
#include <sys/ucontext.h>

extern "C" int safe_read(void* addr);
extern char safe_read_ins;

inline bool is_safe_read(void* context, void* addr) {
    ucontext_t* ucontext = static_cast<ucontext_t*>(context);
    unsigned char* rip = reinterpret_cast<unsigned char*>(
      ucontext->uc_mcontext.gregs[REG_RIP]);
    return rip == reinterpret_cast<unsigned char*>(&safe_read_ins) &&
      reinterpret_cast<void*>(ucontext->uc_mcontext.gregs[REG_RDI]) == addr;
}

inline void safe_read_fail(void* context) {
    ucontext_t* ucontext = static_cast<ucontext_t*>(context);
    unsigned char* rip = reinterpret_cast<unsigned char*>(
      ucontext->uc_mcontext.gregs[REG_RIP]);
    unsigned long rax = ucontext->uc_mcontext.gregs[REG_RAX];
    rip += 2;
    rax = 0xffffffffU;
    ucontext->uc_mcontext.gregs[REG_RIP] = reinterpret_cast<unsigned long>(rip);
    ucontext->uc_mcontext.gregs[REG_RAX] = rax;
}
```

[Example] SIGSEGV: Code (2)

sigsegv_x86.s

```
    .text
    .globl safe_read
    .globl safe_read_ins
safe_read:
    mov $0, %eax
safe_read_ins:
    mov (%rdi), %al
.L0:
    # upon failure, %eax set to -1
    ret
    .data
    .globl safe_read_ins_len
safe_read_ins_len:
    .long .L0 - safe_read_ins
```

[Example] SIGSEGV: Code (3)

sigsegv_main.cpp

```
#include <atomic>
#include <format>
#include <iostream>
#include <signal.h>
#include <unistd.h>
#include "sigsegv_lib.hpp"

std::atomic<unsigned int> sigsegv_count(0);

void sigsegv_handler(int sig_no, siginfo_t* siginfo, void* context) {
    if (siginfo->si_code == SI_USER) {return;}
    if (!is_safe_read(context, siginfo->si_addr)) {std::abort();}
    safe_read_fail(context);
    ++sigsegv_count;
}

int main() {
    long page_size = sysconf(_SC_PAGE_SIZE);
    struct sigaction sa;
    sa.sa_sigaction = &sigsegv_handler;
    sigfillset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    if (sigaction(SIGSEGV, &sa, 0)) {std::abort();}
    uintptr_t addr = 0;
    int c;
    for (;; addr += page_size) {
        if ((c = safe_read(reinterpret_cast<void*>(addr))) >= 0) {break;}
        std::cout << std::format("read failed {:#x}\n", addr);
    }
    std::cout << std::format("read success {:#x} {}\n", addr, c);
    std::cout << std::format("number of faults: {}\n",
      static_cast<unsigned int>(sigsegv_count));
}
```

Section 2.5

Processes

Process Creation

- new process created by `fork` system call
- declaration:

```
int fork();
```

- creates new process (called child process) by duplicating calling process (called parent process)
- parent and child processes run in separate memory spaces
- at time of `fork`, parent and child memory spaces have identical content
- returns twice, once in calling (parent) process, once in newly created (child) process
- upon success, PID of child returned in parent and 0 returned in child
- upon failure, -1 returned in parent and no child process created
- regardless of number of threads in parent process, child process will have exactly one thread, corresponding to thread that called `fork`
- great care must be exercised when `fork` invoked from multithreaded process

[Example] Creating Child Process: Summary

- code example illustrates use of `fork` system call
- program creates child process via `fork`
- parent prints its PID and PID of child
- child prints its PID and PPID
- child also indicates if its PPID matches PID of creator (i.e., process that created child via `fork`) or corresponds to PID of reaper process
- race condition: PPID printed by child will correspond to reaper process if parent terminates prior to child querying PPID via `getppid`
- typically, if program run many times, will observe some instances where child reports its PPID as belonging to creator and others where reported to belong to reaper

[Example] Creating Child Process: Code

fork_1.cpp

```
#include <cassert>
#include <format>
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t parent_pid = getpid();
    std::cout << std::format("[parent] PID: {}\n", parent_pid);
    if (pid_t child_pid = fork(); child_pid > 0) {
        // parent
        assert(getpid() == parent_pid);
        std::cout << std::format("[parent] PID of child: {}\n", child_pid);
    } else if (child_pid == 0) {
        // child
        assert(getpid() != parent_pid);
        std::cout << std::format("[child] PID: {}\n", getpid());
        // note: if parent already terminated prior to getppid
        // call, this call will not return parent_pid
        pid_t ppid = getppid();
        std::cout << std::format("[child] PID of parent: {} ({})\n",
          ppid, ppid == parent_pid ? "creator" : "reaper");
        std::exit(0);
    } else {
        std::cout << "fork failed\n";
    }
}
```

Executing a Program

- can execute program given either pathname or file descriptor referring to program file
- `execve` system call used to execute program file referred to by pathname
- declaration:

```
int execve(const char* filename, char* const argv[],
↪ char* const envp[]);
```

- upon success, function does not return; upon failure, returns -1
- `fexecve` function (which invokes `execveat` system call) used to execute program file referred to by file descriptor
- declaration:

```
int fexecve(int fd, char* const argv[], char* const
↪ envp[]);
```

- similar to `execve` except program file specified by file descriptor instead of pathname

[Example] Running Program

exec_1.cpp

```
#include <cassert>
#include <iostream>
#include <string>
#include <vector>
#include <unistd.h>

int main() {
    std::vector<std::string> s{"ls", "-al", "/"};
    char* args[4]{&s[0][0], &s[1][0], &s[2][0], nullptr};
    char** env = environ; // environ is global variable
    if (execve("/bin/ls", &args[0], env) < 0) {
        std::cerr << "exec failed\n";
        return 1;
    }
    assert(false); // unreachable
}
```

- example illustrates use of `execve` system call
- program runs executable `/bin/ls` with arguments `"ls"`, `"-al"`, and `"/"`

Waiting for Changes in State of Child Process

- several system calls provided for waiting for change in state of child process (e.g., child terminated or stopped) and optionally return status of child
- `wait` system call waits for ***any*** child process to terminate; declaration:

  ```
  pid_t wait(int* status);
  ```

- on success, returns PID of child and, if `status` not null, sets `*status` to child status; on failure, returns -1
- child status provides indication about how child terminated (e.g., child terminated normally with particular exit status or terminated abnormally due to particular signal)
- `waitpid` system call can be used to wait for ***specific*** child or ***any*** child process to change state; declaration:

  ```
  pid_t waitpid(pid_t pid, int* status, int options);
  ```

- `waitid` system call can be used to wait for any child ***in process group***, or ***specific*** child, or ***any*** child to change state; declaration:

  ```
  pid_t waitid(idtype_t idtype, id_t id, siginfo_t* infop,
  ↪ int options);
  ```

Inspecting Wait Status

- several macros provided for extracting information from child status returned by wait family of system calls
- `WIFEXITED(status)` returns true if child terminated normally (i.e., through `exit` system call)
- `WEXITSTATUS(status)` returns exit status of child (assuming that child exited normally)
- `WIFSIGNALED(status)` returns true if child terminated by signal
- `WTERMSIG(status)` returns number of signal that caused child to terminate
- `WCOREDUMP(status)` returns true if child produced core dump; can only be used if process terminated due to signal
- `WIFSTOPPED(status)` returns true if child stopped by delivery of signal
- `WIFCONTINUED(status)` returns true if child resumed due to delivery of `SIGCONT` signal
- `WSTOPSIG(status)` returns number of signal that caused child to stop; only valid if `WIFSTOPPED(status)` is true

[Example] Waiting for Child Process Termination: Summary

- code example illustrates use of `wait` system call as well as some macros used to query child status returned by `wait` family of system calls
- parent process creates child process via `fork`
- then, parent waits for child to terminate via `wait`
- parent prints exit status of child after child terminates
- child process simply sleeps for short period and then terminates with exit status 0

[Example] Waiting for Child Process Termination: Code

fork_wait_1.cpp

```
#include <format>
#include <iostream>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    if (pid_t child_pid = fork(); child_pid > 0) {
        // parent
        int status;
        if (wait(&status) < 0) {std::cerr << "wait failed\n";}
          // or equivalently, waitpid(-1, &status, 0)
        if (WIFEXITED(status)) {
            std::cout << std::format("[parent] child exit status: {}\n",
              WEXITSTATUS(status));
        } else {std::cout << "[parent] unexpected child state change\n";}
    } else if (child_pid == 0) {
        // child
        std::cout << "[child] sleeping\n";
        sleep(2);
        std::cout << "[child] exiting\n";
        std::exit(0);
    } else {
        std::cerr << "fork failed\n";
    }
}
```

[Example] Waiting for Specific Child Termination: Summary

- code example illustrates use of `waitpid` system call as well as some macros used to query child status returned by `wait` family of system calls
- parent process creates several child processes via `fork`
- each child process sleeps for different amounts of time before exiting with different exit statuses
- parent loops waiting for specific child to terminate via `waitpid`
- parent prints exit status for each child when it terminates

[Example] Waiting for Specific Child Termination: Code

fork_wait_2.cpp

```
#include <format>
#include <iostream>
#include <vector>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int exitstatus(int wstatus)
  {return (WIFEXITED(wstatus) ? WEXITSTATUS(wstatus) : -1);}

int main() {
    constexpr int num_children = 8;
    std::vector<pid_t> child_pids;
    for (int i = 0; i < num_children; ++i) {
        if (pid_t child_pid = fork(); child_pid > 0) {
            child_pids.push_back(child_pid);
        } else if (child_pid == 0) {
            std::cout << std::format("[child {}] sleeping {}\n", i, i);
            sleep(i);
            std::cout << std::format("[child {}] exiting\n", i);
            std::exit(i);
        } else {
            std::cerr << "fork failed\n";
        }
    }
    for (int i = 0; i < num_children; ++i) {
        int status;
        if (waitpid(child_pids[num_children - i], &status, 0) < 0)
          {std::cerr << "wait failed\n";}
        std::cout << std::format("[parent] child {} exit status: {}\n", i,
          exitstatus(status));
    }
    std::cout << "[parent] exiting\n";
}
```

[Example] Running Program as Child Process: Summary

- code example illustrates use of `execve` system call (as well as `fork`)
- parent process creates child via `fork`
- then, parent waits for child process to terminate and obtains its exit status
- child process executes another program via `execve`
- pattern of using `fork` followed by `execve` very common when process wants to run another program (without actually transforming into instance of that other program)

[Example] Running Program as Child Process: Code

fork_exec_1.cpp

```
#include <cassert>
#include <iostream>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char** argv, char** envp) {
    if (argc < 2) {std::cerr << "invalid usage\n"; std::exit(1);}
    if (pid_t child_pid = fork(); child_pid > 0) {
        // parent
        int status;
        if (wait(&status) < 0) {std::cerr << "wait failed\n";}
    } else if (child_pid == 0) {
        // child
        if (execve(argv[1], &argv[1], envp) < 0) {
            std::cerr << "exec failed\n";
            std::exit(1);
        }
    } else {
        std::cerr << "fork failed\n";
        std::exit(1);
    }
}
```

Section 2.6

Memory Mappings

- in order to manage virtual memory of process, operating system provides system calls for managing memory mappings
- two types of mappings:
  1. file mapping
  2. anonymous mapping
- **file mapping**: maps region of (regular or block device) file directly into virtual address space of process; after file is mapped, its contents can be accessed simply by accessing corresponding memory region
- file mapping often referred to as memory-mapped file
- **anonymous mapping**: maps zero-filled pages into virtual address space of process

Sharing of Memory Mappings

- memory in mapping of one process can be shared with mappings in other processes
- two or more processes may share pages (i.e., page-table entries of each process refer to same pages of physical memory)
- for example, sharing of pages can result from:
  - two processes mapping same region of same file
  - child process created by fork inherits copies of parent's mappings
- when page shared between two or more processes modified, resulting behavior depends on whether page belongs to mapping that is private or shared
- **private mapping**: modification to contents of mapping not visible to other processes and for file mapping not carried through to underlying file (i.e., copy-on-write semantics)
- **shared mapping**: modification to contents of mapping are visible to other processes that share same mapping and for file mapping are carried through to underlying file

Use Cases for Various Kinds of Mappings

Mapping Type	Modification Visibility	Use Cases
File	Private	initialize memory from contents of file (e.g., loading parts of binary executables or shared libraries)
Anonymous	Private	allocate new (zero-filled) memory for process
File	Shared	memory-mapped I/O; sharing memory between processes for (fast) IPC
Anonymous	Shared	sharing memory between related processes for (fast) IPC

Memory Mappings and Fork/Exec

- mappings inherited by child process of fork, including private/shared attribute
- mapping lost when process performs exec

/proc/$pid/maps File

- information about process's mapping visible in /proc/$pid/maps file, where $pid denotes PID
- example of maps file:

```
00400000-00402000 r-xp 00000000 fd:05 24775559                  /home/jdoe/bin/hello
00402000-00404000 rw-p 00001000 fd:05 24775559                  /home/jdoe/bin/hello
7ffe9b28f000-7ffe9b2b1000 rw-p 00000000 00:00 0                 [stack]
7ffe9b30a000-7ffe9b30d000 r--p 00000000 00:00 0                 [vvar]
7ffe9b30d000-7ffe9b30e000 r-xp 00000000 00:00 0                 [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0         [vsyscall]
```

- [stack] is mapping for process's stack
- [vvar], [vdso], and [vsyscall] are special mappings created by kernel for every process

Creating Memory Mappings

- memory mapping can be created with `mmap` system call, which has declaration:

```
void *mmap(void *addr, size_t length, int prot,
  int flags, int fd, off_t offset);
```

- `addr`: address associated with mapping
- `length`: length of mapping in bytes
- `prot`: protection flags for mapping
- `flags`: flags for mapping
- `fd`: file descriptor specifying file to map
- `offset`: starting offset within file to map
- depending on value of `flags`, meaning of some other parameters may change somewhat
- creates anonymous mapping if `MAP_ANONYMOUS` bit set in `flags`; otherwise, creates file mapping

Creating Memory Mappings (Continued)

- mapped file can be regular or block device file
- some constraints on alignment of region in file and region in memory

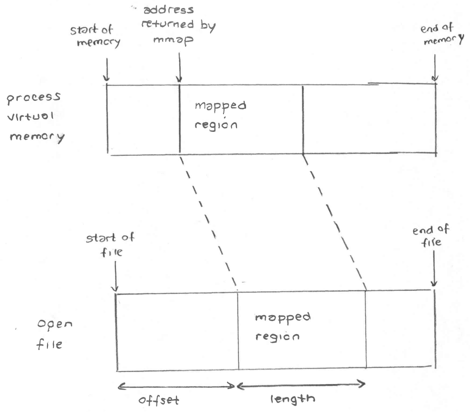
Memory-Mapping Access Values

Flag/Value	Description
PROT_NONE	contents of region cannot be accessed at all
PROT_READ	contents of region can be read
PROT_WRITE	contents of region can be modified (i.e., written)
PROT_EXEC	contents of region can be executed
PROT_SEM	memory can be used for atomic operations
PROT_SAO	memory should have strong access ordering (used by PowerPC architecture)
PROT_GROWSUP	apply protection mode up to end of mapping that grows upwards
PROT_GROWSDOWN	apply protection mode down to beginning of mapping that grows downwards

Some Memory-Mapping Flags

Value	Description
MAP_SHARED	create shared mapping
MAP_SHARED_VALIDATE	create shared mapping but fail if unknown flag specified
MAP_PRIVATE	create private mapping
MAP_FIXED	mapping at fixed address
MAP_32BIT	put mapping in first 4 GB of memory
MAP_ANONYMOUS	create anonymous mapping
MAP_FIXED	mapping at fixed address
MAP_FIXED_NOREPLACE	mapping at fixed address but do not replace existing mapping
MAP_LOCKED	mapped region to be locked in same way as mlock
MAP_STACK	mapping is suitable for process or thread stack
MAP_UNINITIALIZED	do not clear anonymous pages (usually only enabled on embedded devices)

Memory-Mapped File



- address returned by `mmap` is always page aligned
- `offset` must be integer multiple of page size
- above diagram assumes that `length` is multiple of page size

[Example] Creating Memory Mapping: Summary

- code example illustrates how to use `mmap` system call to access memory-mapped file for reading
- program computes System-V checksum on file whose pathname is specified as command-line argument
- file to be checksummed is mapped into address space of process
- then, data to be checksummed accessed via normal memory reads (instead of using `read` system calls)

[Example] Creating Memory Mapping: Code

checksum_1.cpp

