Linux/UNIX System Programming Essentials

Michael Kerrisk
man7.org

January 2024

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

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

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

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

Revision: #9feddaa40cef

Short table of contents

Detailed table of contents

Detailed table of contents

Linux System Programming Essentials

Course Introduction

Michael Kerrisk, man7.org © 2024

January 2024

mtk@man7.org

Outline

Rev: # 9feddaa40cef

Outline

Course prerequisites

- Prerequisites
  - (Good) reading knowledge of C
  - Can log in to Linux / UNIX and use basic commands
- Knowledge of *make(1)* is helpful
  - (Can do a short tutorial during first practical session for those new to *make*)
- Assumptions
  - You are familiar with commonly used parts of standard C library
    - e.g., *stdio* and *malloc* packages

Course goals

- Aimed at programmers building/understanding low-level applications
- Gain strong understanding of programming API that kernel presents to user-space
  - System calls
  - Relevant C library functions
  - Other interfaces (e.g., `/proc`)
  - Necessarily, we sometimes delve into inner workings of kernel
    - (But... not an internals course)
- Course topics
  - Course flyer
  - For more detail, see TOC in course books

Lab sessions

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

Coding exercises

- For coding exercises, you can use any suitable programming language in which you are proficient
  - C/C++ (easiest...)
  - Go, D, Rust, & other languages that compile to native machine code
  - Most features can also be exercised from scripting languages such as Python, Ruby, and Perl
- **Template** solutions are provided for most coding exercises
  - Filenames: `ex.*.c`
  - Look for "FIXMEs" to see what parts you must complete
  - ⚠ You need to edit corresponding `Makefile` to add a new target for the executable
- **Solutions** will be mailed out shortly after end of course

Lab sessions: some thoughts on building code

- Many warnings indicate real problems in the code; fix them
  - And the "harmless warnings" create noise that hides the serious warnings; fix them too
  - **This is a good thing:** `cc -Werror`
    - Treat all warnings as errors
- Rather than writing lots of code before first compile, use a frequent edit-save-build cycle to catch compiler errors early
  - E.g., run the following in a separate window as you edit:

```
$ while inotifywait -q . ; do echo -e '\n\n'; make; done
```

    - *inotifywait* is provided in the *inotify-tools* package
    - (The *echo* command just injects some white space between each build)

Outline

Course materials

- Slides / course book
- Source code tarball
  - Location sent by email
  - Unpacked source code is a Git repository; you can commit/revert changes, etc.
- Kerrisk, M.T. 2010. *The Linux Programming Interface* (TLPI), No Starch Press.
  - Further info on TLPI: `http://man7.org/tlpi/`
    - API changes since publication: `http://man7.org/tlpi/api_changes/`

  (Slides frequently reference TLPI in bottom RHS corner)

Other resources

- POSIX.1-2001 / SUSv3:
  `http://www.unix.org/version3/`
- POSIX.1-2008 / SUSv4:
  `http://www.unix.org/version4/`
- Manual pages
  - Section 2: system calls
  - Section 3: library functions
  - Section 7: overviews
  - Latest version online at
    `http://man7.org/linux/man-pages/`
  - Latest tarball downloadable at
    `https://mirrors.edge.kernel.org/pub/linux/docs/man-pages/`

Books

- General:
  - Stevens, W.R., and Rago, S.A. 2013. *Advanced Programming in the UNIX Environment (3rd edition)*. Addison-Wesley.
    - `http://www.apuebook.com/`
- POSIX threads:
  - Butenhof, D.R. 1996. *Programming with POSIX Threads*. Addison-Wesley.
- TCP/IP and network programming:
  - Fall, K.R. and Stevens, W.R. 2013. *TCP/IP Illustrated, Volume 1: The Protocols (2nd Edition)*. Addison-Wesley.
  - Stevens, W.R., Fenner, B., and Rudoff, A.M. 2004. *UNIX Network Programming,Volume 1 (3rd edition): The Sockets Networking API*. Addison-Wesley.
    - `http://www.unpbook.com/`
  - Stevens, W.R. 1999. *UNIX Network Programming, Volume 2 (2nd edition): Interprocess Communications*. Prentice Hall.
    - `http://www.kohala.com/start/unpv22e/unpv22e.html`
- Operating systems:
  - Tanenbaum, A.S., and Woodhull, A.S. 2006. *Operating Systems: Design And Implementation (3rd edition)*. Prentice Hall.
    - (The Minix book)
  - Comer, D. 2015. *Operating System Design: The Xinu Approach (2nd edition)*

Outline

Common abbreviations used in slides

The following abbreviations are sometimes used in the slides:

- ACL: access control list
- COW: copy-on-write
- CV: condition variable
- CWD: current working directory
- EA: extended attribute
- EOF: end of file
- FD: file descriptor
- FS: filesystem
- FTM: feature test macro
- GID: group ID
  - rGID, eGID, sGID, fsGID
- iff: "if and only if"
- IPC: interprocess communication
- KSE: kernel scheduling entity
- MQ: message queue
- MQD: message queue descriptor
- NS: namespace
- OFD: open file description
- PG: process group
- PID: process ID
- PPID: parent process ID
- SHM: shared memory
- SID: session ID
- SEM: semaphore
- SUS: Single UNIX specification
- UID: user ID
  - rUID, eUID, sUID, fsUID

Outline

Introductions: me

- Programmer, trainer, writer
- UNIX since 1987, Linux since mid-1990s
- Active contributor to Linux
  - API review, testing, and documentation
    - API design and design review
    - Lots of testing, lots of bug reports, a few kernel patches
  - Maintainer of Linux *man-pages* project (2004-2021)
    - Documents kernel-user-space + C library APIs
    - Contributor since 2000
    - As maintainer: ≈23k commits, 196 releases
    - Author/coauthor of ≈440 out of ≈1060 manual pages
- Kiwi in .de
  - (mtk@man7.org, PGP: 4096R/3A35CE5E)
  - @mkerrisk (feel free to tweet about the course as we go...)
  - `http://linkedin.com/in/mkerrisk`

Introductions: you

In brief:

- Who are you?
  - If virtual: where are you?
- (Optionally:) any special goals for the course?
- Two things you like to do when you are not in front of a keyboard, and one thing you don't like doing...

Questions policy

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

Notes

Notes

Linux System Programming Essentials

Fundamental Concepts

January 2024

mtk@man7.org

Outline

Rev: # 9feddaa40cef

Outline

Error handling

- Most system calls and library functions return a status indicating success or failure
- On failure, most system calls:
  - Return –1
  - Place integer value in global variable *errno* to indicate cause
- Some library functions follow same convention
- Often, we'll omit return values from slides, where they follow usual conventions
  - Check manual pages for details

Error handling

- Return status should **always** be tested
- ⚠ Inspect *errno* only if result status indicates failure
  - APIs do not reset *errno* to 0 on success
  - A successful call may modify *errno* (POSIX allows this)
  - E.g., this is wrong:

```
fd = open(pathname, O_RDONLY);

printf("open() has returned\n");    // Might modify errno!

if (fd == -1) {           // Did open() fail?
    perror("open");       // Print message based on 'errno'
    exit(EXIT_FAILURE);
}
```


errno

- When an API call fails, *errno* is set to indicate cause
- Integer value, global variable
  - In multithreading environment, each thread has private *errno*
- Error numbers in *errno* are > 0
- `<errno.h>` defines symbolic names for error numbers

```
#define EPERM    1      /* Operation not permitted */
#define ENOENT   2      /* No such file or directory */
#define ESRCH    3      /* No such process */
...
```

  - *errno(1)* command can be used to search for errors by number, name, or substring in textual message
    - Part of *moreutils* package (since 2012)

Checking for errors

```
cnt = read(fd, buf, numbytes);

if (cnt == -1) {            /* Was there an error? */
    if (errno == EINTR)
        fprintf(stderr, "read() was interrupted by a signal\n");
    else if (errno == EBADF)
        fprintf(stderr, "read() given bad file descriptor\n");
    else {
        /* Some other error occurred */
    }
}
```


Displaying error messages

```
#include <stdio.h>
void perror(const char *msg);
```

- Outputs to *stderr*:
  - *msg* + ": " + string corresponding to value in *errno*
  - E.g., if *errno* contains `EBADF`, *perror("close")* would display:
    `close: Bad file descriptor`
- Simple error handling:

```
fd = open(pathname, flags, mode);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

  - (More sophisticated programs might take actions other than terminating on syscall error)

Displaying error messages