```
#include <format>
#include <iostream>
#include <numeric>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

unsigned int sysv_checksum(unsigned char* buf, size_t len) {
    unsigned long sum = 0;
    for (; len; ++buf, --len) {sum += *buf;}
    unsigned long x = (sum & 0xffff) + ((sum & 0xffffffff) >> 16);
    return (x & 0xffff) + (x >> 16);
}

int main(int argc, char** argv) {
    if (argc < 2) {return 1;}
    int fd;
    if ((fd = open(argv[1], O_RDONLY)) < 0) {return 1;}
    struct stat stat_buf;
    if (fstat(fd, &stat_buf) < 0) {return 1;}
    void* addr;
    if ((addr = mmap(nullptr, stat_buf.st_size, PROT_READ, MAP_PRIVATE,
      fd, 0)) == MAP_FAILED) {return 1;}
    if (close(fd) < 0) {return 1;}
    std::cout << std::format("{}\n", sysv_checksum(
      static_cast<unsigned char*>(addr), stat_buf.st_size));
}
```

Deleting Memory Mappings

- can delete memory mapping with `munmap` system call, which has declaration:

```
int munmap(void *addr, size_t length);
```

- deletes mapping for specified address range
- `addr`: starting address of mapping (which must be page aligned)
- `length`: length of mapping in bytes (which need not be multiple of page size)
- not error if indicated range does not contain any mapped pages

[Example] Deleting Memory Mappings: Summary

- code example illustrates use of `munmap` system call
- unmapping pages used as creative way for process to commit suicide
- program first queries page size for system
- then, program loops, unmapping exponentially-growing number of pages
- program will inevitably crash violently, as it will eventually access address in page that has been unmapped

[Example] Deleting Memory Mappings: Code

crash_1.cpp

```
#include <format>
#include <iostream>
#include <iomanip>
#include <sys/mman.h>
#include <unistd.h>

int main() {
    char buf[256];
    sprintf(buf, "cat /proc/%d/maps", getpid());
    std::cout << "The memory mappings for this process are as follows:\n";
    if (system(buf)) {std::cerr << "cat failed\n"; return 1;}
    long page_size = sysconf(_SC_PAGE_SIZE);
    size_t length = page_size;
    std::cout << "WARNING: This program is likely going to crash very soon.\n";
    for (size_t length = page_size; length; length <<= 1) {
        std::cout << std::format("Deleting all memory mappings up to "
          "(but not including) address {:#8x}.\n", length);
        if (munmap(0, length)) {
            std::cerr << "munmap failed\n";
            return 1;
        }
    }
}
```

Changing Protection of Mapping

- can change access protections for calling process's memory pages using `mprotect` system call, which has declaration:

```
int mprotect(void *addr, size_t len, int prot);
```

- changes access protections for calling process's memory pages containing any part of address range [`addr`, `addr+len`)
- `addr` must be page aligned
- `prot` is combination of access flags

[Example] Memory Protection: Summary

- code example illustrates use of `mprotect` system call
- program allocates block of memory that is page aligned and contains several pages
- applies different memory protections to various pages in block
- accesses pages in various ways to show consequences of memory protections applied to those pages

[Example] Memory Protection: Code

mprotect_1.cpp

```
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <sys/mman.h>
#include <unistd.h>

int main(int argc, char** argv) {
    long page_size = sysconf(_SC_PAGESIZE);
    char* ptr = static_cast<char*>(
      std::aligned_alloc(page_size, 4 * page_size));
    std::fill_n(ptr, 4 * page_size, 'A');
    char* none_ptr = ptr;
    char* ro_ptr = ptr + 1 * page_size;
    char* wo_ptr = ptr + 2 * page_size;
    char* rw_ptr = ptr + 3 * page_size;
    if (mprotect(none_ptr, page_size, 0)) {abort();}
    if (mprotect(ro_ptr, page_size, PROT_READ)) {abort();}
    if (mprotect(wo_ptr, page_size, PROT_WRITE)) {abort();}
    if (mprotect(rw_ptr, page_size, PROT_READ | PROT_WRITE)) {abort();}
    char c;
    // c = *none_ptr; // SEGFAULT (cannot read)
    // *none_ptr = 'B'; // SEGFAULT (cannot write)
    c = *ro_ptr; // OK (can read)
    // *ro_ptr = 'B'; // SEGFAULT (cannot write)
    // c = *wo_ptr; // may SEGFAULT (read may be disallowed)
    *wo_ptr = 'B'; // OK (can write)
    c = *rw_ptr; // OK (can read)
    *rw_ptr = 'B'; // OK (can write)
}
```

Synchronizing Underlying File With Memory

- can synchronize underlying file with memory using `msync` system call, which has declaration:

```
int msync(void *addr, size_t length, int flags);
```

- `addr`: start of memory area
- `length`: length of memory area in bytes
- `flags`: specify how synchronization should be performed
- flag values that can be combined by OR-ing to form `flags`:

Flag	Description
`MS_ASYNC`	requests update but does not wait for it to complete
`MS_SYNC`	requests update and waits for it to complete
`MS_INVALIDATE`	asks to invalidate other mappings of same file (so they can be updated with fresh values just written)

Other System Calls Related to Memory Mappings

- some other functions related to memory mappings:
  - `mincore`
  - `madvise`
  - `mlock`
  - `mlock2`
  - `munlock`
  - `mlockall`
  - `munlockall`
  - `memfd_create`

Remarks on Memory-Mapped Files

- if I/O operation occurs on file operations initiated by read/write to mmapped region, `SIGSEGV` or `SIGBUS` signal is generated
- can be challenging to handle such failures in multithreaded applications
- file-backed mappings less problematic when file opened only for reading (e.g., as in case of DSOs used by dynamic linkers and loaders)

[Example] Shared Mapping of File: Summary

- code example illustrates use of shared file mapping
- program creates/truncates file and writes `"Hello, World!\n"` to it
- uses shared file mapping
- file opened and truncated to size of data to be written
- file mapped to pages in address space of process via `mmap`
- pages written with data intended for file
- memory pages flushed to disk via `msync`

[Example] Shared Mapping of File: Code

mmap_1.cpp

```
#include <iostream>
#include <string>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

int main(int argc, char** argv) {
    if (argc < 2) {std::cerr << "bad usage\n"; return 1;}
    const std::string hello("Hello, World!\n");
    int fd = open(argv[1], O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR);
    if (fd < 0) {std::cerr << "open failed\n"; return 1;}
    if (ftruncate(fd, hello.size()) < 0)
      {std::cerr << "ftruncate failed\n"; return 1;}
    void* ptr = mmap(nullptr, hello.size(),
      PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (ptr == MAP_FAILED) {std::cerr << "mmap failed\n"; return 1;}
    if (close(fd) < 0) {std::cerr << "close failed\n"; return 1;}
    char* cptr = static_cast<char*>(ptr);
    cptr = std::copy(hello.begin(), hello.end(), cptr);
    if (msync(ptr, hello.size(), MS_SYNC))
      {std::cerr << "msync failed"; return 1;}
    if (munmap(ptr, hello.size())) {std::cerr << "munmap failed\n"; return 1;}
}
```

[Example] Shared Anonymous Mapping: Summary

- code example illustrates use of shared anonymous mapping
- shared anonymous mapping used by child process to provide data to parent via memory buffer
- parent process creates shared anonymous mapping
- parent process creates child process (via `fork`) and waits for child process to terminate
- child process copies string into buffer in pages of shared anonymous mapping and exits
- after child process terminates, parent process prints contents of buffer (which contains data written by child process)

[Example] Shared Anonymous Mapping: Code

mmap_2.cpp

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <string>
#include <sys/mman.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    long page_size = sysconf(_SC_PAGE_SIZE);
    void* ptr = mmap(nullptr, page_size,
      PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (ptr == MAP_FAILED) {std::cerr << "mmap failed\n"; return 1;}
    char* cptr = static_cast<char*>(ptr);
    assert(*cptr == '\0');
    if (int child_pid = fork(); child_pid > 0) {
        int status;
        if (waitpid(child_pid, &status, 0) < 0)
          {std::cerr << "wait failed\n"; return 1;}
        if (!(WIFEXITED(status) && WEXITSTATUS(status) == 0))
          {std::cerr << "child failed\n"; return 1;}
        std::cerr << cptr;
    } else if (child_pid == 0) {
        std::string hello("Hello, World!\n");
        std::copy(hello.begin(), hello.end(), cptr);
    } else {std::cerr << "fork failed\n"; return 1;}
}
```

[Example] Mmap Allocator: Summary

- code example illustrates use of private anonymous mapping
- class template `mmap_allocator` provides custom memory allocator (compatible with allocators used by C++ standard library)
- memory allocator ensures that each memory block is page aligned and does not share any pages with other memory blocks
- `mmap_allocator` could be practically useful if, for example, one wanted allocator that provides page-aligned memory blocks so that different memory protections could be used for data stored in different memory blocks
- allocation operation obtains storage via `mmap`
- deallocation operation frees storage via `munmap`
- to illustrate use of `mmap_allocator`, code uses `mmap_allocator` to allocate page-aligned storage for `std::vector` container

[Example] Mmap Allocator: `mmap_allocator` Code

mmap_allocator.hpp

```
#include <cstddef>
#include <new>
#include <limits>
#include <sys/mman.h>
#include <unistd.h>

template <class T> struct mmap_allocator {
    using value_type = T;
    mmap_allocator() noexcept {}
    template <class U> mmap_allocator(const mmap_allocator<U>&) noexcept {}
    T* allocate(std::size_t n) const;
    void deallocate(T* p, std::size_t n) const noexcept;
    template <class U> bool operator==(const mmap_allocator<U>&)
      const noexcept {return true;}
    template <class U> bool operator!=(const mmap_allocator<U>&)
      const noexcept {return false;}
};

template <class T>
T* mmap_allocator<T>::allocate(std::size_t n) const {
    if (!n) {return nullptr;}
    if (n > std::numeric_limits<std::size_t>::max() / sizeof(T))
      {throw std::bad_array_new_length();}
    void* ptr = mmap(nullptr, n * sizeof(T),
      PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (ptr == MAP_FAILED) {throw std::bad_alloc();}
    return static_cast<T*>(ptr);
}

template <class T>
void mmap_allocator<T>::deallocate(T* p, std::size_t n) const noexcept {
    if (!n) {return;}
    munmap(p, n * sizeof(T));
}
```

[Example] Mmap Allocator: User Code

mmap_allocator_main.cpp

```
#include <boost/align/is_aligned.hpp>
#include <cassert>
#include <format>
#include <iostream>
#include <vector>
#include "mmap_allocator.hpp"

int main() {
    long page_size = sysconf(_SC_PAGE_SIZE);
    std::vector<int, mmap_allocator<int>> v{1, 2, 3};
    std::vector<int> w{1, 2, 3};
    bool v_aligned = boost::alignment::is_aligned(page_size, v.data());
    assert(v_aligned);
    bool w_aligned = boost::alignment::is_aligned(page_size, w.data());
    std::cout << std::format("{} {}\n", v_aligned,
      static_cast<void*>(v.data()));
    std::cout << std::format("{} {}\n", w_aligned,
      static_cast<void*>(w.data()));
}
```

Section 2.7

Threads

Linux Threading Model

- Linux kernel does not make any real distinction between process and thread
- as far as Linux kernel is concerned, any task that can be scheduled and run is deemed process
- to avoid confusion, will use term “Linux-kernel task” to refer to Linux kernel notion of process
- each thread is single Linux-kernel task
- each process is simply collection of one or more Linux-kernel tasks, which may share some resources amongst themselves (such as virtual memory), with one of these Linux-kernel tasks designated as main one
- each thread has system-unique thread ID (TID), which corresponds to Linux-kernel task ID
- each process has system-unique process ID (PID) and thread group ID (TGID), which correspond to ID of main Linux-kernel task in collection of tasks that comprise process

Creating New Threads/Processes

- can create new threads/processes with `clone` system call
- declaration:

```
int clone(int (*func)(void *), void* stack, int flags,
  void* arg, ... /* pid_t* parent_tid,
  void* tls, pid_t* child_tid */);
```

- `func`: function for new thread to execute
- `stack`: location of stack to be used for execution
- `flags`: flags that control specific behavior of `clone` operation
- `arg`: argument to be passed to function `func`
- upon success, thread ID of child thread returned (in caller's thread)
- upon failure, no child thread is created and -1 returned (in caller's thread)

Commonly-Used Flags for `clone` System Call

Flag	Description
CLONE_CHILD_CLEARTID	if set, clear child TID at location pointed to by `child_tid` in child memory when child exits and perform wakeup on futex at that address
CLONE_FILES	if set, calling process and child process share same file descriptor table; otherwise, child process inherits copy of all open file descriptors in calling process at time of `clone` call
CLONE_FS	if set, caller and child process share same filesystem information; otherwise, state copied at time of `clone` call
CLONE_PARENT_SETTID	if set, store child thread ID at location pointed to by `parent_tid` in parent's memory
CLONE_SIGHAND	if set, calling process and child process share same signal-handler table; otherwise, child process inherits copy of table at time of `clone` call
CLONE_THREAD	if set, child is placed in same thread group as calling process
CLONE_VM	if set, calling process and child process share same memory space

[Example] Clone: Summary

- code example illustrates how `clone` system call can be used to create new thread or process (although primarily interested in new thread case)
- command-line option used to determine whether to create new thread or process
- program creates new thread/process via `clone`
- parent and child each print various process-related IDs
- child additionally sleeps for short duration and prints message
- in order to avoid race conditions (such as data races), parent must wait for child thread to finish execution
- some complexity in code example comes from need to detect when child thread has finished executing, which necessitates use of futex

clone_1.cpp

```
#include <atomic>
#include <cassert>
#include <format>
#include <iostream>
#include <linux/futex.h>
#include <sched.h>
#include <syscall.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

pid_t gettid() {return syscall(SYS_gettid);}

int futex(int *uaddr, int futex_op, int val, const struct timespec *timeout,
  int *uaddr2, int val3)
{return syscall(SYS_futex, uaddr, futex_op, val, timeout, uaddr2, val3);}

int atomic_load_int(int* ptr) {
    // WARNING: This is ugly and non-portable.
    static_assert(sizeof(std::atomic<int>) == sizeof(int));
    return reinterpret_cast<std::atomic<int>*>(ptr)->load();
}
```

clone_1.cpp

```
struct child_args {
    bool make_thread; // invoker creating thread (as opposed to process)
    pid_t tid; // TID of invoker of clone
    pid_t pid; // PID of invoker of clone
    pid_t ppid; // PPID of invoker of clone
    int ret; // return value (used in thread case)
};

int child_func(void* arg) {
    child_args* args = static_cast<child_args*>(arg);
    assert(args->tid != gettid()); // not invoker of clone
    if (args->make_thread) {
        std::cerr << std::format("[thread] PID {}\n", getpid());
        assert(args->pid == getpid()); // running in same process
        assert(args->ppid == getppid()); // has same parent process
        assert(gettid() != getpid()); // not main thread of process
    } else {
        std::cerr << std::format("[process] PID {}\n", getpid());
        assert(args->pid != getpid()); // running in different process
        assert(args->pid == getppid()); // child of invoker of clone
        assert(gettid() == getpid()); // main thread of process
    }
    sleep(2);
    std::cout << "Hello, World!\n" << std::flush;
    args->ret = std::cout ? 0 : 1;
    return args->ret;
}
```

[Example] Clone: Code (3)

clone_1.cpp

```
int main(int argc, char** argv) {
    bool make_thread = argc < 2;
    std::cerr << std::format("[main] PID {}\n", getpid());
    constexpr std::size_t stack_size = 64 * 1024;
    static char stack[stack_size];
    int clone_flags = (make_thread) ? (CLONE_THREAD | CLONE_SIGHAND |
      CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_CHILD_CLEARTID |
      CLONE_PARENT_SETTID) : SIGCHLD;
    alignas(std::atomic<int>) int tid;
    child_args arg = {make_thread, gettid(), getpid(), getppid(), -1};
    pid_t child_tid;
    if ((child_tid = clone(child_func, stack + stack_size, clone_flags, &arg,
     &tid, 0, &tid)) < 0)
      {std::cerr << "clone failed\n"; return 1;}
    int exit_status;
    if (make_thread) {
        int tmp_tid;
        while ((tmp_tid = atomic_load_int(&tid)) == child_tid) {
            std::cerr << "[main] sleeping\n";
            int ret;
            while ((ret = futex(&tid, FUTEX_WAIT, tmp_tid, nullptr, 0, 0)) < 0
              && ret == EAGAIN) {}
            if (ret < 0) {std::cerr << "futex failed\n"; return 1;}
            std::cerr << "[main] awoken\n";
        }
        exit_status = arg.ret;
    } else {
        int status;
        if (waitpid(child_tid, &status, 0) != child_tid)
          {std::cerr << "wait failed\n"; return 1;}
        exit_status = (WIFEXITED(status)) ? WEXITSTATUS(status) : 1;
    }
    std::cerr << std::format("[main] exit status {}\n", exit_status);
}
```

Section 2.8

Futexes

Locks

- **lock** is synchronization mechanism for providing mutual exclusion for access to shared resource in multithreaded environment
- lock has two basic operations:
  - **acquire**: takes lock
  - **release**: relinquishes lock
- shared resource can only be accessed by thread if thread holds lock
- how many threads may simultaneously acquire lock depends on type of lock
- in case of exclusive lock, only one thread can hold lock at any given time
- if thread attempts to acquire lock and lock cannot currently be acquired, operation either waits until lock can be acquired or fails with error
- acquire operation may spin in loop waiting to acquire lock or block (where blocking requires operating system intervention)
- thread acquires lock before accessing shared resource and releases lock when finished accessing resource

Compiler Intrinsics for Atomic Memory Operations

- typically, compilers provide intrinsic (i.e., built-in) functions for atomic memory operations, such as:
  - load operation
  - store operation
  - test-and-set (TAS) and clear operations
  - compare-and-swap (CAS) operation
  - swap operation
- in what follows, we consider only intrinsics provided by GCC and Clang

Spinlocks

- **spinlock**: lock for which thread trying to acquire it simply waits in loop, repeatedly checking if lock is available
- thread remains active while waiting to acquire lock
- spinlock efficient when time required to acquire lock is very short
- makes poor use of processor resources if wait time is long
- use special processor instructions and typically do not need operating system intervention

Atomic Test and Set (TAS) and Clear

- intrinsic for atomic test-and-set (TAS) has declaration:
  - `bool __atomic_test_and_set(void* ptr, int mem_order);`
- `ptr`: address of byte for operation
- `mem_order` specifies memory order
- atomically performs following:
  1. reads `*ptr`
  2. sets `*ptr` to “true”
  3. returns boolean value indicating if value read was “true”
- intrinsic for atomic clear has declaration:
  - `void __atomic_clear(void* ptr, int mem_order);`
- atomically sets `*ptr` to “false”

Memory Orders

Name	Description
`__ATOMIC_RELAXED`	implies no interthread ordering constraints
`__ATOMIC_CONSUME`	not advisable to use
`__ATOMIC_ACQUIRE`	creates interthread happens-before constraint from release (or stronger order) store to this acquire load; can prevent hoisting of code to before operation
`__ATOMIC_RELEASE`	creates interthread happens-before constraint to acquire (or stronger) loads that read from this store; can prevent sinking of code to after operation
`__ATOMIC_ACQ_REL`	combines effects of `__ATOMIC_ACQUIRE` and `__ATOMIC_RELEASE`
`__ATOMIC_SEQ_CST`	enforces total ordering with all other `__ATOMIC_SEQ_CST` operations (i.e., sequentially consistent)

Atomic TAS and Clear on x86-64

atomic_tas.cpp

```
bool test_and_set(bool& x) noexcept {
    return __atomic_test_and_set(&x, __ATOMIC_ACQUIRE); // GCC/Clang
}

void clear(bool& x) noexcept {
    __atomic_clear(&x, __ATOMIC_RELEASE); // GCC/Clang
}
```

- atomic test-and-set with acquire memory order maps to:

```
# calling convention: rdi = &x
mov $1, %eax
xchg (%rdi), %al # swap x and al
ret
```

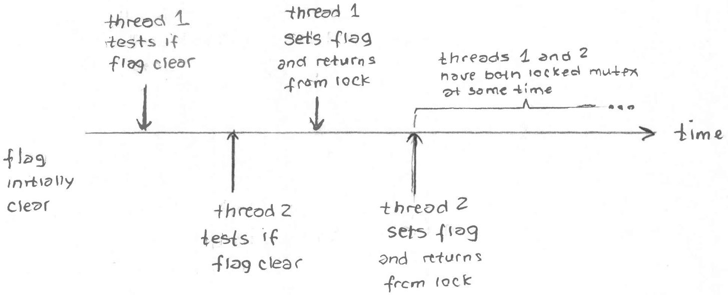
- atomic clear with release memory order maps to:

```
# calling convention: rdi = &x
movb $0, (%rdi)
ret
```

spinlock1.hpp

```
class spinlock {
public:
    spinlock() {clear(m_);}
    spinlock(const spinlock&) = delete;
    spinlock& operator=(const spinlock&) = delete;
    void lock() noexcept {
        while (test_and_set(m_)) {}
    }
    bool try_lock() noexcept {
        return !test_and_set(m_);
    }
    void unlock() noexcept {
        clear(m_);
    }
private:
    static bool test_and_set(bool& x) noexcept {
        return __atomic_test_and_set(&x, __ATOMIC_ACQUIRE); // GCC/Clang
    }
    static void clear(bool& x) noexcept {
        __atomic_clear(&x, __ATOMIC_RELEASE); // GCC/Clang
    }
    bool m_;
    static_assert(__atomic_always_lock_free(sizeof(char), 0));
};
```

- if test-and-set (TAS) operation not atomic, situations like following become possible:



[Example] TAS-Based Spinlock: User Code

spinlock1_app.cpp

```
#include <format>
#include <iostream>
#include <thread>
#include <vector>
#include "spinlock1.hpp"

spinlock m;
unsigned long long count = 0;

void worker() {
    for (int i = 0; i < 10'000; ++i) {
        m.lock();
        ++count;
        m.unlock();
    }
}

int main() {
    std::vector<std::jthread> threads;
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(worker);
    }
    for (auto&& t : threads) {t.join();}
    std::cout << std::format("{}\n", count);
}
```

Atomic Compare and Swap (CAS)

- intrinsic for atomic compare-and-swap (CAS) has declaration:

```
bool __atomic_compare_exchange_n(T* ptr, T* expected,
  T desired, bool weak, int success_mem_order,
  int fail_mem_order);
```

- `ptr`: address of object (of type `T`) for CAS operation
- `expected`: address of expected value
- `desired`: desired value
- `weak`: boolean flag indicating if weak CAS (i.e., operation allowed to fail spuriously)
- `success_mem_order`: memory order for read-modify-write operation in case of success
- `fail_mem_order`: memory order for read-modify-write operation in case of failure
- atomically performs following: tests if `*ptr == *expected` and
  - if true (i.e., success), sets `*ptr` to `desired`
  - otherwise (i.e., failure), sets `*expected` to `*ptr`
- returns true upon success (i.e., `*ptr` written) and false otherwise

Atomic CAS on x86-64

atomic_cas.cpp

```
int atomic_cas(int& x, int expected, int desired) noexcept {
    __atomic_compare_exchange_n(&x, &expected, desired, false,
      __ATOMIC_ACQUIRE, __ATOMIC_ACQUIRE); // GCC/Clang
    return expected; // return initial value of *ptr
}

void atomic_store(int& x, int value) noexcept {
    __atomic_store(&x, &value, __ATOMIC_RELEASE); // GCC/Clang
}
```

- atomic compare-and-swap with acquire memory order maps to:

```
# calling conventions: edi = &x; esi = &expected; edx = desired
mov %esi, %eax # eax = &expected
lock cmpxchg %edx, (%rdi)
  # eax = x; if eax == expected, x = desired
ret
```

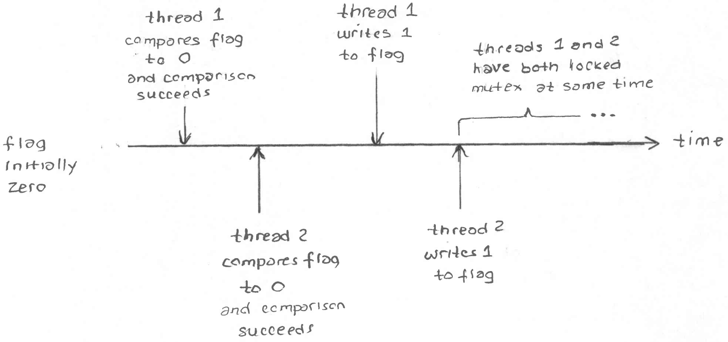
- atomic store with release memory order maps to:

```
# calling conventions: edi = &x
mov %esi, (%rdi) # write value to *ptr
ret
```

- on x86, all 32-bit loads/stores with 4-byte alignment are atomic spinlock2.hpp

```
class spinlock {
public:
    spinlock() : m_(0) {}
    spinlock(const spinlock&) = delete;
    spinlock& operator=(const spinlock&) = delete;
    void lock() noexcept {
        while (atomic_cas(&m_, 0, 1) != 0) {}
    }
    bool try_lock() noexcept {
        return atomic_cas(&m_, 0, 1) == 0;
    }
    void unlock() noexcept {
        atomic_store(&m_, 0);
    }
private:
    static int atomic_cas(int* addr, int expected, int desired)
      noexcept {
        __atomic_compare_exchange_n(addr, &expected, desired, false,
          __ATOMIC_ACQUIRE, __ATOMIC_ACQUIRE); // GCC/Clang
        return expected; // return initial value of *ptr
    }
    static void atomic_store(int* addr, int value) noexcept {
        __atomic_store(addr, &value, __ATOMIC_RELEASE); // GCC/Clang
    }
    int m_;
    static_assert(__atomic_always_lock_free(sizeof(int), 0));
};
```

- if compare-and-swap (CAS) operation not atomic, situations like following become possible:



Ticket Spinlocks

- ticket spinlock is type of spinlock that provides fairness guarantee
- follows model used by some businesses to serve customers in order of arrival without forcing customers to stand in a line
- each customer issued ticket upon arrival with integer value
- successive tickets issued have successive integer values
- business serves customers in order of increasing ticket number
- in case of ticket spinlock, each thread wanting to acquire lock given ticket consisting of integer value
- spinlock grants lock to threads in order of ticket number
- ticket spinlocks used internally by some operating systems (e.g., Linux kernel)

Advantages/Disadvantages of Ticket Spinlocks

- ticket spinlocks are fair in sense that thread cannot be indefinitely starved out of acquiring mutex by other threads
- do not need to know maximum number of threads that will attempt to acquire mutex at given time
- if number of threads trying to acquire mutex exceeds number of threads that hardware can run simultaneously, performance can degrade significantly (so called problem of preemption intolerance)
- number of cache line invalidations triggered when acquiring/releasing lock is $O(n)$ where n is number of threads (whereas $O(1)$ would be preferable)

[Example] Ticket Spinlock: `spinlock` Class

ticketlock1.hpp

```
class ticketlock {
public:
    ticketlock() : ticket_(0), current_(0) {}
    ticketlock(const ticketlock&) = delete;
    ticketlock& operator=(const ticketlock&) = delete;
    void lock() noexcept {
        unsigned int ticket = atomic_fetch_and_inc(ticket_);
        while (atomic_load(current_) != ticket) {}
    }
    void unlock() noexcept {
        atomic_fetch_and_inc(current_);
    }
private:
    static unsigned int atomic_fetch_and_inc(unsigned int& x) noexcept {
        return __atomic_fetch_add(&x, 1, __ATOMIC_ACQ_REL); // GCC/Clang
    }
    static unsigned int atomic_load(unsigned int& x) noexcept {
        return __atomic_load_n(&x, __ATOMIC_ACQUIRE); // GCC/Clang
    }
    static constexpr int cacheline_size = 128;
    alignas(cacheline_size) unsigned int ticket_;
    alignas(cacheline_size) unsigned int current_;
    static_assert(__atomic_always_lock_free(sizeof(unsigned int), 0));
};
```

Fast Userspace Mutexes (Futexes)

- futex mechanism allows kernelspace wait queue to be associated with integer in userspace
- perform operations on futex using `futex` system call, which has declaration:

```
int futex(int *uaddr, int futex_op, int val,
  const struct timespec *timeout, int *uaddr2, int val3);
```

- meaning of various arguments and whether they are used depends on type of operation specified by `futex_op`
- types of futex operations that can be specified by `futex_op`: `FUTEX_WAIT`, `FUTEX_WAKE`, `FUTEX_FD`, `FUTEX_REQUEUE`, `FUTEX_CMP_REQUEUE`, `FUTEX_WAKE_OP`, `FUTEX_WAIT_BITSET`, `FUTEX_WAKE_BITSET`
- option flags that can be included in `futex_op`: `FUTEX_PRIVATE_FLAG`, `FUTEX_CLOCK_REALTIME`
- no glibc wrapper for `futex` system call, so must use `syscall` function

Futex Wait Operation

- wait operation provides means for thread to block waiting on futex
- recall declaration of `futex` system call:

```
int futex(int *uaddr, int futex_op, int val,
  const struct timespec *timeout, int *uaddr2, int val3);
```

- `futex_op`: `FUTEX_WAIT`, possibly with option flags added (e.g., `FUTEX_PRIVATE_FLAG`)
- `uaddr`: address of futex
- `val`: expected value for futex
- `timeout`: if `timeout` not null, `*timeout` specifies maximum amount of time to wait on futex
- other arguments for `futex` system call are ignored
- if futex has expected value, thread is blocked; otherwise, call fails, returning immediately
- returns nonzero value upon failure
- spurious awakenings are permitted

Futex Wake Operation

- wake operation provides means to awaken threads currently blocked waiting on futex
- recall declaration of `futex` system call:

```
int futex(int *uaddr, int futex_op, int val,
  const struct timespec *timeout, int *uaddr2, int val3);
```

- `futex_op`: `FUTEX_WAKE`, possibly with option flags added (e.g., `FUTEX_PRIVATE_FLAG`)
- `uaddr`: address of futex
- `val`: maximum number of waiters to awaken or `INT_MAX` to awaken all waiters
- other arguments for `futex` system call are ignored
- no guarantee as to which waiters are awoken (e.g., waiter with higher scheduling priority not guaranteed to be awoken in preference to waiter with lower priority)

[Example] Futex Wrapper

futex.hpp

```
#include <linux/futex.h>
#include <syscall.h>
#include <sys/time.h>
#include <unistd.h>

inline int futex_wait_private(int* addr, int expected,
  struct timespec* timeout) noexcept {
    return syscall(SYS_futex, addr, FUTEX_WAIT_PRIVATE, expected, timeout,
      nullptr, 0);
}

inline int futex_wake_private(int* addr, int count) noexcept {
    return syscall(SYS_futex, addr, FUTEX_WAKE_PRIVATE, count, nullptr,
      nullptr, 0);
}
```