```
#include <string.h>
char *strerror(int errnum);
```

- Returns an error string corresponding to error in *errnum*
  - Same string as printed by *perror()*
- Unknown error number? ⇒ *"Unknown error nnn"*
  - Or `NULL` on some systems

Outline

System data types

- Various system info needs to be represented in C
  - Process IDs, user IDs, file offsets, etc.
- Using native C data types (e.g., *int*, *long*) in application code would be nonportable; e.g.:
  - *sizeof(long)* might be 4 on one system, but 8 on another
  - One system might use *int* for PIDs, while another uses *long*
  - Even on same system, things may change across versions
    - E.g., in kernel 2.4, Linux switched from 16 to 32-bit UIDs
- ⇒ POSIX defines system data types:
  - Implementations must suitably define each system data type
  - Defined via `typedef`; e.g., `typedef int pid_t`
    - Most types have names suffixed "`_t`"
  - Applications should use these types; e.g., `pid_t mypid;`
    - ⇒ will compile to correct types on any conformant system

[TLPI §3.6.2]

Examples of system data types

Data type	POSIX type requirement	Description
uid_t	Integer	User ID
gid_t	Integer	Group ID
pid_t	Signed integer	Process ID
id_t	Integer	Generic ID type; can hold *pid_t*, *uid_t*, *gid_t*
off_t	Signed integer	File offset or size
sigset_t	Integer or structure	Signal set
size_t	Unsigned integer	Size of object (in bytes)
ssize_t	Signed integer	Size of object or error indication
time_t	Integer/real-floating	Time in seconds since Epoch
timer_t	Arithmetic type	POSIX timer ID

(Arithmetic type ∈ integer or floating type)

Printing system data types

- Need to take care when passing system data types to *printf()*
- Example: *pid_t* can be *short*, *int*, or *long*
- Suppose we write:

```
printf("My PID is: %d\n", getpid());
```

- Works fine if:
  - *pid_t* is *int*
  - *pid_t* is *short* (C promotes *short* argument to *int*)
- But **what if *pid_t* is *long*** (and *long* is bigger than *int*)?
  - ⇒ argument exceeds range understood by format specifier (top bytes will be lost)

Printing system data types

- On virtually all implementations, most integer system data types are *long* or smaller
  - ⇒ Promote to *long* when printing system data types

```
printf("My PID is: %ld\n", (long) getpid());
```

- Most notable exception: *off_t* is typically *long long*
  - Promote to *long long* for *printf()*

```
printf("Offset is %lld\n",
        (long long) lseek(fd, 0, SEEK_CUR));
```

- Can also use `%zu` and `%zd` for *size_t* and *ssize_t*
- C99 has *intmax_t* (*uintmax_t*) with `%jd` (`%ju`) *printf()* specifier
  - Solution for all integer types, but not on pre-C99 systems
  - Must include `<stdint.h>` to get these type definitions

Outline

Code examples presented in course

- **Code tarball** == code from TLPI + further code for course
- **Examples on slides edited/excerpted** for brevity
  - E.g., error-handling code may be omitted
- Slides always show **pathname for full source code**
  - Full source code always includes error-handling code
- Code license:
  - GNU GPL v3 for programs
  - GNU Lesser GPL v3 for library functions
  - `http://www.gnu.org/licenses/#GPL`
    - *Understanding Open Source and Free Software Licensing*, A.M. St Laurent, 2004
    - *Open Source Licensing: Software Freedom and Intellectual Property Law*, L. Rosen, 2004
    - *Open Source Software: Rechtliche Rahmenbedingungen der Freien Software*, Till Jaeger, 2020
    - *Droit des logiciels*, F. Pellegrini & S. Canevet, 2013

Example code `lib/` subdirectory

- `lib/` subdirectory contains code of a few functions commonly used in examples
- *camelCase* function name?
  - $\Rightarrow$ It's mine

Common header file

- Many code examples make use of header file `tlpi_hdr.h`
- Goal: make code examples a little shorter
- `tlpi_hdr.h`:
  - Includes a few frequently used header files
  - Includes declarations of some error-handling functions

[TLPI §3.5.2]

Error-handling functions used in examples

- Could handle errors as follows:

```
fd = open(pathname, flags, mode);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

- Verbose! To make error handling more compact, I define some simple error-handling functions

Error-handling functions used in examples

```
#include "tlpi_hdr.h"
errExit(const char *format, ...);
```

- Prints error message on *stderr* that includes:
  - Symbolic name for *errno* value (via some trickery)
  - *strerror()* description for current *errno* value
  - Text from the *printf()*-style message supplied in arguments
  - A terminating newline
- Terminates program with exit status `EXIT_FAILURE` (1)
- Example:

```
if (close(fd) == -1)
    errExit("close (fd=%d)", fd);
```

  might produce:

```
ERROR [EBADF Bad file descriptor] close (fd=5)
```

Error-handling functions used in examples

```
#include "tlpi_hdr.h"
errMsg(const char *format, ...);
```

- Like *errExit()*, but does not terminate program

```
#include "tlpi_hdr.h"
fatal(const char *format, ...);
```

- Displays a *printf()*-style message + newline
- Terminates program with exit status `EXIT_FAILURE` (1)

Building the sample code

- You can manually compile the example programs, but there is also a **Makefile** in each directory
- ⇒ Typing `make` in source code root directory builds all programs in all subdirectories
- If you encounter build errors relating to ACLs, capabilities, or SELinux, see `http://man7.org/tlpi/code/faq.html`
  - Preferred solution is to install the necessary packages:
    - Debian, Ubuntu: *libcap-dev*, *libacl1-dev*, *libreadline-dev* *libcrypt-dev*
    - RPM-based systems: *libcap-devel*, *libacl-devel*, *readline-devel* *libxcrypt-devel*

Using library functions from the sample code

To use my library functions in your code:

- **Include** `tlpi_hdr.h` in your C source file
  - Located in `lib/` subdirectory in source code
- **Link against my library**, `libtlpi.a`, located in source code root directory
  - To build library, run `make` in the source code root directory or in `lib/` subdirectory
- **Method 1**: Place your program in one of "my" directories, add target to corresponding `Makefile`, and build using `make`
- **Method 2**: Manually compile with the following command:

```
cc -Isrc-root/lib yourprog.c src-root/libtlpi.a
```

  - *src-root* must be replaced with the absolute or relative path of source code root directory

Notes

Linux System Programming Essentials

File I/O

Michael Kerrisk, man7.org © 2024

January 2024

mtk@man7.org

Outline

Outline

Files

- "On UNIX, everything is a file"
  - More correctly: "everything is a file descriptor"
- Note: the term **file** can be ambiguous:
  - A **generic term**, covering disk files, directories, sockets, FIFOs, terminals and other devices and so on
  - Or specifically, a **disk file** in a filesystem
  - To clearly distinguish the latter, the term **regular file** is sometimes used

System calls versus *stdio*

- C programs usually use *stdio* package for file I/O
- Library functions layered on top of I/O system calls

System calls	**Library functions**
file descriptor (*int*)	file stream (*FILE **)
open(), *close()*	*fopen()*, *fclose()*
lseek()	*fseek()*, *ftell()*
read()	*fgets()*, *fscanf()*, *fread()* ...
write()	*fputs()*, *fprintf()*, *fwrite()*, ...
–	*feof()*, *ferror()*

- We presume understanding of *stdio* ; ⇒ focus on system calls

File descriptors

- All I/O is done using file descriptors (FDs)
  - nonnegative integer that identifies an open file
- Used for all types of files
  - terminals, regular files, pipes, FIFOs, devices, sockets, ...
- 3 FDs are normally available to programs run from shell:
  - (POSIX names are defined in `<unistd.h>`)

FD	**Purpose**	POSIX name	*stdio* stream
0	Standard input	`STDIN_FILENO`	*stdin*
1	Standard output	`STDOUT_FILENO`	*stdout*
2	Standard error	`STDERR_FILENO`	*stderr*

Key file I/O system calls

Four fundamental calls:

- *open()*: open a file, optionally creating it if needed
  - Returns file descriptor used by remaining calls
- *read()*: input
- *write()*: output
- *close()*: close file descriptor

Outline

open(): opening a file

```
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags, ... /* mode_t mode */);
```

- Opens existing file / creates and opens new file
- Arguments:
  - *pathname* identifies file to open
  - *flags* controls semantics of call
    - e.g., open an existing file vs create a new file
  - *mode* specifies permissions when creating new file
- Returns: a file descriptor (nonnegative integer)
  - (Guaranteed to be lowest available FD)

[TLPI §4.3]

open() *flags* argument

flags is formed by ORing (|) together:

- Access mode
  - Specify exactly one of `O_RDONLY`, `O_WRONLY`, or `O_RDWR`
- File creation flags (bit flags)
- File status flags (bit flags)

[TLPI §4.3.1]