- `FUTEX_WAIT_PRIVATE` equals `FUTEX_WAIT | FUTEX_PRIVATE_FLAG`
- `FUTEX_WAKE_PRIVATE` equals `FUTEX_WAKE | FUTEX_PRIVATE_FLAG`

[Example] Mutex Version 1: Summary

- first attempt at mutex implementation using futexes
- use single futex
- futex has one of two values:
  - 0: mutex unlocked
  - 1: mutex locked
- in uncontested case:
  - lock operation does not make system call
  - unlock operation always makes unnecessary system call (futex wake)

[Example] Mutex Version 1: Code (1)

mutex1.hpp

```
class mutex {
public:
    mutex() : m_(0) {}
    mutex(const mutex&) = delete;
    mutex& operator=(const mutex&) = delete;
    void lock() noexcept;
    bool try_lock() noexcept;
    void unlock() noexcept;
private:
    static int atomic_cas(int* addr, int expected, int desired) noexcept;
    static void atomic_store(int* addr, int value) noexcept;
    int m_;
    static_assert(__atomic_always_lock_free(sizeof(int), 0));
};
```

[Example] Mutex Version 1: Code (2)

mutex1.cpp

```
#include "futex.hpp"
#include "mutex1.hpp"

inline int mutex::atomic_cas(int* addr, int expected, int desired)
  noexcept {
    __atomic_compare_exchange_n(addr, &expected, desired, false,
      __ATOMIC_ACQUIRE, __ATOMIC_ACQUIRE); // GCC/Clang
    return expected; // return initial value of *ptr
}

inline void mutex::atomic_store(int* addr, int value) noexcept {
    __atomic_store(addr, &value, __ATOMIC_RELEASE); // GCC/Clang
}

void mutex::lock() noexcept {
    while (atomic_cas(&m_, 0, 1) != 0) {futex_wait_private(&m_, 1, nullptr);}
}

bool mutex::try_lock() noexcept {return atomic_cas(&m_, 0, 1) == 0;}

void mutex::unlock() noexcept {
    atomic_store(&m_, 0);
    futex_wake_private(&m_, 1);
}
```

[Example] Mutex Version 2: Summary

- second attempt at mutex implementation using futexes
- single futex used
- futex has one of three values:
  - 0: mutex unlocked
  - 1: mutex locked with no waiters (i.e., no wake needed in unlock)
  - 2: mutex locked with possibly one or more waiters (i.e., wake needed in unlock)
- in uncontested case:
  - lock operation does not make system call
  - unlock operation does not make system call (for futex wake)
- in other words, this mutex implementation only makes system calls in case that mutex is contested

[Example] Mutex Version 2: Code (1)

mutex2.hpp

```
class mutex {
public:
    mutex() : m_(0) {}
    void lock() noexcept;
    void unlock() noexcept;
private:
    static int atomic_cas(int* addr, int expected, int desired) noexcept;
    static int atomic_dec(int* ptr) noexcept;
    int m_;
};
```

mutex2.cpp

```
#include "futex.hpp"
#include "mutex2.hpp"

inline int mutex::atomic_cas(int* addr, int expected, int desired)
  noexcept {
    __atomic_compare_exchange_n(addr, &expected, desired, false,
      __ATOMIC_ACQUIRE, __ATOMIC_ACQUIRE); // GCC/Clang
    return expected; // return initial value of *ptr
}

inline int mutex::atomic_dec(int* ptr) noexcept {
     return __atomic_sub_fetch (ptr, 1, __ATOMIC_RELEASE) + 1; // GCC/Clang
}

void mutex::lock() noexcept {
    if (int c = atomic_cas(&m_, 0, 1); c != 0) {
        do {
            if (c == 2 || atomic_cas(&m_, 1, 2) != 0)
                futex_wait_private(&m_, 2, nullptr);
        } while ((c = atomic_cas(&m_, 0, 2)) != 0);
    }
}

void mutex::unlock() noexcept {
    if (atomic_dec(&m_) != 1) {
        m_ = 0;
        futex_wake_private(&m_, 1);
    }
}
```

Barriers

- **barrier** is synchronization mechanism that allows set of threads to wait until all threads in set have reached particular point in code execution before any thread continues
- barrier maintains two values:
  1. number of threads associated with barrier
  2. number of threads currently waiting on barrier
- wait operation: block thread until all threads associated with barrier are waiting on barrier
- barriers useful for phased computation, in which threads executing same code in parallel must compute one phase of computation before proceeding to next

[Example] Barrier: Summary

- code example illustrates how barrier can be implemented using futexes
- `barrier` class provides futex-based implementation of barrier
- each barrier uses two futexes (i.e., futex-based mutex plus one additional futex)
- sample multithreaded program using `barrier` class also provided to show use of barrier

[Example] Barrier: `barrier` Class (1)

barrier.hpp

```
#include "mutex2.hpp"
#include <climits>

class barrier {
public:
    barrier(unsigned int count) : m_(), event_(0), threshold_(count),
      count_(count) {}
    barrier(const barrier&) = delete;
    barrier& operator=(const barrier&) = delete;
    void wait() noexcept;
private:
    mutex m_;
    unsigned int event_;
    unsigned int threshold_; // total number of threads
    unsigned int count_; // number of threads currently waiting
};
```

- `mutex` is futex-based mutex type from earlier example
- `event_` used for futex operations

[Example] Barrier: barrier Class (2)

barrier.cpp

```
#include <climits>
#include "barrier.hpp"
#include "futex.hpp"

void barrier::wait() noexcept {
    m_.lock();
    if (count_-- > 1) {
        unsigned int e = event_;
        m_.unlock();
        do {
            futex_wait_private(reinterpret_cast<int*>(&event_), e, nullptr);
        } while (event_ == e);
    } else {
        ++event_;
        count_ = threshold_;
        futex_wake_private(reinterpret_cast<int*>(&event_), INT_MAX);
        m_.unlock();
    }
}
```

barrier_app.cpp

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional>
#include "barrier.hpp"

void worker(int instance, barrier& b) {
    for (int i = 0; i < 10; ++i) {
        b.wait();
        std::cout << "1";
        b.wait();
        std::cout << "2";
        b.wait();
        if (!instance) {std::cout << '\n';}
    }
}

int main() {
    int num_threads = 16;
    std::vector<std::jthread> threads;
    barrier b(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(worker, i, std::ref(b));
    }
}
```

References I

1. U. Drepper. Futexes are tricky (version 1.6). Technical report, Red Hat Inc., Nov. 2011. https://www.akkadia.org/drepper/futex.pdf.
2. H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proc. of Ottawa Linux Summit*, 2002. https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf.

Section 2.9

Capabilities

Capabilities

- **capability**: right to be able to perform particular type of privileged operation
- each privileged operation associated with capability
- capabilities can be associated with thread or file
- capability given to thread known as **thread capability**
- each thread in process granted or denied privileges on per-capability basis
- thread can perform particular privileged operation only if it has corresponding capability
- capability associated with file known as **file capability**
- capabilities can be attached to executable program files so that when program run corresponding process will be granted particular capabilities regardless of user who invokes program
- set of capabilities can be represented by bit mask
- currently, about 40 capabilities

- in case of normal user, login shell normally given no capabilities
- in case of superuser, login shell normally given all capabilities
- above two cases are extremes
- many system processes granted only subset of capabilities
- for security reasons, always preferable to grant capability only when strictly needed for particular task

Some Examples of Capabilities

Capability	Description
CAP_CHOWN	can make arbitrary changes to file UIDs and GIDs
CAP_DAC_OVERRIDE	can bypass file read/write/execute permission checks
CAP_SETUID	make arbitrary manipulations of process UIDs (via setuid, setreuid, setresuid, setfsuid)
CAP_SETGID	make arbitrary manipulations of process GIDs and supplementary GID list
CAP_NET_ADMIN	allow various network-related operations to be performed (e.g., interface configuration, modify routing tables)
CAP_NET_BIND_SERVICE	allow binding socket to Internet domain privileged ports (port numbers less than 1024)

Thread Capabilities

- kernel maintains five capability sets for each thread:
  1. **permitted set**: set of capabilities that process permitted to employ (i.e., can be added to effective set)
  2. **effective set**: set of capabilities used by kernel to decide if process allowed to perform each privileged operation
  3. **inheritable set**: set of capabilities that may be propagated to permitted set of program run by exec operation (propagated if not disallowed by file capability)
  4. **bounding set**: set of capabilities that are allowed to be given to process (used to limit capabilities given to process via file capabilities)
  5. **ambient set**: set of capabilities that are preserved across exec of program that is not privileged; ambient set obeys invariant that no capability can ever be ambient if not both permitted and inheritable (automatically adjusted to ensure this)
- permitted, effective, inheritable, bounding, and ambient sets preserved across fork system call

File Capabilities

- file may have no capability information at all or following three capability settings:
  1. effective bit
  2. permitted mask
  3. inheritable mask
- file capabilities provide some control over capabilities of process after exec operation
- **effective bit**: if bit set, then during exec, capabilities that are enabled in process's new permitted set are also enabled in process's new effective set; if bit clear, then after exec, process's new effective set initially empty
- **permitted set**: set of capabilities that may be added to process's permitted set during exec operation, regardless of process's existing capabilities
- **inheritable set**: set masked against process's inheritable set to determine set of capabilities to be enabled in process's permitted set after exec

Thread Capabilities After Exec

- let P_x and P'_x denote capability set x of thread before and after exec, respectively
- let F_x denote file capability set x if file has capabilities and $\emptyset$ otherwise
- let $F_{\text{effective}}$ denote file capability effective bit if file has capabilities and 0 otherwise
- let S_{all} denote set of all existing capabilities
- file said to be privileged if has capabilities or has set-UID or set-GID bit set
- inheritable and bounding capability sets of thread preserved by exec
- aside from some exceptional cases (considered later), capability sets after exec given by:

$$P'_{\text{ambient}} = \begin{cases} \emptyset & \text{file is privileged} \\ P_{\text{ambient}} & \text{otherwise,} \end{cases}$$

$$P'_{\text{permitted}} = (P_{\text{bounding}} \cap F_{\text{permitted}}) \cup (P_{\text{inheritable}} \cap F_{\text{inheritable}}) \cup P'_{\text{ambient}},$$

$$P'_{\text{effective}} = \begin{cases} P'_{\text{permitted}} & F_{\text{effective}} \neq 0 \\ P'_{\text{ambient}} & \text{otherwise} \end{cases}$$

Thread Capabilities After Exec (Continued 1)

- in certain cases involving UID 0, equations from previous slide modified
- let P_{RUID} and P_{EUID} respectively denote RUID and EUID of program run by exec, after any modification made to EUID due to set-UID bit of program file
- if both 1) $P_{\text{RUID}} = 0$ or $P_{\text{EUID}} = 0$, and 2) file does not have capabilities or $P_{\text{RUID}} = 0$ or $P_{\text{EUID}} \neq 0$, then values of $F_{\text{inheritable}}$ and $F_{\text{permitted}}$ replaced by $F_{\text{inheritable}} = S_{\text{all}}$ and $F_{\text{permitted}} = S_{\text{all}}$ (i.e., file inheritable and permitted capability sets ignored)
- if $P_{\text{EUID}} = 0$, then value of $F_{\text{effective}}$ replaced by $F_{\text{effective}} = 1$
- consequently, when thread with nonzero UIDs execs set-UID 0 program file that does not have capabilities, or when thread whose RUID and EUID are 0 execs program, equations simplify to:

$$\begin{aligned} P'_{\text{permitted}} &= P_{\text{bounding}} \cup P_{\text{inheritable}} \\ P'_{\text{effective}} &= P'_{\text{permitted}} \end{aligned}$$

- special treatment of UID 0 can be disabled via securebits mechanism

Thread Capabilities After Exec (Continued 2)

- if file not privileged, equations simplify to:

$$
\begin{aligned}
P'_{\mathsf{ambient}} &= P_{\mathsf{ambient}}; \\
P'_{\mathsf{permitted}} &= P_{\mathsf{ambient}}; \\
P'_{\mathsf{effective}} &= P_{\mathsf{ambient}}
\end{aligned}
$$

- if $F_{\mathsf{effective}} \neq 0$, exec will fail if $F_{\mathsf{permitted}} \not\subset P'_{\mathsf{permitted}}$

Querying and Setting Process/File Capabilities

- can list capabilities of file using `getcap` command
- example output for `getcap /bin/ping`:

```
/bin/ping = cap_net_admin,cap_net_raw+p
```

- can set capabilities of file using `setcap` command
- can get capabilities of process using `getpcaps` command
- example output for `getpcaps $$`:

```
Capabilities for '25526': = cap_chown,cap_dac_override,cap_dac_read_search,
    ↪ cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,
    ↪ cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,
    ↪ cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module
    ↪ ,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,
    ↪ cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,
    ↪ cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write
    ↪ ,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,
    ↪ cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read+i
```

[Example] Unique Handle for cap_t

unique_cap.hpp

```
#include "unique_handle.hpp"
#include <sys/capability.h>

struct cap_uh_policy {
    using handle_type = cap_t;
    static void free(handle_type h) {cap_free(h);}
    static handle_type get_null() {return nullptr;}
    static bool is_null(handle_type h) {return !h;}
};

using unique_cap = unique_handle<cap_uh_policy>;
```

- new policy class for use with `unique_handle` class template (introduced earlier) in order to allow managing `cap_t` objects

[Example] Querying Thread Capabilities: Summary

- code example illustrates use of `prctl` as well as various functionality in `cap` library (e.g., `cap_get_proc` and `cap_get_flag`) for querying thread capabilities
- program queries various thread capabilities and prints results

[Example] Querying Thread Capabilities: Code (1)

getpcap_1.cpp

```
#include <format>
#include <iostream>
#include <sys/capability.h>
#include <sys/prctl.h>
#include <sys/types.h>
#include "unique_cap.hpp"

int64_t getcapmask_cap(cap_t cap, cap_flag_t set) {
    int64_t mask = 0;
    for (int i = 0; i <= CAP_LAST_CAP; ++i) {
        cap_flag_value_t value;
        if (cap_get_flag(cap, i, set, &value)) {return -1;}
        mask = (mask << 1) | value;
    }
    return mask;
}

int64_t getcapmask_prctl(int type) {
    int64_t mask = 0;
    for (int i = 0; i <= CAP_LAST_CAP; ++i) {
        int value = -1;
        if (type == PR_CAPBSET_READ) {value = prctl(PR_CAPBSET_READ, i);}
        else if (type == PR_CAP_AMBIENT)
            {value = prctl(PR_CAP_AMBIENT, PR_CAP_AMBIENT_IS_SET, i, 0, 0);}
        if (value < 0) {return -1;}
        mask = (mask << 1) | value;
    }
    return mask;
}
```

`getpcap_1.cpp` (Continued)

```
int main() {
    unique_cap cap(cap_get_proc());
    if (!cap) {std::cerr << "cap_get_proc failed\n";}
    int64_t all = (static_cast<int64_t>(1) << (CAP_LAST_CAP) + 1) - 1;
    int64_t emask = getcapmask_cap(cap.get(), CAP_EFFECTIVE);
    int64_t imask = getcapmask_cap(cap.get(), CAP_INHERITABLE);
    int64_t pmask = getcapmask_cap(cap.get(), CAP_PERMITTED);
    int64_t bmask = getcapmask_prctl(PR_CAPBSET_READ);
    int64_t amask = getcapmask_prctl(PR_CAP_AMBIENT);
    int sbits = prctl(PR_GET_SECUREBITS);
    if (emask < 0 || imask < 0 || pmask < 0 || bmask < 0 || amask < 0 ||
     sbits < 0)
        {std::cerr << "cannot get capability information\n"; return 1;}
    std::cout << std::format(
      "all capabilities: {:#016x}\n"
      "   permitted set: {:#016x}\n"
      "   effective set: {:#016x}\n"
      " inheritable set: {:#016x}\n"
      "    bounding set: {:#016x}\n"
      "     ambient set: {:#016x}\n"
      "     secure bits: {:#02x}\n",
      all, pmask, emask, imask, bmask, amask, sbits);
}
```

[Example] Querying File Capabilities: Summary

- code example illustrates use of various functionality in `cap` library (e.g., `cap_get_file` and `cap_get_flag`) for querying file capabilities
- command-line argument used to specify pathname of file whose capabilities to be queried
- program queries various capabilities of specified file and prints results

[Example] Querying File Capabilities: Code

getfcap_1.cpp

```
#include <format>
#include <iostream>
#include <sys/capability.h>
#include <sys/types.h>
#include "unique_cap.hpp"

int64_t getcapmask(cap_t cap, cap_flag_t set) {
    uint64_t mask = 0;
    for (int i = 0; i <= CAP_LAST_CAP; ++i) {
        cap_flag_value_t value;
        if (cap_get_flag(cap, i, set, &value)) {return -1;}
        mask = (mask << 1) | value;
    }
    return mask;
}

int main(int argc, char** argv) {
    if (argc < 2) {std::cerr << "bad usage\n"; return 1;}
    unique_cap cap(cap_get_file(argv[1]));
    if (!cap) {
        if (errno == ENODATA) {std::cout << "no capabilities\n"; return 0;}
        else {std::cerr << "cap_get_proc failed\n"; return 1;}
    }
    int64_t pmask = getcapmask(cap.get(), CAP_PERMITTED);
    int64_t emask = getcapmask(cap.get(), CAP_EFFECTIVE);
    int64_t imask = getcapmask(cap.get(), CAP_INHERITABLE);
    if (emask < 0 || imask < 0 || pmask < 0)
        {std::cerr << "cannot get capabilities\n"; return 1;}
    std::cout << std::format(
      "  permitted set: {:#016x}\n"
      "  effective bit: {:#01x}\n"
      "inheritable set: {:#016x}\n",
      pmask, emask, imask);
}
```

Section 2.10

Namespaces

Talks I

1. Michael Kerrisk. Containers Unplugged: Linux Namespaces. NDC TechTown, Kongsberg, Norway, Sept. 4, 2019. Available online at `https://youtu.be/0kJPa-1FuoI`.
2. Michael Kerrisk. Containers Unplugged: Understanding User Namespaces. NDC TechTown, Kongsberg, Norway, Sept. 4, 2019. Available online at `https://youtu.be/73nB9-HYbAI`.

References I

1. Michael Kerrisk. Namespaces in operation, part 1: namespaces overview. https://lwn.net/Articles/531114/, Jan. 4, 2013.
2. Michael Kerrisk. Namespaces in operation, part 2: the namespaces API. https://lwn.net/Articles/531381/, Jan. 8, 2013.
3. Michael Kerrisk. Namespaces in operation, part 3: PID namespaces. https://lwn.net/Articles/531419/, Jan. 16, 2013.
4. Michael Kerrisk. Namespaces in operation, part 4: more on PID namespaces. https://lwn.net/Articles/532748/, Jan. 23, 2013.
5. Michael Kerrisk. Namespaces in operation, part 5: User namespaces. https://lwn.net/Articles/532593/, Feb. 27, 2013.
6. Michael Kerrisk. Namespaces in operation, part 6: more on user namespaces. https://lwn.net/Articles/540087/, March 6, 2013.
7. Jake Edge. Namespaces in operation, part 7: Network namespaces. https://lwn.net/Articles/580893/, Jan. 22, 2014.

8 Lizzie Dixon. Linux Containers in 500 Lines of Code. https://blog.lizzie.io/linux-containers-in-500-loc.html, Oct. 17, 2016.

Section 2.11

Ptrace

Tracing/Controlling Execution of Other Processes

- Linux allows for one process to trace/control execution of another process
- such functionality associated with `ptrace` system call
- process can cause thread in another process to suspend execution after each instruction or when entering/exiting system calls
- process can read/write memory of another process
- process can intercept signals sent to another process
- functionality like that listed above useful, for example, for implementing source-level debuggers
- for security reasons, many restrictions imposed on which processes permitted to control execution of which other processes

Program Tracing

- can trace execution of thread/process using `ptrace` system call
- declaration:

```
long ptrace(enum __ptrace_request request, pid_t pid,
↪ void *addr, void *data);
```

- `request`: type of tracing operation to perform
- `pid`: PID of process being traced
- `addr`: address for trace operation
- `data`: data for trace operation
- meaning of `addr` and `data` depends on type of operation being performed

Ptrace Request Types

Attaching and Detaching Tracee

Request Type	Description
PTRACE_ATTACH	attach to specified process, making it tracee and stopping it
PTRACE_SEIZE	attach to specified process, making it tracee without stopping it
PTRACE_DETACH	restart stopped tracee and detach from it
PTRACE_TRACEME	indicate process to be traced by parent

Resuming Execution of Tracee

Request Type	Description
PTRACE_CONT	restart stopped tracee
PTRACE_SYSCALL	restart stopped tracee but stop at next system-call entry/exit point
PTRACE_SINGLESTEP	restart stopped tracee but stop at next instruction
PTRACE_SYSEMU	continue and stop on entry to next system call, which is not executed
PTRACE_SYSEMU_SINGLESTEP	same as PTRACE_SYSEMU but also single step if not system call

Ptrace Request Types (Continued 1)

Reading and Writing Tracee Memory and User Area

Request Type	Description
PTRACE_PEEKTEXT	read code memory in tracee
PTRACE_POKETEXT	write code memory in tracee
PTRACE_PEEKDATA	read data memory in tracee
PTRACE_POKEDATA	write data memory in tracee
PTRACE_GET_THREAD_AREA	reads thread area for tracee
PTRACE_SET_THREAD_AREA	writes thread area for tracee
PTRACE_PEEKUSER	read user area for tracee
PTRACE_POKEUSER	write user area for tracee

Reading and Writing Processor Registers for Tracee

Request Type	Description
PTRACE_GETREGS	read general-purpose registers of tracee
PTRACE_SETREGS	set general-purpose registers of tracee
PTRACE_GETFPREGS	read floating-point registers of tracee
PTRACE_SETFPREGS	set floating-point registers of tracee
PTRACE_GETREGSET	read registers of tracee
PTRACE_SETREGSET	set registers of tracee

Ptrace Request Types (Continued 2)

Get and Set Signal Mask of Tracee

Request Type	Description
`PTRACE_GETSIGMASK`	get mask of blocked signals in tracee
`PTRACE_SETSIGMASK`	set mask of blocked signals

Query or Set Stop-Related Information

Request Type	Description
`PTRACE_GETSIGINFO`	get information about signal that caused stop
`PTRACE_SETSIGINFO`	set signal information
`PTRACE_PEEKSIGINFO`	retrieve `siginfo_t` structures without removing signals from queue
`PTRACE_GETEVENTMSG`	retrieve message about ptrace event that caused stop
`PTRACE_GET_SYSCALL_INFO`	retrieve information about system call that caused stop

Ptrace Requests (Continued 3)

Other

Request Type	Description
PTRACE_SETOPTIONS	set options for ptrace
PTRACE_LISTEN	restart stopped tracee but prevent it from executing
PTRACE_KILL	send tracee SIGKILL signal to terminate it
PTRACE_INTERRUPT	stop tracee, interrupting system call if necessary
PTRACE_SECCOMP_GET_FILTER	get BPF filters for tracee

Turning Thread into Tracee

- PTRACE_TRACEME request turns calling thread into tracee with parent as tracer
- after PTRACE_TRACEME request, calling thread continues to run (i.e., does not enter ptrace stop)
- if exec performed, SIGSTOP signal sent just before new program starts executing (causing tracee to enter ptrace stop)
- often follow PTRACE_TRACEME request with raise(SIGSTOP) so that parent (which is tracer) can observe signal-delivery stop

Tracing Options

Option	Description
PTRACE_O_EXITKILL	send SIGKILL signal to tracee if tracer exits
PTRACE_O_TRACECLONE	stop tracee at next clone and start tracing newly cloned process
PTRACE_O_TRACEEXEC	stop tracee at next execve
PTRACE_O_TRACEEXIT	stop tracee at exit
PTRACE_O_TRACEFORK	stop tracee at next fork and automatically start tracing newly forked process
PTRACE_O_TRACESYSGOOD	distinguish normal traps from those caused by system calls
PTRACE_O_TRACEVFORK	stop tracee at next vfork and automatically start tracing newly vforked process
PTRACE_O_TRACEVFORKDONE	stop tracee at completion of next vfork
PTRACE_O_TRACESECCOMP	stop tracee when seccomp SECCOMP_RET_TRACE rule triggered
PTRACE_O_SUSPEND_SECCOMP	suspend tracee's seccomp protections

Reading and Writing Memory of Tracee

- PTRACE_PEEKDATA, PTRACE_POKEDATA, PTRACE_PEEKTEXT, PTRACE_POKETEXT access memory one **long** at time
- address being accessed should be suitably aligned for **long**
- although separate operations provided for accessing code and data, can be used interchangeably
- in case of PTRACE_PEEKTEXT and PTRACE_PEEKDATA, can distinguish between read word with value -1 and failure by setting errno to 0 prior to call and checking if errno set if -1 returned
- read and write operations bypass memory protection (e.g., can write to write-protected page)

Program Tracing

- ptrace system call provides means by which one process (called tracer) can observe and control execution of another process (called tracee)
- intended to be used for debuggers and system call tracing
- tracee can be in one of two states:
  1. running (which includes being blocked in system call)
  2. stopped
- when tracee moves from running to stopped state, ptrace stop said to occur
- ptrace stops can be subdivided into four categories:
  1. signal-delivery stops
  2. group stops
  3. ptrace-event stops
  4. syscall stops
- most ptrace commands (all except `PTRACE_ATTACH`, `PTRACE_TRACEME`, `PTRACE_KILL`, and `PTRACE_SETOPTIONS`) require tracee to be in ptrace stop

Program Tracing (Continued)

- tracer can receive ptrace-stop notifications via wait system call (i.e., `wait`, `waitpid`, `waitid`, `wait3`, and `wait4`) in addition to usual child-death notifications
- since (by definition) ptrace stop means tracee has stopped, wait notification for ptrace stop with status `status` always such that `WIFSTOPPED(status)` is true
- type of ptrace stop can be determined based on:
  - value of `WSTOPSIG(status)`
  - value of `status >> 16`
  - result of `PTRACE_GETSIGINFO` request

Signal-Delivery Stop

- for any signal other than `SIGKILL`, kernel selects thread to handle signal
- if thread being traced sent signal, enters signal-delivery stop
- signal not yet delivered to thread and can be suppressed by tracer
- if not suppressed, signal passed to tracee in next ptrace restart request
- if signal blocked, signal-delivery stop does not happen until signal unblocked (except for `SIGSTOP` which cannot be blocked)
- in case of signal-delivery stop, `WSTOPSIG(status)` is signal being delivered
- how to determine if ptrace stop is signal-delivery stop considered later

Signal Injection and Suppression

- after signal-delivery stop, tracer should restart with `ptrace(restart, pid, 0, sig)` where `restart` is ptrace request to resume tracee execution
- if `sig` is 0, no signal delivered; otherwise, signal `sig` injected into thread
- `sig` can be different from `WSTOPSIG(status)`
- signals can only be injected by restarting ptrace command issued after signal-delivery stop
- for this reason, important to distinguish between group stop and signal-delivery stop

Group Stop

- when process receives stopping signal, all threads stop
- if some threads traced, enter group stop
- stopping signal will first cause signal-delivery stop
- only after signal injected by tracer will group stop be initiated on all tracees in process
- in case of group stop, `WSTOPSIG(status)` is stopping signal
- how to determine if ptrace stop is group stop considered later

Ptrace-Event Stops

- ptrace-event stop used to notify certain types of events
- ptrace stop is ptrace-event stop if `WSTOPSIG(status)` is `SIGTRAP` and one of following bits in `status` is set:
  - `PTRACE_EVENT_VFORK`
  - `PTRACE_EVENT_FORK`
  - `PTRACE_EVENT_CLONE`
  - `PTRACE_EVENT_VFORK_DONE`
  - `PTRACE_EVENT_EXEC`
  - `PTRACE_EVENT_EXIT`
  - `PTRACE_EVENT_STOP`
  - `PTRACE_EVENT_SECCOMP`
- particular type ptrace-event stop determined by which of above bits is set

Types of Ptrace-Event Stops

Type	Description
`PTRACE_EVENT_VFORK`	stop (in parent) before return from: 1) vfork or 2) clone with `CLONE_VFORK` flag
`PTRACE_EVENT_FORK`	stop (in parent) before return from: 1) fork or 2) clone with exit signal set to SIGCHLD
`PTRACE_EVENT_CLONE`	stop (in parent) before return from clone
`PTRACE_EVENT_VFORK_DONE`	stop (in parent) after child unblocks tracee by exiting/execing before return from: 1) vfork or 2) clone with `CLONE_VFORK` flag
`PTRACE_EVENT_EXEC`	stop before return from execve
`PTRACE_EVENT_EXIT`	stop before exit (including `exit_group`, signal death, exit caused by execve in multithreaded process)
`PTRACE_EVENT_STOP`	stop induced by `PTRACE_INTERRUPT` command, group stop, initial ptrace-stop when new child attached using `PTRACE_SEIZE`
`PTRACE_EVENT_SECCOMP`	stop triggered by seccomp rule on tracee syscall entry when `PTRACE_O_TRACESECCOMP` flag set

Syscall Stops

- when tracee enters or exits system call, tracee can enter syscall stop
- two types of syscall stops:
  1. syscall enter (tracee about to enter system call)
  2. syscall exit (tracee about to leave system call)
- if tracee restarted by `PTRACE_SYSCALL` or `PTRACE_SYSEMU`, tracee enters syscall-enter stop just prior to entering any system call
- if tracee in syscall-enter stop restarted with `PTRACE_SYSCALL`, tracee enters syscall-exit stop just after system call completes
- signal-delivery stop will never occur between syscall-enter and syscall-exit stops
- ptrace event stops may occur between syscall-enter and syscall-exit stops
- if tracee in syscall-enter stop restarted with `PTRACE_SYSEMU`, no syscall-exit stop occurs
- particular system call can be determined by inspecting processor registers

Syscall Stops (Continued)

- syscall-enter stop always followed by syscall-exit stop, ptrace-event stop, or tracee's death
- seccomp ptrace-event stops can cause syscall-exit stop without preceding syscall-entry stop
- syscall-enter and syscall-exit stops can be distinguished by tracking which syscall stop last occurred
- how to determine if ptrace stop is syscall stop to be considered shortly

Distinguishing Signal-Delivery, Group, and Syscall Stops

- if ptrace stop not ptrace-event stop, following approach can be used to distinguish between remaining type of stops
- if `WSTOPSIG(status)` is `SIGTRAP`, either signal-delivery stop or syscall stop occurred
- if `WSTOPSIG(status)` is stopping signal (i.e., `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, and `SIGTTOU`), either signal-delivery stop or group stop occurred
- otherwise, signal-delivery stop occurred
- group stops can be distinguished from signal-delivery stops for stopping signals by using `PTRACE_GETSIGINFO` request
- if `PTRACE_GETSIGINFO` results in `EINVAL`, group stop
- syscall-stops can be distinguished from signal-delivery stops with `SIGTRAP` by querying `PTRACE_GETSIGINFO` or using `PTRACE_O_TRACESYSGOOD` ptrace option (with latter being more efficient)

Additional Remarks

- if `PTRACE_O_TRACESYSGOOD` set, signal number ORed with `0x80` in case of syscall-stop, allowing distinction to be made between `SIGTRAP` signal for process and syscall-stop
- syscall-enter stop has `-ENOSYS` in register used for return value (rax on x86-64)
- syscall-exit stop has actual return value in register
- since system call never returns `-ENOSYS`, can distinguish syscall-enter stop from syscall-exit stop based on this register
- ptrace saves original rax register to `orig_rax`
- at syscall-exit, `orig_rax` has original value on entry to system call
- therefore, can tell which system call exiting

Additional Remarks (Continued 1)

- restriction on what thread can trace another child/descendant same UID `CAP_SYS_ADMIN`
- particular setting can be found in `/proc/sys/kernel/yama/ptrace_scope`
- in case of many Linux distributions, kernel is configured by default to prevent any process from calling ptrace on another process that it did not create (e.g., via fork)
- only one tracer of thread/process at any given time
- to trace children, can use `PTRACE_O_TRACEFORK`, `PTRACE_O_TRACECLONE`, and `PTRACE_O_TRACEVFORK` options
- options inherited by new tracees that are created and auto-attached via active `PTRACE_O_TRACEFORK`, `PTRACE_O_TRACEVFORK`, and `PTRACE_O_TRACECLONE` settings

[Example] Changing Tracee Memory: Summary

- code example illustrates use of `ptrace` system call to read/write memory of child process
- parent process initializes integer variables `x` and `y`
- then parent process creates child process via `fork`
- before requesting to be traced, child process sets `x` and `y` to different values (from those set by parent process)
- then, child process requests to be traced via `PTRACE_TRACEME` request of `ptrace` (which results in child process being stopped)
- before parent process resumes execution of child process, parent process prints value of `x` from child's memory and also changes value of `y` in child's memory
- then, parent process allows child process to continue execution
- after being resumed, child process prints value of `y` to show that parent did, in fact, change value of `y` in child's memory

[Example] Changing Tracee Memory: Code

ptrace_2.cpp