File creation flags

- **File creation flags**:
  - Affect behavior of *open()* call
  - Can't be retrieved or changed
- Examples:
  - `O_CREAT`: create file if it doesn't exist
    - *mode* argument must be specified
    - Without `O_CREAT`, can open only an existing file (else: `ENOENT`)
  - `O_EXCL`: create "exclusively"
    - Give an error (`EEXIST`) if file already exists
    - Only meaningful with `O_CREAT`
  - `O_TRUNC`: truncate existing file to zero length
    - I.e., discard existing file content

File status flags

- **File status flags**:
  - Affect semantics of subsequent file I/O
  - Can be retrieved and modified using *fcntl()*
- Examples:
  - `O_APPEND`: always append writes to end of file
  - `O_NONBLOCK`: nonblocking I/O

open() examples

- Open existing file for reading:

```
fd = open("script.txt", O_RDONLY);
```

- Open file for read-write, create if necessary, ensure we are creator:

```
fd = open("myfile.txt", O_CREAT | O_EXCL | O_RDWR,
          S_IRUSR | S_IWUSR);       /* rw------- */
```

- Open file for writing, creating if necessary:

```
fd = open("myfile.txt", O_CREAT | O_WRONLY, S_IRUSR);
```

  - File opened for writing, but created with only read permission!

read(): reading from a file

```
#include <unistd.h>
ssize_t read(int fd, void *buffer, size_t count);
```

- *fd*: file descriptor
- *buffer*: pointer to buffer to store data
- *count*: number of bytes to read
  - (*buffer* must be at least this big)
  - (*ssize_t* and *size_t* are signed and unsigned integer types)
- Returns:
  - > 0: number of bytes read
    - May be $<$ *count* (e.g., terminal *read()* gets only one line)
  - 0: end of file
  - -1: error
- ⚠ No terminating null byte is placed at end of buffer

write(): writing to a file

```
#include <unistd.h>
ssize_t write(int fd, const void *buffer, size_t count);
```

- *fd*: file descriptor
- *buffer*: pointer to data to be written
- *count*: number of bytes to write
- Returns:
  - Number of bytes written
    - May be $<$ *count* (a "partial write")
      (e.g., write fills device, or insufficient space to write entire buffer to nonblocking socket)
  - −1 on error

close(): closing a file

```
#include <unistd.h>
int close(fd);
```

- *fd*: file descriptor
- Returns:
  - 0: success
  - −1: error
- Really should check for error!
  - Accidentally closing same FD twice
    - I.e., detect program logic error
  - Filesystem-specific errors
    - E.g., NFS commit failures may be reported only at *close()*
- **Note**: *close()* **always** releases FD, even on failure return
  - See *close(2)* manual page

Example: copy.c

```
$ ./copy old-file new-file
```

- A simple version of *cp(1)*

Example: fileio/copy.c (snippet)

Always remember to handle errors!

```
#define BUF_SIZE 1024
char buf[BUF_SIZE];

int infd = open(argv[1], O_RDONLY);
if (infd == -1) errExit("open %s", argv[1]);

int flags = O_CREAT | O_WRONLY | O_TRUNC;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP;     /* rw-r----- */
int outfd = open(argv[2], flags, mode);
if (outfd == -1) errExit("open %s", argv[2]);

ssize_t nread;
while ((nread = read(infd, buf, BUF_SIZE)) > 0)
    if (write(outfd, buf, nread) != nread)
        fatal("write() returned error or partial write occurred");
if (nread == -1) errExit("read");

if (close(infd) == -1) errExit("close");
if (close(outfd) == -1) errExit("close");
```

Universality of I/O

- The fundamental I/O system calls work on almost all file types:

```
$ ls > mylist
$ ./copy mylist new          # Regular file

$ ./copy mylist /dev/tty     # Device

$ mkfifo f                   # FIFO
$ cat f &                    # (reads from FIFO)
$ ./copy mylist f            # (writes to FIFO)
```


Notes for online practical sessions

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

Shared screen etiquette

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

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

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

Using *tmate* in in-person practical sessions

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

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

```
$ tmate
...
Connecting to ssh.tmate.io...
Note: clear your terminal before sharing readonly access
web session read only: ...
ssh session read only: ssh SOmErAnDOm5Tr1Ng@lon1.tmate.io
web session: ...
ssh session: ssh SOmEoTheRrAnDOm5Tr1Ng@lon1.tmate.io
```

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

Exercise notes

- For many exercises, there are templates for the solutions
  - Filenames: `ex.*.c`
  - Look for FIXMEs to see what pieces of code you must add
  - ⚠ You will need to edit the corresponding `Makefile` to add a new target for the executable
    - Look for the `EXERCISE_FILES_EXE` macro

```
-EXERCISE_FILES_EXE = # ex.prog_a ex.prob_b
+EXERCISE_FILES_EXE = ex.prog_a # ex.prog_b
```

- Get a *make* tutorial now if you need one

Exercise

1. Using *open()*, *close()*, *read()*, and *write()*, implement the command `tee [-a] file` ([template: `fileio/ex.tee.c`]). This command writes a copy of its standard input to standard output and to `file`. If `file` does not exist, it should be created. If `file` already exists, it should be truncated to zero length (`O_TRUNC`). The program should support the `-a` option, which appends (`O_APPEND`) output to the file if it already exists, rather than truncating the file.

   Some hints:
   - You can build `../libtlpi.a` by doing *make* in source code root directory.
   - Standard input & output are automatically opened for a process.
   - Remember that you will need to add a target in the `Makefile`!
   - After first doing some simple command-line testing, test using the unit test in the `Makefile`: `make tee_test`.
   - Why does "`man open`" show the wrong manual page? It finds a page in the wrong section first. Try "`man 2 open`" instead.
   - `while inotifywait -q . ; do echo -e '\n\n'; make; done`
     - You may need to install the *inotify-tools* package
   - Command-line options can be parsed using *getopt(3)*.

Linux System Programming Essentials

Processes

Michael Kerrisk, man7.org

January 2024

mtk@man7.org

Outline

Rev: # 9feddaa40cef

Outline

Process ID

```
#include <unistd.h>
pid_t getpid(void);
```

- **Process** == running instance of a program
  - Program + program loader (kernel) ⇒ process
- Every process has a process ID (PID)
  - *pid_t*: positive integer that uniquely identifies process
  - *getpid()* returns callers's PID
  - Maximum PID is 32767 on Linux
    - "Elevator" algorithm then cycles, reusing PIDs, starting at low numbers
    - All PID slots used? ⇒ *fork()* fails with `EAGAIN`
    - Limit adjustable via `/proc/sys/kernel/pid_max` (up to kernel's `PID_MAX_LIMIT` constant, typically 4*1024*1024)

[TLPI §6.2]

Parent process ID

```
#include <unistd.h>
pid_t getppid(void);
```

- Every process has a parent
  - Typically, process that created this process using *fork()*
  - Parent process is informed when its child terminates
- All processes on system thus form a tree
  - At root is *init*, PID 1, the ancestor of all processes
  - "Orphaned" processes are "adopted" by *init*
- *getppid()* returns PID of caller's parent process (PPID)

[TLPI §6.2]

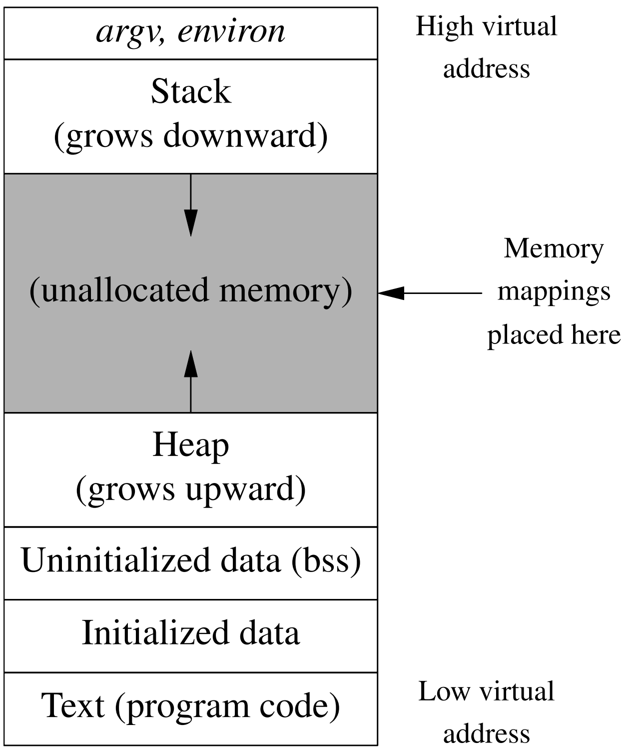
Outline

Process memory layout

Virtual memory of a process is divided into **segments**:

- **Text**: machine-language instructions
  - Marked read-only to prevent self-modification
  - Multiple processes can share same code in memory
- **Initialized data**: global and static variables that are explicitly initialized
  - Values read from program file when process is created
- **Uninitialized data**: global and static variables that are not explicitly initialized
  - Initialized to zero when process is created
- **Stack**: storage for function local variables and call linkage info (saved SP and PC registers)
- **Heap**: an area from which memory can be dynamically allocated and deallocated
  - *malloc()* and *free()*

Process memory layout (simplified)

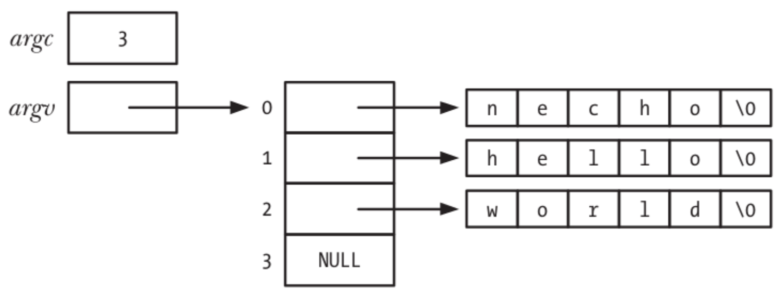


[TLPI §6.3]

Outline

Command-line arguments

- Command-line arguments of a program provided as first two arguments of *main()*
  - Conventionally named *argc* and *argv*
- *int argc*: number of arguments
- *char *argv[]*: array of pointers to arguments (strings)
  - *argv[0]* == name used to invoke program
  - *argv[argc]* == *NULL*
- E.g., for the command, `necho hello world`:



[TLPI §6.6]

Outline

Environment list (*environ*)

Each process has a list of **environment variables**

- Strings of form *name=value*
- New process inherits copy of parent's environment
  - Simple (one-way) interprocess communication
- Commonly used to control behavior of programs
- Examples:
  - `HOME`: user's home directory (initialized at login)
  - `PATH`: list of directories to search for executable programs
  - `EDITOR`: user's preferred editor

[TLPI §6.7]

Environment list (*environ*)

- Can create environment variables within shell:

```
$ MANWIDTH=72           # Create shall var.
$ export MANWIDTH       # Turn shell var. into environment var.
$ man getpid
```

  - Or: `export MANWIDTH=72`
- All processes created by shell will inherit definition
- Creating an environment variable for a single command (does not modify shell's environment):

```
$ MANWIDTH=72 man getpid
```

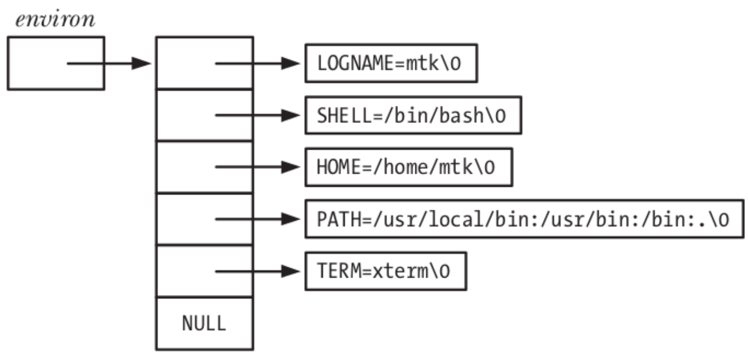
- To list all environment variables, use *env(1)* or *printenv(1)*

Accessing the environment from a program

- Environment list can be accessed via a global variable:

```
extern char **environ;
```

- NULL-terminated array of pointers to strings:



- Displaying environment:

```
for (char **ep = environ; *ep != NULL; ep++)
    puts(*ep);
```

Environment variable APIs

- Fetching value of an EV: `value = getenv("NAME");`
- Creating/modifying an EV:
  - `putenv("NAME=value");`
  - `setenv("NAME", "value", overwrite);`
- Removing an EV: `unsetenv("NAME");`
- `/proc/PID/environ` can be used (with suitable permissions) to view environment of another process
- See manual pages and TLPI §6.7

Outline

The /proc filesystem

- Pseudofilesystem that exposes kernel information via filesystem metaphor
  - Structured as a set of subdirectories and files
  - *proc(5)* manual page
- Files don't really exist
  - Created on-the-fly when pathnames under `/proc` are accessed
- Many files read-only
- Some files are writable ⇒ can update kernel settings

The /proc filesystem: examples

- `/proc/cmdline`: command line used to start kernel
- `/proc/cpuinfo`: info about CPUs on the system
- `/proc/meminfo`: info about memory and memory usage
- `/proc/modules`: info about loaded kernel modules
- `/proc/sys/fs/`: files and subdirectories with filesystem-related info
- `/proc/sys/kernel/`: files and subdirectories with various readable/settable kernel parameters
- `/proc/sys/net/`: files and subdirectories with various readable/settable networking parameters

/proc/*PID*/ directories

- One `/proc/PID/` subdirectory for each running process
- Subdirectories and files exposing info about process with corresponding PID
- Some files publicly readable, some readable only by process owner; a few files writable
- Examples
  - `cmdline`: command line used to start program
  - `cwd`: current working directory
  - `environ`: environment of process
  - `fd`: directory with info about open file descriptors
  - `limits`: resource limits
  - `maps`: mappings in virtual address space
  - `status`: (lots of) info about process

Notes

Linux System Programming Essentials

Signals

Michael Kerrisk, man7.org © 2024

January 2024

mtk@man7.org

Outline

Rev: # 9feddaa40cef

Outline

Signals are a notification mechanism

- Signal == notification to a process that an event occurred
  - "Software interrupts"
  - **asynchronous**: receiver (generally) can't predict when a signal will occur

Signal types

- 64 signals (on Linux)
- Each signal has a unique integer value
  - Numbered starting at 1 ⚠
- Defined symbolically in `<signal.h>`:
  - Names of form `SIGxxx`
  - e.g., signal 2 is `SIGINT` ("terminal interrupt")
- Two broad categories of signals:
  - "Standard" signals (1 to 31)
    - Mostly for kernel-defined purposes
  - Realtime signals (32 to 64)
    - Exist for user-defined purposes

[TLPI §20.1]

Signal generation

- Signals can be sent to a process by:
  - The kernel (the common case)
  - Another process (with suitable permissions)
    - *kill(pid, sig)* and related APIs
- Kernel generates signals for various events, e.g.:
  - Attempt to access a nonexistent memory address (`SIGSEGV`)
  - Terminal *interrupt* character (Control-C) was typed (`SIGINT`)
  - Child process terminated (`SIGCHLD`)
  - Process CPU time limit exceeded (`SIGXCPU`)

[TLPI §20.1]

Terminology

Some terminology:

- A signal is **generated** when an event occurs
- Later, a signal is **delivered** to the process, which then takes some action in response
- Between generation and delivery, a signal is **pending**
- We can **block** (delay) delivery of specific signals by adding them to process's **signal mask**
  - **Signal mask == set of signals whose delivery is blocked**
  - Pending signal is delivered only after it is unblocked

[TLPI §20.1]

Outline

Signal default actions

- When a signal is delivered, a process takes one of these default actions:
  - **Ignore**: signal is discarded by kernel, has no effect on process
  - **Terminate**: process is terminated ("killed")
  - **Core dump + terminate**: process produces a core dump and is terminated
    - Core dump file can be used to examine state of program inside a debugger
    - See also *core(5)* manual page
  - **Stop**: execution of process is suspended
  - **Continue**: execution of a stopped process is resumed
- Default action for each signal is signal-specific

[TLPI §20.2]

Standard signals and their default actions

Name	Description	Default
SIGABRT	Abort process	Core
SIGALRM	Real-time timer expiration	Term
SIGBUS	Memory access error	Core
SIGCHLD	Child stopped or terminated	Ignore
SIGCONT	Continue if stopped	Cont
SIGFPE	Arithmetic exception	Core
SIGHUP	Hangup	Term
SIGILL	Illegal Instruction	Core
SIGINT	Interrupt from keyboard	Term
SIGIO	I/O Possible	Term
SIGKILL	Sure kill	Term
SIGPIPE	Broken pipe	Term
SIGPROF	Profiling timer expired	Term
SIGPWR	Power about to fail	Term
SIGQUIT	Terminal quit	Core
SIGSEGV	Invalid memory reference	Core
SIGSTKFLT	Stack fault on coprocessor	Term
SIGSTOP	Sure stop	Stop
SIGSYS	Invalid system call	Core
SIGTERM	Terminate process	Term
SIGTRAP	Trace/breakpoint trap	Core
SIGTSTP	Terminal stop	Stop
SIGTTIN	Terminal input from background	Stop
SIGTTOU	Terminal output from background	Stop
SIGURG	Urgent data on socket	Ignore
SIGUSR1	User-defined signal 1	Term
SIGUSR2	User-defined signal 2	Term
SIGVTALRM	Virtual timer expired	Term
SIGWINCH	Terminal window size changed	Ignore
SIGXCPU	CPU time limit exceeded	Core
SIGXFSZ	File size limit exceeded	Core

- Signal default actions are:
  - Term: terminate the process
  - Core: produce core dump and terminate the process
  - Ignore: ignore the signal
  - Stop: stop (suspend) the process
  - Cont: resume process (if stopped)
- SIGKILL and SIGSTOP can't be caught, blocked, or ignored
- TLPI §20.2

Stop and continue signals

- Certain signals **stop** a process, freezing its execution
- Examples:
  - `SIGTSTP`: “terminal stop” signal, generated by typing Control-Z
  - `SIGSTOP`: “sure stop” signal
- `SIGCONT` causes a stopped process to resume execution
  - `SIGCONT` is ignored if process is not stopped
- Most common use of these signals is in **shell job control**

Changing a signal’s disposition

- Instead of default, we can change a signal’s disposition to:
  - **Ignore** the signal
  - **Handle (“catch”) the signal**: execute a user-defined function upon delivery of the signal
  - Revert to the **default action**
    - Useful if we earlier changed disposition
- Can’t change disposition to *terminate* or *core dump* + *terminate*
  - But, a signal handler can emulate these behaviors
- Can’t change disposition of `SIGKILL` or `SIGSTOP` (error: `EINVAL`)
  - So, they always kill or stop a process

Changing a signal's disposition: *sigaction()*

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act,
                       struct sigaction *oldact);
```

sigaction() changes (and/or retrieves) disposition of signal *sig*

- *sigaction* structure describes a signal's disposition
- *act* points to structure specifying new disposition for *sig*
- *oldact* returns previous disposition for *sig*
  - Can be `NULL` if we don't care
- `sigaction(sig, NULL, &oldact)` returns current disposition, without changing it

[TLPI §20.13]

sigaction structure

```
struct sigaction {
    void     (*sa_handler)(int);
    sigset_t sa_mask;
    int      sa_flags;
    void     (*sa_restorer)(void);
};
```

- *sa_handler* specifies disposition of signal:
  - Address of a signal handler function
  - `SIG_IGN`: ignore signal
  - `SIG_DFL`: revert to default disposition
- *sa_mask*: signals to block while handler is executing
  - Field is initialized using macros described in *sigsetops(3)*
- *sa_flags*: bit mask of flags affecting invocation of handler
- *sa_restorer*: not for application use
  - Used internally to implement "signal trampoline"

Ignoring a signal (signals/ignore_signal.c)

```
int ignoreSignal(int sig)
{
    struct sigaction sa;

    sa.sa_handler = SIG_IGN;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    return sigaction(sig, &sa, NULL);
}
```

- A "library function" that ignores specified signal
- *sa_mask* field is significant only when establishing a signal handler, but for best practice we initialize to sensible value

Outline

Displaying signal descriptions

```
#define _GNU_SOURCE
#include <string.h>
char *strsignal(int sig);
```

- Returns string describing signal *sig*
- NSIG constant is 1 greater than maximum signal number
  - Define _GNU_SOURCE to get definition from <signal.h>

[TLPI §20.8]

Example: signals/t_strsignal.c

```
int main(int argc, char *argv[]) {
    for (int sig = 1; sig < NSIG; sig++)
        printf("%2d: %s\n", sig, strsignal(sig));

    exit(EXIT_SUCCESS);
}
```

```
$ ./t_strsignal
 1: Hangup
 2: Interrupt
 3: Quit
 4: Illegal instruction
 5: Trace/breakpoint trap
 6: Aborted
 7: Bus error
 8: Floating point exception
 9: Killed
10: User defined signal 1
11: Segmentation fault
12: User defined signal 2
13: Broken pipe
...
```

Waiting for a signal: *pause()*

```
#include <unistd.h>
int pause(void);
```

- Blocks execution of caller until a signal is caught
- Always returns –1 with *errno* set to `EINTR`
  - (Standard return for blocking system call that is interrupted by a signal handler)
- (See also *sigsuspend(2)*)

[TLPI §20.14]

Outline

Signal handlers

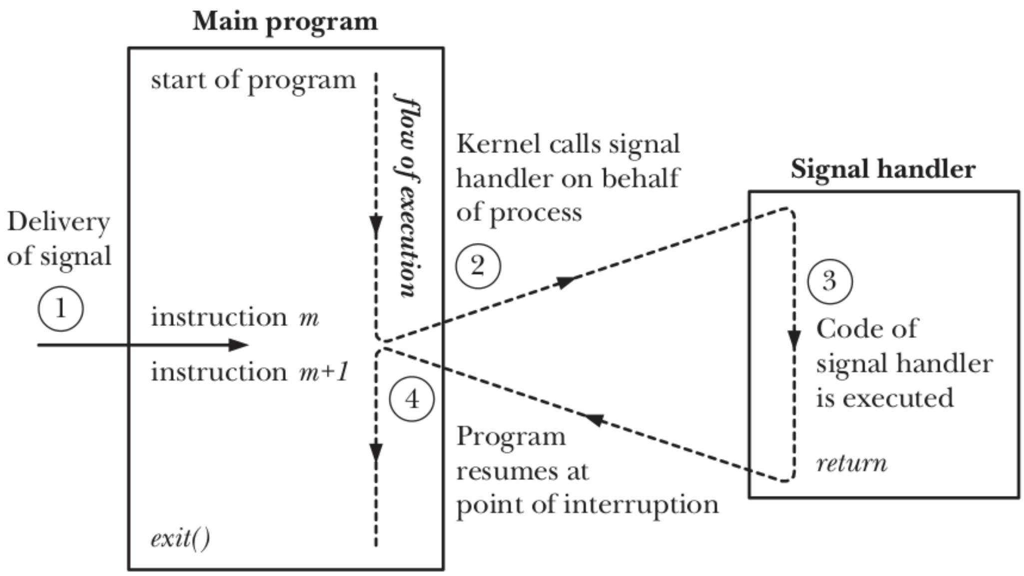
- Programmer-defined function
- Called with one integer argument: number of signal
  - ⇒ handler installed for multiple signals can differentiate...
- Returns `void`

```
void
myHandler(int sig)
{
    /* Actions to be performed when signal is delivered */
}
```

[TLPI §20.4]

Signal handler invocation

- Automatically invoked by kernel when signal is delivered:
  - Can interrupt main program flow at any time
  - On return, execution continues at point of interruption



Example: signals/ouch_sigaction.c

Print "Ouch!" when Control-C is typed at keyboard

```
static void sigHandler(int sig) {
    printf("Ouch!\n");            /* UNSAFE */
}

int main(int argc, char *argv[]) {
    struct sigaction sa;
    sa.sa_flags = 0;              /* No flags */
    sa.sa_handler = sigHandler;   /* Handler function */
    sigemptyset(&sa.sa_mask);     /* Don't block additional signals
                                     during invocation of handler */
    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

    for (;;)
        pause();                  /* Wait for a signal */
}
```


Using *tmate* in in-person practical sessions

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

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

```
$ tmate
...
Connecting to ssh.tmate.io...
Note: clear your terminal before sharing readonly access
web session read only: ...
ssh session read only: ssh SOmErAnDOm5Tr1Ng@lon1.tmate.io
web session: ...
ssh session: ssh SOmEoTheRrAnDOm5Tr1Ng@lon1.tmate.io
```

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

Exercise

- While a signal handler is executing, the signal that caused it to be invoked is (by default) temporarily added to the signal mask, so that it is blocked from further delivery until the signal handler returns. Consequently, execution of a signal handler can't be interrupted by a further execution of the same handler. To demonstrate that this is so, modify the signal handler in the `signals/ouch_sigaction.c` program to include the following after the existing *printf()* statement:

```
sleep(5);
printf("Bye\n");
```

  Build and run the program, type control-C once, and then, while the signal handler is executing, type control-C three more times. What happens? In total, how many times is the signal handler called?

Outline

Signal sets

- Various signal-related APIs work with **signal sets**
- Signal set == data structure that represents multiple signals
- Data type: *sigset_t*
  - Typically a bit mask, but not necessarily

[TLPI §20.9]

Manipulating signal sets

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
int sigismember(const sigset_t *set, int sig);
```