```
#include <format>
#include <iostream>
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    long x = -1;
    long y = -1;
    if (pid_t child_pid = fork(); child_pid > 0) {
        int status;
        if (waitpid(child_pid, &status, 0) != child_pid) {return 1;}
        if (!WIFSTOPPED(status)) {return 1;}
        if (ptrace(PTRACE_SETOPTIONS, child_pid, 0, PTRACE_O_EXITKILL))
          {return 1;}
        errno = 0;
        long w = ptrace(PTRACE_PEEKDATA, child_pid, &x, 0);
        if (w == -1 && errno) {return 1;}
        if (ptrace(PTRACE_POKEDATA, child_pid, &y, w)) {return 1;}
        if (ptrace(PTRACE_CONT, child_pid, 0, 0)) {return 1;}
    } else if (child_pid == 0) {
        x = 42;
        y = 0;
        if (ptrace(PTRACE_TRACEME, 0, 0, 0)) {return 1;}
        if (raise(SIGSTOP)) {return 1;}
        std::cout << std::format("value is {}\n", y);
    } else {return 1;}
}
```

[Example] Single Stepping: Summary

- code example illustrates use of `PTRACE_SINGLESTEP` request of `ptrace` system call
- program takes command-line arguments that specify another program to run and zero or more arguments for that other program
- process creates child process (via `fork`) so that child can run other program (via `execve`)
- parent process then waits for child to stop (due to request to be traced)
- before calling `execve`, child process asks to be traced via `PTRACE_TRACEME` request of `ptrace`
- parent process loops issuing `PTRACE_SINGLESTEP` request of `ptrace` to stop child execution after each instruction, at which point value of child's instruction pointer queried (via `ptrace`) and printed
- after child process terminates, total number of instructions executed by child is printed
- for example, to single step through execution of `/bin/true`, use command:

```
ptrace_3 /bin/true
```

[Example] Single Stepping: Code

ptrace_3.cpp

```
#include <chrono>
#include <format>
#include <iostream>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/user.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char** argv) {
    if (argc < 2) {std::cerr << "no program specified\n"; return 1;}
    if (pid_t child_pid = fork(); child_pid > 0) {
        struct user_regs_struct regs;
        unsigned long long count = 0;
        auto start_time = std::chrono::high_resolution_clock::now();
        for (;;) {
            int status;
            if (waitpid(child_pid, &status, 0) < 0) {return 1;}
            if (WIFEXITED(status)) {break;}
            if (ptrace(PTRACE_GETREGS, child_pid, nullptr, &regs) < 0) {return 1;}
            std::cerr << std::format("rip: {:#x}\n", regs.rip);
            if (ptrace(PTRACE_SINGLESTEP, child_pid, nullptr, nullptr) < 0) {return 1;}
            ++count;
        }
        double elapsed_time = std::chrono::duration<double>(
          std::chrono::high_resolution_clock::now() - start_time).count();
        std::cerr << std::format("instruction count: {}\n"
          "elapsed time: {}\n"
          "instructions/second: {}\n",
          count, elapsed_time, count / elapsed_time);
    } else if (child_pid == 0) {
        if (ptrace(PTRACE_TRACEME, 0, nullptr, nullptr) < 0) {return 1;}
        if (execve(argv[1], &argv[1], environ)) {return 1;}
    } else {return 1;}
}
```

[Example] System Call Tracer: Summary

- code example illustrates use of `PTRACE_SYSCALL` request of `ptrace` system call
- program takes command-line arguments that specify another program to run and zero or more arguments for that other program
- process creates child process (via `fork`) so that child can run other program (via `execve`)
- parent process then waits for child to stop (due to request to be traced)
- before calling `execve`, child process asks to be traced via `PTRACE_TRACEME` request of `ptrace`
- parent process loops issuing `PTRACE_SYSCALL` request of `ptrace` to stop child execution at each system call, at which point information about type of system call recorded and printed
- after child process terminates, which system calls used by child printed (along with counts)
- for example, to process execution of "`/bin/ls /`", use command:

```
ptrace_1 /bin/ls /
```

[Example] System Call Tracer: Code (1)

ptrace_1.cpp (Continued)

```
#include <format>
#include <iostream>
#include <map>
#include <stdexcept>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/user.h>
#include <sys/wait.h>
#include <syscall.h>
#include <unistd.h>
#include "syscall_names.hpp"
```

[Example] System Call Tracer: Code (2)

`ptrace_1.cpp` (Continued)

```
void parent(int child_pid) {
    std::map<long, size_t> syscall_counts;
    int status;
    if (waitpid(child_pid, &status, 0) < 0) {throw std::runtime_error("waitpid failed");}
    if (!WIFSTOPPED(status)) {throw std::runtime_error("unexpected tracee state");}
    if (ptrace(PTRACE_SETOPTIONS, child_pid, 0, PTRACE_O_EXITKILL))
      {throw std::runtime_error("ptrace failed");}
    for (;;) {
        if (ptrace(PTRACE_SYSCALL, child_pid, 0, 0) < 0)
          {throw std::runtime_error("ptrace failed");}
        if (waitpid(child_pid, &status, 0) < 0)
          {throw std::runtime_error("waitpid failed");}
        if (!WIFSTOPPED(status)) {throw std::runtime_error("unexpected tracee state");}
        struct user_regs_struct regs;
        if (ptrace(PTRACE_GETREGS, child_pid, 0, &regs) < 0)
          {throw std::runtime_error("cannot get registers");}
        long syscall = regs.orig_rax;
        syscall_counts[syscall]++;
        std::cout << std::format("entering syscall {}\n", syscall_names[syscall]);
        if (ptrace(PTRACE_SYSCALL, child_pid, 0, 0) < 0)
          {throw std::runtime_error("ptrace failed");}
        if (waitpid(child_pid, &status, 0) < 0)
          {throw std::runtime_error("waitpid failed");}
        if (WIFEXITED(status)) {break;}
        if (!WIFSTOPPED(status)) {throw std::runtime_error("unexpected tracee state");}
    }
    std::cout << "total system call counts:\n";
    for (auto [k, v] : syscall_counts)
      {std::cout << std::format("{:9d} {:s}\n", v, syscall_names[k]);}
}
```

`ptrace_1.cpp` (Continued)

```
void child(int argc, char** argv) {
    ptrace(PTRACE_TRACEME, 0, 0, 0);
    if (execve(argv[1], &argv[1], environ) < 0)
      {throw std::runtime_error("execve failed");}
}

int main(int argc, char** argv) try {
    if (argc <= 1) {throw std::runtime_error("invalid usage");}
    if (pid_t pid = fork(); pid > 0) {parent(pid);}
    else if (pid == 0) {child(argc, argv);}
    else {throw std::runtime_error("fork failed");}
} catch(const std::exception& e) {
    std::cerr << "fatal error: " << e.what() << '\n';
}
```

References I

1. Pradeep Padala. Playing with ptrace, Part I. Linux Journal, https://www.linuxjournal.com/article/6100, Oct. 31, 2002.
2. Pradeep Padala. Playing with ptrace, Part II. Linux Journal, https://www.linuxjournal.com/article/6210, Nov. 30, 2002.
3. Eli Bendersky. How debuggers work: Part 1 - Basics. https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1, Jan. 23, 2011.
4. Eli Bendersky. How debuggers work: Part 2 - Breakpoints. https://eli.thegreenplace.net/2011/01/27/how-debuggers-work-part-2-breakpoints, Jan. 27, 2011.
5. Eli Bendersky. How debuggers work: Part 3 - Debugging information. https://eli.thegreenplace.net/2011/02/07/how-debuggers-work-part-3-debugging-information, Feb. 7, 2011.

Talks I

1. Michael Kerrisk. Strace: Monitoring The Kernel-User-Space Conversation. NDC TechTown, Kongsberg, Norway, Aug. 29, 2018. Available online at https://youtu.be/oFt6V56BOlo.
2. Greg Law and Dewang Li. Modern Linux C++ Debugging Tools - Under the Covers. CppCon, Aurora, CO, USA, Sept. 20, 2019. Available online at https://youtu.be/WoRmXjVxuFQ.

Section 2.12

Seccomp

Talks I

1. Paul Moore and Tom Hromatka. The Why and How of libseccomp. Linux Security Summit North America, San Diego, CA, USA, Aug. 19, 2019. Available online at https://youtu.be/6lRHK_LLUGI.
2. Tycho Andersen. Forwarding syscalls to userspace. linux.conf.au, Christchurch, New Zealand, Jan. 22, 2019. Available online at https://youtu.be/sqvF_Mdtzgg.

Section 2.13

Shared Libraries and Dynamic Linking and Loading

[Example] Greet: Summary

- code example illustrates use of dynamic loading
- shared library provides `greet` function that prints greeting message
- application program invokes `greet` function and exits
- two versions of application program provided, one using dynamic loading of library and one not
- application using dynamic loading loads shared library and then resolves `greet` symbol in order to locate function to call

[Example] Greet: CMakeLists.txt

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.14)

project(greet LANGUAGES CXX)

set(CMAKE_VERBOSE_MAKEFILE true)
set(CMAKE_CXX_STANDARD 20)

add_library(greet_static STATIC greet.cpp)
set_target_properties(greet_static PROPERTIES OUTPUT_NAME greet)

add_library(greet_shared SHARED greet.cpp)
set_target_properties(greet_shared PROPERTIES OUTPUT_NAME greet)

add_executable(app_static app.cpp)
target_link_libraries(app_static PUBLIC greet_static)

add_executable(app_shared app.cpp)
target_link_libraries(app_shared PUBLIC greet_shared)

add_executable(app_dl app_dl.cpp)
target_link_libraries(app_dl PUBLIC dl)
```

[Example] Greet: Library

greet.hpp

```
bool greet();
```

greet.cpp

```
#include <iostream>

bool greet() {
    return bool(std::cout << "Hello, World\n" << std::flush);
}
```

[Example] Greet: Application Without Dynamic Loading

app.cpp

```
#include "greet.hpp"

int main() {
    return greet() ? 0 : 1;
}
```

[Example] Greet: Application With Dynamic Loading

app_dl.cpp

```
#include <format>
#include <iostream>
#include <dlfcn.h>
#include <stdlib.h>

int main(void) {

    using greet_func_t = bool (*)(void);

    constexpr char lib[] = "libgreet.so";
    const char *error;
    void *module;
    greet_func_t greet;

    module = dlopen(lib, RTLD_LAZY);
    if (!module) {
        std::cerr << std::format("cannot open {}: {}\n", lib, dlerror());
        return 1;
    }

    dlerror();
    greet = reinterpret_cast<greet_func_t>(dlsym(module, "_Z5greetv"));
    if ((error = dlerror())) {
        std::cerr << std::format("cannot find greet: {}\n", error);
        return 1;
    }

    bool status = greet();

    dlclose(module);
    return status ? 0 : 1;
}
```

[Example] PIE: Summary

- consider simple C++ program that prints address of several variables and functions in order to demonstrate address-space layout randomization (ASLR)
- CMake used to build code
- print value of global function `func`, global variable `var`, and local variable `i` in `main` function
- in non-PIE case, address of `func` and `var` do not change across multiple invocations of program
- in PIE case, addresses of `func` and `var` change across multiple invocations of program

[Example] PIE: CMakeLists File

CMakeLists.txt

```
# Needs CMake 3.14 for CMP0083 NEW
cmake_minimum_required(VERSION 3.14)
project(aslr_demo LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_VERBOSE_MAKEFILE true)

include(CheckPIESupported)
check_pie_supported()
if(NOT CMAKE_CXX_LINK_PIE_SUPPORTED)
    message(FATAL_ERROR "PIE is not supported\n")
endif()

add_executable(aslr_nopie aslr.cpp)
set_property(TARGET aslr_nopie PROPERTY POSITION_INDEPENDENT_CODE false)
add_executable(aslr_pie aslr.cpp)
set_property(TARGET aslr_pie PROPERTY POSITION_INDEPENDENT_CODE true)
```

aslr.cpp

```
#include <iostream>
#include <iomanip>
#include <cstdint>

int var = 42;

int func() {return 42;}

int main() {
    int i;
    std::cout << std::hex << std::setfill('0')
      << " addr(var): " << std::setw(16)
      << reinterpret_cast<uintptr_t*>(&var) << '\n'
      << "addr(func): " << std::setw(16)
      << reinterpret_cast<uintptr_t>(func) << '\n'
      << "   addr(i): " << std::setw(16)
      << reinterpret_cast<uintptr_t>(&i) << '\n'
      << "addr(var) - addr(func): " << std::setw(16)
      << reinterpret_cast<uintptr_t>(&var) -
      reinterpret_cast<uintptr_t>(&func) << '\n';
}
```

Part 3

Other Topics

Section 3.1

Assembly Language

Section 3.1.1

Basic Computer Architecture

Processors

- **core** is independent processing unit that reads and executes program instructions, and consists of:
  - registers
  - arithmetic logic unit (ALU)
  - control unit
  - usually cache
- **processor** is computing element that consists of:
  - one or more cores
  - external bus interface
  - possibly shared cache
- **thread** is sequence of instructions (which can be executed by core)

Registers

- **register**: very small memory (typically 32 or 64 bits in size) located on processor itself
- number of registers and their sizes varies from one processor architecture to another
- access to data contained in registers extremely fast, since registers inside processor
- modern processor architectures tend to use registers to hold operands for most or all operations performed

Code Execution

- each possible instruction represented by sequence of one or more bytes called **opcode** (which stands for operation code)
- **instruction pointer** is special register that holds address of next instruction to execute
- code execution consists of following steps repeated in infinite loop:
  1. processor reads opcode from address pointed to by instruction pointer
  2. operation specified by opcode is performed
  3. based on instruction performed, instruction pointer updated
- normally, instruction pointer updated to point to address immediately following end of opcode of instruction just executed
- some instructions, however, can cause code execution to deviate from this normal linear path
- other than loads and stores, operands for instructions usually taken from registers (or instruction opcode itself)