- *sigemptyset()* initializes *set* to contain no signals
- *sigfillset()* initializes *set* to contain all signals
  - We **must** initialize set using *sigemptyset()* or *sigfillset()* before employing macros below
  - Using *memset()* to zero a signal set is *not* correct
- *sigaddset()* adds *sig* to *set*
- *sigdelset()* removes *sig* from *set*
- *sigismember()* returns 1 if *sig* is in *set*, 0 if it is not, or –1 on error (e.g., *sig* is invalid)

Blocking signals (the signal mask)

- Each process has a **signal mask**–a set of signals whose delivery is currently blocked
  - (In truth: each **thread** has a signal mask...)
- If a blocked signal is generated, it remains pending until removed from signal mask
- The signal mask can be changed in various ways:
  - While handler is invoked, the **signal that triggered the handler** is (temporarily) added to signal mask
  - While handler is invoked, any signals specified in ***sa_mask*** are (temporarily) added to signal mask
  - Explicitly, using ***sigprocmask()***
- Attempts to block `SIGKILL`/`SIGSTOP` are silently ignored

[TLPI §20.10]

sigprocmask()

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- Adds signals to, or removes signals from, caller's signal mask
  - (Typical use: prevent interruption by signal handler while updating a shared data structure)
- *how* specifies change to signal mask:
  - `SIG_BLOCK`: **add** signals in *set* to signal mask
    - I.e., *union* with existing signal mask
  - `SIG_UNBLOCK`: **remove** signals in *set* from signal mask
  - `SIG_SETMASK`: **assign** *set* to signal mask
    - I.e., *overwrite* existing signal mask

[TLPI §20.10]

sigprocmask()

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- *oldset* returns previous signal mask
  - Can be `NULL` if we don't care
- `sigprocmask(how, NULL, &oldset)` retrieves current mask without changing it
  - *how* is ignored

[TLPI §20.10]

Example: temporarily blocking a signal

- The following code snippet shows how to temporarily block a signal (`SIGINT`) while executing a block of code

```
sigset_t blocking, prev;

sigemptyset(&blocking);
sigaddset(&blocking, SIGINT);
sigprocmask(SIG_BLOCK, &blocking, &prev);

/* ... Code to execute with SIGINT blocked ... */

sigprocmask(SIG_SETMASK, &prev, NULL);
```

  - We might do this because main program wants to operate on global variables that signal handle would also access

Pending signals

```
#include <signal.h>
int sigpending(sigset_t *set);
```

- Between generation and delivery, a signal is **pending**
  - Pending state is normally unobservable unless signal is explicitly blocked
- *sigpending()* returns (in *set*) the set of signals currently pending for caller
  - We do **not** need to initialize *set* before calling *sigpending()*
- Can examine *set* using *sigismember()*:

```
sigset_t pending;
sigpending(&pending);
if (sigismember(&pending, SIGINT))
    printf("SIGINT (%s) is pending\n", strsignal(SIGINT));
```

[TLPI §20.11]

Signals are not queued

- The set of pending (standard) signals is a mask
- ⇒ If same signal is generated multiple times while blocked, it will be delivered just once

Exercises

The goal of these exercises is to experiment with signal handlers and the use of the signal mask to block delivery of signals. A template for both part 1 and part 2 of the exercise is provided ([template: `signals/ex.pending_sig_expt.c`])

Hint: don't confuse the signal mask with the *sa_mask* field that is passed to *sigaction()*. The signal mask is a process attribute maintained inside the kernel that can be directly modified using calls to *sigaction()*. The *sa_mask* field specifies additional signals that should be *temporarily* added to the signal mask while a signal handler is executing.

1. Write a program that:
   - Blocks all signals except `SIGINT`. This will require the use of *sigprocmask()* (slides 5-31 + 5-32) as well as the APIs for manipulating signal sets (slide 5-28).
   - Uses *sigaction()* (slides 5-13, 5-14, and 5-23) to establish a `SIGINT` handler that does nothing but return.
   - Calls *pause()* to wait for a signal.

Exercises

   - After *pause()* returns:
     - determines the set of pending signals for the process (use *sigpending()*, slide 5-33);
     - tests which signals are in that set (use *sigismember()*, iterating through all signals in the range `1 <= s < NSIG`; see slide 5-18);
     - and prints the descriptions of those signals (*strsignal()*).

   Run the program and send it various signals (other than `SIGINT` and signals that are ignored by default), using either the *kill* command from another terminal (`kill -<sig> <pid>`), or by typing signal-generating keys from the terminal where you run the program (Control-Z for `SIGTSTP`, Control-\for `SIGQUIT`). Then type Control-C to generate `SIGINT` and inspect the list of pending signals displayed by the program.

   [Exercises continue on following slide]

Exercises

2. Extend the program created in the preceding exercise so that:
   - Just after installing the handler for `SIGINT`, the program also installs a handler for `SIGQUIT` (generated when the Control-\ key is pressed). The handler should print a message "SIGQUIT received", and return.
   - After displaying the list of pending signals, the program unblocks `SIGQUIT` and calls *pause()* once more. (⚠ Which *how* value should be given to *sigprocmask()*?)

   While the program is blocking signals (i.e., before typing Control-C), try typing Control-\ multiple times. After Control-C is typed, how many times does the `SIGQUIT` handler display its message? Why?

3. If you run the program once more, and then from another terminal send the `SIGKILL` signal to the program (`kill -KILL <pid>`), what happens? Why?

Outline

Designing signal handlers

- Signal handlers can, in theory, do anything
- But, complex signal handlers can easily have subtle bugs (e.g., race conditions)
  - E.g., if main program and signal handler access same global variables
- ⇒ Design handlers to be as simple as possible

Designing signal handlers

- Some simple signal-handler designs:
  - Set a global flag and return
    - Main program periodically checks (and clears) flag, and takes appropriate action
    - (See the discussion of *sig_atomic_t* in TLPI §21.1.3)
  - Signal handler does some clean-up and terminates process
    - (TLPI §21.2)
  - Signal handler performs a nonlocal goto to unwind stack
    - *sigsetjmp()* and *siglongjmp()* (TLPI §21.2.1)
    - E.g., some shells do this when handling signals

Signals are not queued

- Signals are not queued
- A blocked signal is marked just once as pending, even if generated multiple times
- ⇒ **One signal may correspond to multiple "events"**
  - Programs that handle signals must be designed to allow for this
- Example:
  - `SIGCHLD` is generated for parent when child terminates
  - While `SIGCHLD` handler executes, `SIGCHLD` is blocked
  - Suppose **two** more children terminate while handler executes
  - Only one `SIGCHLD` signal will be queued
  - Solution: `SIGCHLD` handler should loop, checking if multiple children have terminated

Notes

Notes

Notes

Linux System Programming Essentials

Process Lifecycle

Michael Kerrisk, man7.org © 2024

January 2024

mtk@man7.org

Outline

Outline

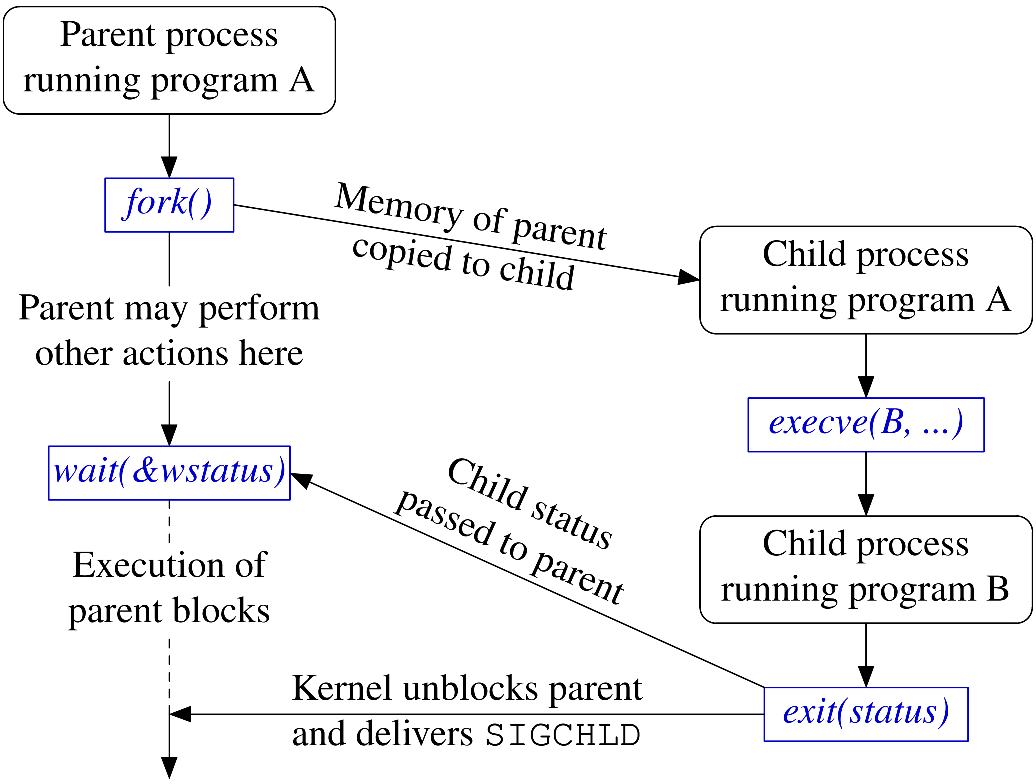
Creating processes and executing programs