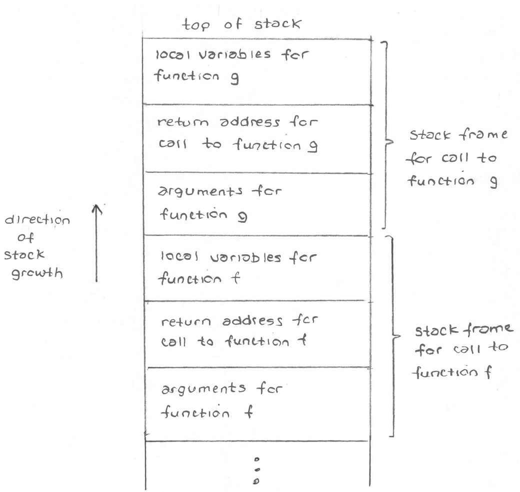
Types of Instructions

- memory operations:
  - load: read value from memory (or immediate value) into register
  - store: write value in register (or immediate value) to memory
- arithmetic and logic operations:
  - integer arithmetic (e.g., add, subtract, multiply, divide, increment, decrement)
  - bitwise-logic and bit-shifting operations for integers (e.g., AND, OR, NOT, XOR, logical shift, arithmetic shift)
  - comparison operations
  - floating-point arithmetic (e.g., add, subtract, multiply, divide, square root)
- control-flow operations:
  - conditional and unconditional branches
  - subroutine call and return
  - system call and return
- coprocessor instructions:
  - operations to move data to and from coprocessor
  - coprocessor operations (e.g., floating-point arithmetic operations for math coprocessor)

Call Stack

- **call stack** (often simply called **stack**) is stack data structure used for managing function calls
- stack pointer holds address of element at top of stack
- whether stack grows upwards or downwards in memory, depends on particular processor architecture
- call stack comprised of entries known as stack frames
- each active function call has corresponding stack frame on call stack
- stack frame for function call stores:
  - return address for function call
  - arguments to function call
  - local variables for function
  - saved copies of registers modified by function
  - return value produced by function call in cases where value not returned in register

Stack Example



- some function (whose stack frame is not shown) calls function `f` which then calls function `g`

Status Register

- status register holds numerous condition/status flags and control settings
- many processor instructions can set condition/status flags
- particular state stored in status register will depend on particular processor architecture
- some common condition flags include:
  - zero (result zero)
  - carry/borrow (result generated carry or borrow)
  - negative (result negative)
  - parity (result has even/odd parity)
  - overflow (operation caused overflow)
- for example, arithmetic operation that compares two values (i.e., computes their difference) would set zero flag in status register if result of comparison is zero
- conditional branches use condition flags in status register to decide if branch should be taken

Calling Conventions

- in order for mechanics of function calls to work correctly, caller and callee must agree on numerous things:
  - how are arguments passed to functions (e.g., on stack or in registers, where on stack, in which registers)
  - how return value propagated from callee to caller
  - what registers must be preserved by callee (i.e., callee-saved registers)
  - what registers must be saved by caller because they are allowed to be changed by callee (i.e., caller-saved registers)
- particular set of choices made in regard to above issues referred to as **calling conventions**

Endianness

- when storing multibyte values in memory, more than one choice possible on how to order bytes in memory
- **big endian**: multibyte values stored in order of most significant to least significant byte (i.e., big end first)
- **little endian**: multibyte values stored in order of least significant to most significant byte (i.e., little end first)
- consider 32-bit integer value DEADBEEF (in hexadecimal), which requires 4 bytes of storage:

Big Endian

Address	Value
i	DE
$i+1$	AD
$i+2$	BE
$i+3$	EF

Little Endian

Address	Value
i	EF
$i+1$	BE
$i+2$	AD
$i+3$	DE

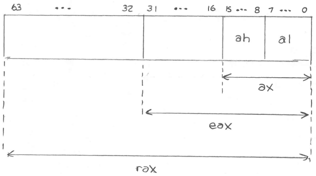
Section 3.1.2

x86-64 Architecture

x86-64 Architecture

- 16 64-bit general-purposes registers (including stack pointer):
  - rax, rbx, rcx, rdx, rbp, rsi, rdi, rsp, r8, r9, r10, r11, r12, r13, r14, r15
- rsp is stack pointer
- rbp normally used as frame pointer
- 16 128-bit SSE (vector) registers:
  - xmm0 to xmm15
- 64-bit instruction pointer register rip
- status/control register
- 6 segment registers:
  - cs, ds, es, ss, fs, gs
- uses little-endian byte ordering

- information in 64-bit general-purpose register can be accessed as:
    - 64-bit register in its entirety
    - 32-bit register comprised of 32 least significant bits (i.e., bits 0–31) of 64-bit register
    - 16-bit register comprised of 16 least significant bits (i.e., bits 0–15) of 64-bit register
    - 8-bit register comprised of 8 least significant bits (i.e., bits 0–7) of 64-bit register
- in some special cases, can access bits 8–15 of 64-bit register as 8-bit register as well
- for example, in case of (64-bit) rax register:



General-Purpose Registers (Continued)

64-Bit Register	Bits 0–31	Bits 0–15	Bits 15–8	Bits 0–7
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rbp	ebp	bp	—	bpl
rsi	esi	si	—	sil
rdi	edi	di	—	dil
rsp	esp	sp	—	spl
r8	r8d	r8w	—	r8b
r9	r9d	r9w	—	r9b
r10	r10d	r10w	—	r10b
r11	r11d	r11w	—	r11b
r12	r12d	r12w	—	r12b
r13	r13d	r13w	—	r13b
r14	r14d	r14w	—	r14b
r15	r15d	r15w	—	r15b

Flags Register

- 64-bit rflags register holds condition/status flags and control information
- eflags refers to 32 least-significant bits of rflags
- flags refers to 16 least-significant bits of rflags
- cannot directly read or write flags register as whole
- flags register can be pushed on and popped off stack (and value on stack can be modified)
- some arithmetic and logical operations set one or more flags
- conditional branch instructions decide if branch should take place based on value of one or more flags

Some Condition Flags in Flags Register

Flag	Description
carry flag (CF)	set on high-order bit carry or borrow; clear otherwise
parity flag (PF)	set if low-order eight bits of result contain even number of one bits; cleared otherwise
zero flag (ZF)	set if result zero; cleared otherwise
sign flag (SF)	set equal to high-order bit of result (0 if positive; 1 if negative)
overflow flag (OF)	set if result too large in magnitude to fit in destination operand; clear otherwise

Conditional Branches

Instruction	Description	Signedness	Flags
jo	jump if overflow	—	OF = 1
jno	jump if not overflow	—	OF = 0
js	jump if sign	—	SF = 1
jns	jump if not sign	—	SF = 0
je jz	jump if equal jump if zero	—	ZF = 1
jne jnz	jump if not equal jump if not zero	—	ZF = 0
jb jnae jc	jump if below jump if not above or equal jump if carry	unsigned	CF = 1
jnb jae jnc	jump if not below jump if above or equal jump if not carry	unsigned	CF = 0
jbe jna	jump if below or equal jump if not above	unsigned	CF = 1 or ZF = 1
ja jnbe	jump if above jump if not below or equal	unsigned	CF = 0 and ZF = 0
jl jnge	jump if less jump if not greater or equal	signed	SF != OF
jge jnl	jump if greater or equal jump if not less	signed	SF = OF
jle jng	jump if less or equal jump if not greater	signed	ZF = 1 or SF != OF
jg jnle	jump if greater jump if not less or equal	signed	ZF = 0 and SF = OF
jp jpe	jump if parity jump if parity even		PF = 1
jnp jpo	jump if not parity jump if parity odd		PF = 0
jcxz jecxz	jump if cx is zero jump if ecx is zero		CX = 0 ECX = 0

Function Calls

- function invoked using `call` instruction
- call instruction takes operand that specifies where start of function to be called resides in memory
- address of next instruction after call instruction pushed on stack; then jumps to start address for function
- return from function using `ret` instruction
- pops new value for instruction pointer from stack

Calling Conventions

- callee must preserve values of rbx, rbp, r12, r13, r14, and r15 registers (i.e., value at exit from callee must be same as value at entry)
- all other registers must be saved by caller if it wishes to preserve their values
- first six integer or pointer arguments passed (in order) in registers:
  - rdi, rsi, rdx, rcx, r8d, and r9d
- first eight floating-point arguments passed (in order) in registers:
  - xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, and xmm7
- other types of arguments and additional integer/pointer/floating-point arguments passed on stack, pushed in right-to-left order
- return value placed in:
  - rax for integer/pointer value
  - xmm0 for floating-point value
  - at location specified by hidden function parameter for other types
- in C++, **`this`** is first argument

Calling Conventions (Continued)

- stack pointer must have 16-byte alignment prior to making function call
- implies that upon entry to function, stack pointer minus 8 always 16-byte aligned

System Calls

- service provided by operating system accessed via system call
- system call performed via `syscall` instruction
- `syscall` instruction allows code to transition from non-privileged execution in user space to privileged execution in kernel space

System-Call Calling Conventions

- system call allowed to modify rcx and r11 as well as rax (used for return value) but other registers preserved
- number of system call passed in rax
- system calls limited to six arguments, which are always integers or pointers
- all arguments passed in registers (i.e., stack not used)
- arguments assigned to registers in (left-to-right) order as follows:
  - rdi, rsi, rdx, r10, r8, and r9
- note that above policy for assigning arguments to registers differs from policy for user-level code
- return value between `-4095` and `-1` indicates error that corresponds to `-errno`
- return value placed in rax

[Example] True Program: Source Code (C++ and Assembly Versions)

`true_0.cpp` (C++ version)

```
#include <cstdlib>
int main() {
    std::exit(0);
    // not reached
}
```

`true_1.s` (functionally equivalent to C++ version, but without C++ runtime)

```
    .set SYS_exit, 60 # system call number for exit
    .text
    .globl _start
# _start is linker's default entry point for program
_start:
    # exit(0)
    mov $0, %edi # note: xor %edi, %edi would have shorter opcode
    mov $SYS_exit, %rax
    syscall
    # not reached
```

output of `objdump -d true_1`

```

true_1:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:   bf 00 00 00 00          mov    $0x0,%edi
  401005:   48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  40100c:   0f 05                   syscall
```

[Example] Maximum Function

max_0.cpp

```cpp
int max(int m, int n) noexcept {return (m > n) ? m : n;}
```

max_1.s

```
# note: sizeof(int) is 4
    .text
    .globl _Z3maxii
# int max(int, int)
_Z3maxii:
    cmp %edi, %esi # esi - edi = ?
    jle .L0 # branch if esi - edi <= 0 (i.e., esi <= edi)
    mov %esi, %eax # set return value to esi
    jmp .L1
.L0:
    mov %edi, %eax # set return value to edi
.L1:
    ret
```

[Example] Factorial Function

factorial_0.cpp

```cpp
unsigned long long factorial(unsigned long long n) noexcept {
    unsigned long long result = 1;
    for (; n > 1; result *= n, --n) {}
    return result;
}
```

factorial_1.s

```
# note: sizeof(unsigned long long) is 8
    .text
    .globl _Z9factorialy
# unsigned long long factorial(unsigned long long)
_Z9factorialy:
    # note: clears upper 32 bits of eax
    mov $1, %eax
.L_loop_start:
    cmp $1, %rdi # rdi - 1 = ?
    jbe .L_loop_end # branch if rdi - 1 <= 0 (i.e., rdi <= 1)
    # note: signed multiply gives correct lower 64-bits of result
    imul %rdi, %rax
    sub $1, %rdi
    jmp .L_loop_start
.L_loop_end:
    ret
```

[Example] Hamming-Weight Function

hamming_weight_0.cpp

```
unsigned int hamming_weight(unsigned int n) {
    unsigned int count = 0;
    for (; n; count += n & 1, n >>= 1) {}
    return count;
}
```

hamming_weight_1.s

```
# note: sizeof(unsigned int) is 4
.text
.globl _Z14hamming_weightj
# unsigned int hamming_weight(unsigned int)
_Z14hamming_weightj:
    # note: xor %eax, %eax has shorter opcode than mov $0, %eax
    xorl %eax, %eax
.L_loop_start:
    # note: test %edi, %edi has shorter opcode than cmp $0, %edi
    test %edi, %edi # edi & edi = edi = ?
    jz .L_loop_end # branch if edi == 0
    mov %edi, %edx
    and $1, %edx
    add %edx, %eax
    shr %edi
    jmp .L_loop_start
.L_loop_end:
    ret
```

[Example] Hello World: Assembly Code

hello_2.s

```
    .text
    .globl _start
_start:
    # n = write(1, hello, hello_len)
    mov $1, %rdi
    mov $hello, %rsi
    mov $hello_len, %rdx
    call write
    # exit(n == hello_len ? 0 : 1)
    mov $1, %rdi
    cmp $hello_len, %rax # rax - $hello_len = ?
    jne .L0 # branch if rax - $hello_len != 0 (i.e., rax != $hello_len)
    mov $0, %rdi
.L0:
    call exit # does not return
write:
    .set SYS_write, 1 # system call number for write
    mov $SYS_write, %rax
    syscall
    ret
exit:
    .set SYS_exit, 60 # system call number for exit
    mov $SYS_exit, %rax
    syscall # does not return
    .data
hello:
    .ascii "Hello, World!\n"
    .set hello_len, . - hello
```

[Example] Hello World: Disassembled Code

output of `objdump -d hello_2`

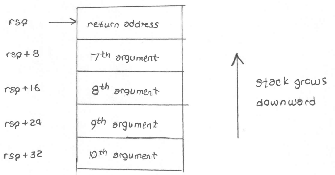
```

hello_2:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:   48 c7 c7 01 00 00 00    mov    $0x1,%rdi
  401007:   48 c7 c6 00 20 40 00    mov    $0x402000,%rsi
  40100e:   48 c7 c2 0e 00 00 00    mov    $0xe,%rdx
  401015:   e8 1b 00 00 00          callq  401035 <write>
  40101a:   48 c7 c7 01 00 00 00    mov    $0x1,%rdi
  401021:   48 3d 0e 00 00 00       cmp    $0xe,%rax
  401027:   75 07                   jne    401030 <_start+0x30>
  401029:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
  401030:   e8 0a 00 00 00          callq  40103f <exit>

0000000000401035 <write>:
  401035:   48 c7 c0 01 00 00 00    mov    $0x1,%rax
  40103c:   0f 05                   syscall
  40103e:   c3                      retq

000000000040103f <exit>:
  40103f:   48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  401046:   0f 05                   syscall
```

addints_0_a.cpp

```
int add(int a0, int a1, int a2, int a3, int a4, int a5, int a6,
  int a7, int a8, int a9) {
    return a0 + a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9;
}
```

addints_1.s

```
    .text
    .globl _Z3addiiiiiiiiii
# int add(int a0, int a1, int a2, int a3,
#   int a4, int a5, int a6, int a7, int a8,
#   int a9)
_Z3addiiiiiiiiii:
    mov %edi, %eax # result = a0
    add %esi, %eax # result += a1
    add %edx, %eax # result += a2
    add %ecx, %eax # result += a3
    add %r8d, %eax # result += a4
    add %r9d, %eax # result += a5
    add 8(%rsp), %eax # result += a6
    add 16(%rsp), %eax # result += a7
    add 24(%rsp), %eax # result += a8
    add 32(%rsp), %eax # result += a9
    ret
```



[Example] Add Integers: Caller

addints_0_b.cpp

```
int do_add() {
    return add(1, 2, 3, 4, 5, -5, -4, -3, -2, -1);
}
```

addints_2.s

```
    .text
    .globl _Z6do_addv
_Z6do_addv:
    # int add(1, 2, 3, 4, 5, -5, -4, -3, -2, -1)
    # note: x86-64 cannot push 32-bit register
    pushq $-1 # 10th argument
    pushq $-2 # 9th argument
    pushq $-3 # 8th argument
    pushq $-4 # 7th argument
    mov $-5, %r9d # 6th argument
    mov $5, %r8d # 5th argument
    mov $4, %ecx # 4th argument
    mov $3, %edx # 3rd argument
    mov $2, %esi # 2nd argument
    mov $1, %edi # 1st argument
    call _Z3addiiiiiiiiii
    add $32, %rsp # remove arguments from stack
    ret
```

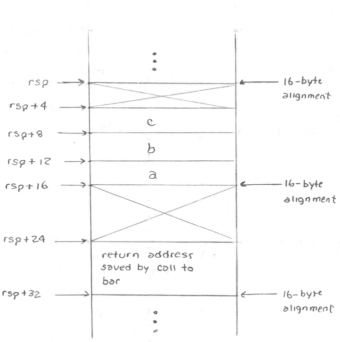
[Example] Local Variables

example1_0.cpp

```
void foo(int& i, int& j, int& k);

int bar(int a, int b, int c) {
    foo(a, b, c);
    return a + b + c;
}
```

example1_1.s

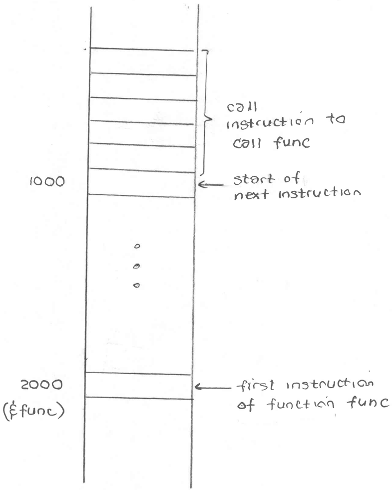
```
    .text
    .globl _Z3bariii
# int bar(int a, int b, int c)
_Z3bariii:
    subq   $24, %rsp
    movl   %edi, 12(%rsp) # a
    leaq   12(%rsp), %rdi
    movl   %esi, 8(%rsp) # b
    leaq   8(%rsp), %rsi
    movl   %edx, 4(%rsp) # c
    leaq   4(%rsp), %rdx
    call   _Z3fooRiS_S_
    movl   8(%rsp), %eax # ret = b
    addl   12(%rsp), %eax # ret += a
    addl   4(%rsp), %eax # ret += c
    addq   $24, %rsp # free space
    ret
```



Addressing Modes

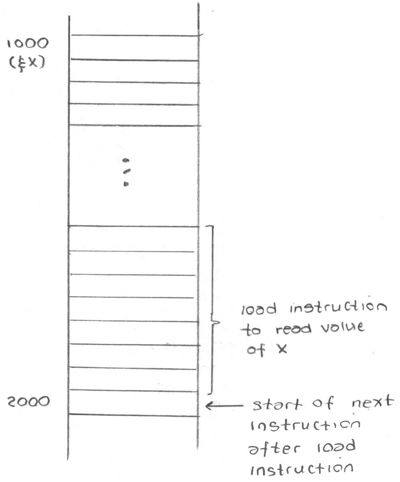
- **immediate**: operand specified in opcode itself
  - `mov $16, %rax`
- **register mode**: operand in register
  - `add %rax, %rax`
- **register indirect**: address of operand contained in register
  - `mov %rax, (%edi)`
  - `call *%rax`
- **direct addressing**: address of operand contained in opcode itself
  - `mov 1024, %rax`
- **displacement addressing**: address of operand obtained by adding displacement in opcode to register
  - `mov %rax, -4(%rbp)`
- **relative addressing**: address of operand specified relative to instruction pointer
  - `jne loop_start`

Relative Addressing for Branches and Calls



- absolute addressing would encode value of 2000 for call target
- relative addressing would encode signed value of 1000 for call target

IP-Relative Addressing for Loads/Stores



- absolute addressing would encode value of 1000 for load address
- relative addressing would encode signed value of -1000 for load address

Relocatable and Position-Independent Code

- **absolute code**: code that must be loaded at specific address in order to function correctly
- **load-time locatable (LTL) code**: code that can be modified at load time to accommodate being run at specific memory location (by effectively patching code to work correctly when loaded at particular address)
- LTL code requires metadata in executable to specify how to perform relocation
- **position-independent code (PIC)**: code that will work correctly (without modification) when loaded at any address
- PIC commonly used for shared libraries so that multiple processes can share ***identical*** copy of library code
- **position-independent executable (PIE)**: executable that is made entirely from PIC code
- PIE binaries potentially advantageous in terms of security (due to address space layout randomization)

[Example] PIE: C++ Code

main.cpp

```
#include <sys/syscall.h>

int answer = 42;

extern "C" void exit(int status) {
    /* invoke exit system call (SYS_exit) */
}

extern "C" int main() {
    return answer;
}

extern "C" void _start() {
    exit(main());
}
```

no_pie.s

```
    .text
# extern "C" void _start()
    .globl _start
_start:
    call main
    mov %rax, %rdi
    call exit # does not return
# extern "C" int main()
    .globl main
main:
    mov answer, %rax
    ret
# extern "C" void exit(int status)
    .globl exit
exit:
    .set SYS_exit, 60 # system call number for exit
    mov $SYS_exit, %rax
    syscall # does not return
    .data
# int answer = 42;
    .globl answer
answer:
    .align 4
    .long 0x2a # 42
```

Output of `objdump -d -r -s no_pie`

```

nopie:     file format elf64-x86-64

Contents of section .text:
 401000 e8080000 004889c7 e8090000 00488b04  .....H.......H..
 401010 25002040 00c348c7 c03c0000 000f05    %. @..H..<.....
Contents of section .data:
 402000 2a000000                             *...

Disassembly of section .text:

0000000000401000 <_start>:
  401000:   e8 08 00 00 00          callq  40100d <main>
  401005:   48 89 c7                mov    %rax,%rdi
  401008:   e8 09 00 00 00          callq  401016 <exit>

000000000040100d <main>:
  40100d:   48 8b 04 25 00 20 40    mov    0x402000,%rax
  401014:   00
  401015:   c3                      retq

0000000000401016 <exit>:
  401016:   48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  40101d:   0f 05                   syscall
```

[Example] PIE: PIE Case

pie.s

```
    .text
# extern "C" void _start()
    .globl _start
_start:
    call main
    mov %rax, %rdi
    call exit # does not return
# extern "C" int main()
    .globl main
main:
    mov answer(%rip), %rax # NOTE: THIS LINE CHANGED!
    ret
# extern "C" void exit(int status)
    .globl exit
exit:
    .set SYS_exit, 60 # system call number for exit
    mov $SYS_exit, %rax
    syscall # does not return
    .data
# int answer = 42;
    .globl answer
answer:
    .align 4
    .long 0x2a # 42
```

[Example] PIE: PIE Case Executable

objdump -d -r -s pie

```

pie:     file format elf64-x86-64


Disassembly of section .text:

0000000000001000 <_start>:
    1000:   e8 08 00 00 00          callq  100d <main>
    1005:   48 89 c7                mov    %rax,%rdi
    1008:   e8 08 00 00 00          callq  1015 <exit>

000000000000100d <main>:
    100d:   48 8b 05 ec 1f 00 00    mov    0x1fec(%rip),%rax
        ↪          # 3000 <answer>
    1014:   c3                      retq

0000000000001015 <exit>:
    1015:   48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
    101c:   0f 05                   syscall
```

Breakpoints

- `int3` instruction intended to be used for breakpoints
- has one-byte opcode (`0xcc`)
- ensures opcode can be replaced with breakpoint instruction without overwriting any other instructions (which could be branched to before breakpoint is hit)
- breakpoint instruction generates processor exception, which is translated into `SIGTRAP` signal by operating system

Why Breakpoint Instruction Has Single-Byte Opcode

breakpoint_single_byte_opcode_1.s

```
    # ...
    jnz skip
    xchg %eax, %ecx # consider placing breaking here
skip:
    mov %ecx, (%rdi)
    # ...
```

Machine code

```
Opcode    Assembly

          # ...
75 01     jne     skip
91        xchg    %eax,%ecx # consider placing breakpoint here
          skip:
89 0f     mov     %ecx,(%rdi)
          # ...
```

Local Variables

- storage for local variables allocated on stack
- on x86, stack grows downwards in memory
- allocate space on stack by subtracting from stack pointer
- free space on stack by adding to stack pointer
- must ensure that stack pointer at function exit matches stack pointer at function entry; otherwise, wrong return address will be taken from stack
- if n bytes needed for local variables (plus padding for stack alignment), subtract n from stack pointer at start of function and add n to stack pointer at end of function

Local Variables Without Frame Pointer

without_frame_pointer.s

```
    .text
    .globl func
func:
    # size of storage (in bytes) for local variables, which
    # must be odd multiple of 8 if any function calls made
    .set local_size, 64
    # allocate storage for local variables
    sub $local_size, %rsp
    # locals at 0(%rsp) to local_size-1(%rsp)
    # return address at local_size(%rsp)
    # arguments passed on stack (if any) start at local_size+8(%rsp)
    # Note that the addresses of the arguments depend on rsp and
    # local_size, and the addresses of local variables depend on rsp.
    # These dependencies are often undesirable.
    # ... (do something useful)
    add $local_size, %rsp
    ret
```

Motivation for Use of Frame Pointer

- when frame pointer not used:
  - address of function arguments depends on both stack pointer and size of storage for locals
  - addresses of local variables depend on stack pointer
- this type of dependence often not desirable, as it makes code more cumbersome to write and also more error prone
- dependence on size of local storage is bad because it may need to change when code modified
- dependence on stack pointer is bad because function may temporarily push registers on stack which would change stack pointer

Use of Frame Pointer

- frame pointer is register used to point to fixed position in stack frame
- since frame pointer points to fixed position in stack frame, all items stored in stack frame (e.g., function arguments and local variables) can be accessed using fixed offsets relative to frame pointer
- on x86-64, rbp normally used for frame pointer
- `enter` and `leave` instructions assume rbp used for frame pointer
- on entry to function, save frame-pointer register on stack and then move stack pointer into frame-pointer register
- additionally, can allocate storage for locals by subtracting from stack pointer
- at function exit, move frame pointer into stack pointer and then restore old value of frame-pointer register by popping from stack
- for performance reasons, `enter` instruction not normally used
- `leave` instruction sometimes used

Local Variables With Frame Pointer

with_frame_pointer.s

```
    .text
    .globl func
func:
    # size of storage (in bytes) for local variables, which
    # must be multiple of 16 if any function calls made
    .set local_size, 64
    # establish rbp as frame pointer
    push %rbp
    mov %rsp, %rbp
    # allocate storage for local variables
    sub $local_size, %rsp
    # locals at -local_size(%rbp) to -1(%rbp)
    # saved rbp at 0(%rbp)
    # return address at 8(%rbp)
    # arguments passed on stack (if any) start at 16(%rbp)
    # Note that the addresses of function arguments
    # and local variables depend neither on rsp
    # nor local_size, which is often desirable.
    # ... (do something useful)
    leave
    # leave is equivalent to:
    # mov %rbp, %rsp
    # pop %rbp
    ret
```

References I

1. Compiler Explorer, https://godbolt.org.
2. Online x86 / x64 Assembler and Disassembler, https://defuse.ca/online-x86-assembler.htm.

1. Matt Godbolt. What Has My Compiler Done for Me Lately? Unbolting the Compiler's Lid. CppCon, Sept. 29, 2017. Available online at https://youtu.be/bSkpMdDe4g4.
2. Matt Godbolt, The Bits Between the Bits: How We Get to main(). CppCon, Sept. 28, 2018. Available online at https://youtu.be/dOfucXtyEsU.
3. Matt Godbolt. What Else Has My Compiler Done For Me Lately? C++Now, May 8, 2018. Available online at https://youtu.be/nAbCKa0FzjQ.
4. x86 Assembly Crash Course. Collegiate Cyber Defense Club, University of Central Florida, Available online at https://youtu.be/75gBFiFtAb8.

Section 3.2

Miscellany

Sandboxing

- **sandboxing**: run untrusted and possibly malicious code in manner that it cannot cause serious harm
- interested in sandboxing techniques for Unix-based systems, such as Linux
- many different security features in Unix/Linux, including:
  - control groups (cgroups), namespaces, ptrace, seccomp, file/thread capabilities, setuid/setgid programs, Berkeley packet filter (BPF), extended Berkeley packet filter (eBPF), discretionary access control (DAC), mandatory access control (MAC), SELinux, KVM, resource limits
- containerization frameworks like Docker utilize many of above features
- virtualization frameworks like KVM typically rely on hardware support for virtualization

Security-Related Terminology

- **authentication**: process of confirming identity of person or device
- **authorization**: security mechanism used to determine user/client privileges or access levels related to system resources (such as programs, files, services, data and application features)
- **privacy**: protecting against unauthorized sharing of information and tracking of users
- **integrity**: data is real, accurate, and safeguarded from unauthorized modification
- **nonrepudiation**: assurance that someone cannot deny something (e.g., sender cannot deny having sent email message or recipient cannot deny having received it)
- **availability**: ability of user to access information or resources (e.g., systems, applications)

Well-Known TCP Ports

Port Number	Protocol
7	Echo
21	FTP
22	SSH
23	Telnet
24	SMTP
80	HTTP
109	POP2
110	POP3
143	IMAP
220	IMAP v3
443	HTTPS
993	IMAP over SSL
995	POP3 over SSL

Part 4

References

Talks I

1. Michael Kerrisk. Using Seccomp to Limit the Kernel Attack Surface. Embedded Linux Conference Europe, Edinburgh, UK, Oct. 22, 2018. Available online at https://youtu.be/-hmG5An2bN8.
2. Michael Kerrisk. An Introduction to Linux IPC Facilities. linux.conf.au, Canberra, Australia, Jan. 30, 2013. Available online at https://youtu.be/vU2HDf5ZhO4.
3. Michael Kerrisk. What's New in Control Groups (cgroups) Version 2?. linux.conf.au, Christchurch, New Zealand, Jan. 23, 2019. Available online at https://youtu.be/yZpNsDe4Qzg.
4. James Morris. Overview of the Linux Kernel Security Subsystem. Linux Security Summit Europe, Edinburgh, Scotland, Oct. 25, 2018. Available online at https://youtu.be/L7KHvKRfTzc.
5. Brendan Gregg. Linux 4.x Tracing: Performance Analysis with bcc/BPF (eBPF). Southern California Linux Expo (SCALE), Pasadena, CA, USA, March 4, 2017. Available online at https://youtu.be/w8nFRoFJ6EQ.

6. Greg Law. Modern Linux C++ Debugging Tools — Under the Covers. CppCon, Aurora, CO, USA, Sept. 20, 2019. Available online at https://youtu.be/WoRmXjVxuFQ.
7. Thomas Cameron. Security-Enhanced Linux for Mere Mortals. Red Hat Summit, San Francisco, CA, USA, May 10, 2018. Available online at https://youtu.be/_WOKRaM-HI4.
8. Stephane Graber. On the Way to Safe Containers. Linux Security Summit, Toronto, ON, Canada, Aug. 25, 2016. Available online at https://youtu.be/FJ2nDQ-aXHM.
9. Greg Law. Linux User/Kernel ABI: The Realities of How C and C++ Programs Really Talk to the OS. ACCU, Bristol, United Kingdom, Apr. 13, 2018. Available online at https://youtu.be/4CdmGxc5BpU.
10. Dave Martin. Moving the Linux ABI to Userspace. Linux Plumbers Conference, Lisbon, Portugal, Sept. 11, 2019. Available online at https://youtu.be/ZIcrT4dw1QE.

11 Christian Brauner. pidfds: Process File Descriptors on Linux. Linux Plumbers Conference, Lisbon, Portugal, Sept. 11, 2019. Available online at https://youtu.be/aCrFujGG8MM.

12 Philip Guo. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. Google TechTalk, Feb. 11, 2011. Available online at https://youtu.be/6XdwHo1BWwY.

References I

[1] Michael Kerrisk. Various conference presentations, slide decks, and videos. Available online at https://man7.org/conf/index.html.