Four key system calls (and their variants):

- *fork()*: create a new ("child") process
- *exit()*: terminate calling process
- *wait()*: wait for a child process to terminate
- *execve()*: execute a new program in calling process

[TLPI §24.1]

Using *fork()*, *execve()*, *wait()*, and *exit()* together



Outline

Creating a new process: *fork()*

```
#include <unistd.h>
pid_t fork(void);
```

fork() creates a new process ("**the child**")

- Child is a **near exact duplicate of caller** ("the parent")
- Notionally, memory of parent is duplicated to create child
  - In practice, copy-on-write duplication is used
    - ⇒ Only page tables must be duplicated at time of *fork()*
- Two processes share same (read-only) text segment
- Two processes have separate copies of stack, data, and heap segments
  - ⇒ Each process can modify variables without affecting other process

[TLPI §24.2]

Return value from *fork()*

```
#include <unistd.h>
pid_t fork(void);
```

- **Both** processes continue execution by returning from *fork()*
- *fork()* returns different values in parent and child:
  - Parent:
    - On success: PID of new child (allows parent to track child)
    - On failure: –1
  - Child: returns 0
    - Child can obtain its own PID using *getpid()*
    - Child can obtain PID of parent using *getppid()*

Using *fork()*

Common idioms for using *fork()*:

```
pid_t pid = fork();

if (pid == -1) {
   /* Handle error */
} else if (pid == 0) {
   /* Code executed by child */
} else {
   /* Code executed by parent */
}
```

```
pid_t pid = fork();
switch (pid) {
case -1:
   /* Handle error */
case 0:
   /* Code executed by child */
default:
   /* Code executed by parent */
}
```


A Linux-specific alternative: *clone()*

- *clone()*/*clone3()* is another way of creating a process
- Much more flexibility than *fork()* (multiple arguments)
- Features include:
  - Parent and child may share various attributes (threads!)
    - Process ID
    - File descriptors
    - Virtual address space
    - Signal dispositions
  - Create new namespaces
  - Can obtain PID file descriptor that refers to child (*clone3()*)
    - Can wait/signal via PID FD
- Used to implement *pthread_create()* (and, in glibc, *fork()*!)

Exercise

1. Write a program that uses *fork()* to create a child process ([template: `procexec/ex.fork_var_test.c`]). After the *fork()* call, both the parent and child should display their PIDs (*getpid()*). Include code to demonstrate that the child process created by *fork()* can modify its copy of a local variable in *main()* without affecting the value in the parent's copy of the variable.

   Note: you may find it useful to use the *sleep(num-secs)* library function to delay execution of the parent for a few seconds, to ensure that the child has a chance to execute before the parent inspects its copy of the variable.

2. Processes have many attributes. When a new process is created using *fork()*, which of those attributes are inherited by the child and which are not (e.g., are reset to some default)? Here, we explore whether two process attribute–signal dispositions and alarm timers–are inherited by a child process.

   [Exercise continues on the next slide]

Exercise

Write a program ([template: `procexec/ex.inherit_alarm.c`]) that performs the following steps in order to determine if a child process inherits signal dispositions and alarm timers from the parent:

- Establishes a `SIGALRM` handler that prints the process's PID.
- Starts an alarm timer that expires after two seconds. Do this using the call *alarm(2)*. When the timer expires, it will notify by sending a `SIGALRM` signal to the process.
- Creates a child process using *fork()*.
- After the *fork()*, the child fetches the disposition of the `SIGALRM` signal (*sigaction()*) and tests whether the *sa_handler* field in the returned structure is the address of the signal handler
- Both processes then loop 5 times, sleeping for half a second (use *usleep()*) and displaying the process PID. Which of the processes receives a `SIGALRM` signal?

Outline

Terminating a process

A process can terminate itself using two APIs:

- *_exit(2)* (system call)
- *exit(3)* (library function)

[TLPI §25.1]

Terminating a process with *_exit(2)*

```
#include <unistd.h>
void _exit(int status);
```

_exit() terminates the calling process

- AKA **normal termination**
  - **abnormal termination** == killed by a signal
- (In truth: on Linux, *_exit()* is a wrapper for Linux-specific *exit_group(2)*, which terminates all threads in a process)

Process exit status

```
#include <unistd.h>
void _exit(int status);
```

- Least significant 8 bits of *status* define **exit status**
  - Remaining bits ignored
  - 0 == success
  - nonzero == failure
- POSIX specifies two constants:

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

Terminating a process with *exit(3)*

- Most programs employ *exit(3)*, rather than *_exit(2)*

```
#include <stdlib.h>
void exit(int status);
```

- The *exit(3)* library function:
  - Calls exit handlers registered by process
    - Exit handler == callback function automatically called at normal process termination
    - *atexit(3)*, *on_exit(3)*
  - Flushes *stdio* buffers
  - Calls: `_exit(status)`
    - (If we call *_exit()* directly, then exit handlers are **not** called and *stdio* buffers are **not** flushed)
- `return n` inside *main()* is equivalent to `exit(n)`

Process teardown

As part of process termination (normal or abnormal), the kernel performs various clean-ups:

- All open **file descriptors** are closed
  - Associated **file locks** are released
- Open **POSIX IPC objects** are closed (message queues, semaphores, shared memory)
- **Memory mappings** are unmapped
- **Memory locks** are removed
- **System V shared memory segments** are detached
- And more...

[TLPI §25.2]

Outline

Overview

- Parent processes can use the "wait" family of system calls to monitor state change events in child processes:
  - Termination
  - Stop (because of a signal)
  - Continue (after `SIGCONT` signal)
- Parent can obtain various info about state changes:
  - Exit status of process
  - What signal stopped or killed process
  - Whether process produced a core dump before terminating
- For historical reasons, there are multiple "wait" functions

Waiting for children with *waitpid()*

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- *waitpid()* waits for a child process to change state
  - No child has changed state ⇒ call blocks
  - Child has already changed state ⇒ call returns immediately
- *wstatus* argument returns **wait status** value that describes child state transition
  - *wstatus* can be `NULL`, if we don't need this info
  - (More details later)
- Return value:
  - On success: PID of child whose status is being reported
  - On error, –1
    - No more children? ⇒ *errno* set to `ECHILD`

[TLPI §26.1.2]

Waiting for children with *waitpid()*

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

pid specifies which child(ren) to wait for:

- *pid* $== -1$: **any** child of caller
- *pid* > 0: child whose **PID** equals *pid*
- (plus other possibilities, as documented in manual page)

Waiting for children with *waitpid()*

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- By default, *waitpid()* reports only **terminated** children
- The *options* bit mask can specify additional state changes to report:
  - `WUNTRACED`: report **stopped** children
  - `WCONTINUED`: report stopped children that have **continued**
- Specifying `WNOHANG` in *options* causes **nonblocking** wait
  - If no children have changed state, *waitpid()* returns immediately, with return value of 0

waitpid() example

Wait for all children to terminate, and report their PIDs:

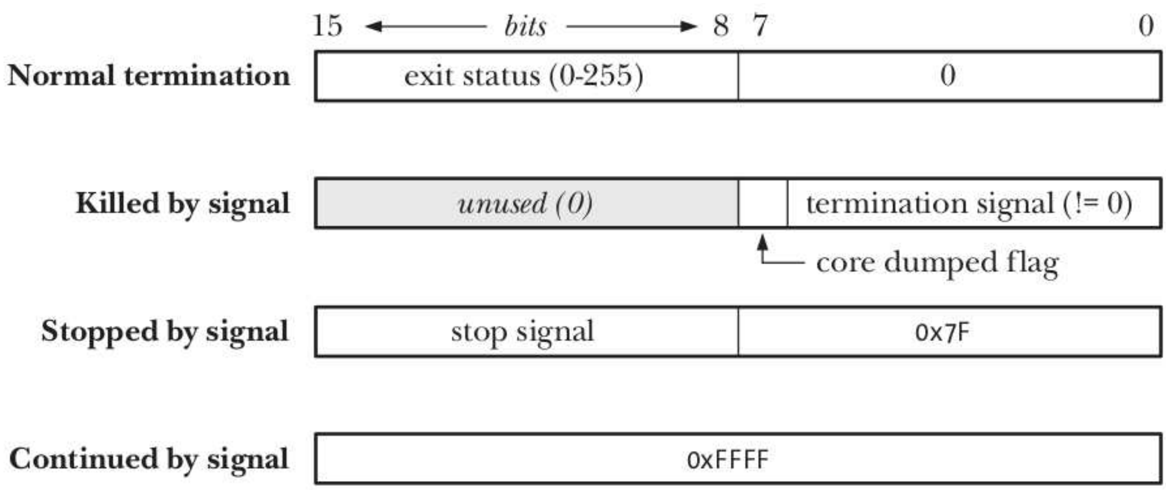
```
for (;;) {
    childPid = waitpid(-1, NULL, 0);
    if (childPid == -1) {
        if (errno == ECHILD) {
            printf("No more children!\n");
            break;
        } else {            /* Unexpected error */
            errExit("waitpid");
        }
    }

    printf("waitpid() returned PID %ld\n", (long) childPid);
}
```

The wait status value

- *wstatus* distinguishes 4 types of event:
  - Child **terminated via *_exit()***, specifying an *exit status*
  - Child was **killed by a signal**
  - Child was **stopped by a signal**
  - Child was **continued by a signal**

The wait status value

16 lowest bits of *wstatus* returned by *waitpid()* encode status in such a way that the 4 cases can be distinguished:



(Encoding is an implementation detail we don't really need to care about)

Dissecting the wait status

- `<sys/wait.h>` defines macros for dissecting a wait status
- Only one of the headline macros in this list will return true:
  1. `WIFEXITED(wstatus)`: true if child exited normally
     - `WEXITSTATUS(wstatus)` returns exit status of child
  2. `WIFSIGNALED(wstatus)`: true if child was killed by signal
     - `WTERMSIG(wstatus)` returns number of killing signal
     - `WCOREDUMP(wstatus)` returns true if child dumped core
  3. `WIFSTOPPED(wstatus)`: true if child was stopped by signal
     - `WSTOPSIG(wstatus)` returns number of stopping signal
  4. `WIFCONTINUED(wstatus)`: true if child was resumed by `SIGCONT`
- The subordinate macros may be used only if the corresponding headline macro tests true

Example: procexec/print_wait_status.c

Display wait status value in human-readable form

```
void printWaitStatus(const char *msg, int status) {
    if (msg != NULL)
        printf("%s", msg);

    if (WIFEXITED(status)) {
        printf("child exited, status=%d\n", WEXITSTATUS(status));

    } else if (WIFSIGNALED(status)) {
        printf("child killed by signal %d (%s)",
                WTERMSIG(status), strsignal(WTERMSIG(status)));
        if (WCOREDUMP(status))
            printf(" (core dumped)");
        printf("\n");

    } else if (WIFSTOPPED(status)) {
        printf("child stopped by signal %d (%s)\n",
                WSTOPSIG(status), strsignal(WSTOPSIG(status)));

    } else if (WIFCONTINUED(status))
        printf("child continued\n");
}
```

An older wait API: *wait()*

```
#include <sys/wait.h>
pid_t wait(int *wstatus);
```

- The original “wait” API
- Equivalent to: `waitpid(-1, &wstatus, 0);`
- Still commonly used to handle the simple, common case: **wait for any child to terminate**

An newer wait API: *waitid()*

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- Similar to *waitpid()*, but provides additional functionality, including:
  - Independently choose which events (termination / stopped / continued) to wait on
    - *waitpid()* always waits for at least termination events
  - Wait via PID file descriptor

Outline

Orphans

- An **orphan** is a process that lives longer than its parent
- Orphaned processes are **adopted by *init***
- ***init* waits for its adopted children** when they terminate
- After orphan is adopted, *getppid()* returns PID of *init*
  - Conventionally, *init* has PID 1
- On systems where the *init* system is *systemd*, then, depending on the configuration, things are different:
  - A helper process (PID != 1) becomes parent of orphaned children
    - When run with the *--user* option, *systemd* organizes all processes in the user's session into a subtree with such a subreaper
  - See discussion of `PR_SET_CHILD_SUBREAPER` in *prctl(2)*

[TLPI §26.2]

Zombies

- Suppose a **child terminates before parent waits** for it
- Parent must still be able to collect status later
- ⇒ Child becomes a **zombie**:
  - Most process resources are recycled
  - A process slot is retained
    - PID, status, and resource usage statistics
- Zombie is removed when parent does a “wait”

[TLPI §26.2]

Creating a zombie: `procexec/zombie.c`

Usage: `zombie [num-zombies [sleep-secs]]`

```
int main(int argc, char *argv[]) {
    int nzombies =  (argc > 1) ? atoi(argv[1]) : 1;
    int sleepSecs = (argc > 2) ? atoi(argv[2]) : 0;
    printf("Parent (PID %ld)\n", (long) getpid());

    for (int j = 0; j < nzombies; j++) {
        switch (fork()) {
        case -1:
            errExit("fork-%d", j);
        case 0:                    /* Child: exits to become zombie */
            printf("Child  (PID %ld) exiting\n", (long) getpid());
            if (sleepSecs > 0);
                sleep(sleepSecs);
            exit(EXIT_SUCCESS);
        default:                   /* Parent continues in loop */
            break;
        }
    }
    sleep(3600);                   /* Children are zombies during this time */
    while (wait(NULL) > 0)         /* Reap zombie children */
        continue;
    exit(EXIT_SUCCESS);
}
```

- Create one or more zombie child processes

Creating a zombie: `procexec/zombie.c`

```
$ ./zombie &
[1] 23425
Parent (PID 23425)
Child  (PID 23427) exiting
$ ps -C zombie
  PID TTY          TIME CMD
23425 pts/1    00:00:00 zombie
23427 pts/1    00:00:00 zombie <defunct>
$ kill -KILL 23427
$ ps -C zombie
  PID TTY          TIME CMD
23425 pts/1    00:00:00 zombie
23427 pts/1    00:00:00 zombie <defunct>
```

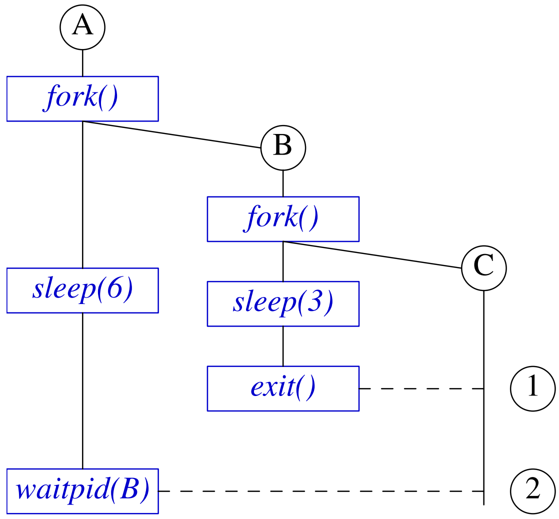
- **Zombies can't be killed** by signals!
  - (Since parent must still be able to "wait")
  - Even silver bullets (`SIGKILL`) don't work

Reap your zombies

- **Zombie may live for ever**, if parent fails to "wait" on it
  - Or until parent is killed, so zombie is adopted by *init*
- **Long-lived processes that create children must ensure that zombies are "reaped"** ("waited" for)
  - Shells, network servers, ...

Exercise

1. Suppose that we have three processes related as grandparent (A), parent (B), and child (C), and that the parent exits after a few seconds, but the grandparent does **not** immediately perform a *wait()* after the parent exits, with the result that the parent becomes a zombie, as in the following diagram.



Exercise

When do you expect the child (C) to be adopted by *init* (so that *getppid()* in the child returns 1): after the parent (B) terminates or after the grandparent (A) does a *wait()*? In other words, is the child adopted at point 1 or point 2 in the diagram? Write a program, [(minimal) template: `procexec/ex.zombie_parent.c`], to verify the answer.

Note the following points:

- For a reminder of the usage of *fork()*, see slide 6-9.
- You will need to use to *sleep()* in various parts of the program:
  - The child (C) could loop 10 times, displaying the value returned by *getppid()* and sleeping for 1 second on each loop iteration.
  - The parent (B) sleeps for 3 seconds before terminating.
  - The grandparent (A) sleeps for 6 seconds before calling *waitpid()* on the PID of the parent (B).
- Depending on your distribution (e.g., if you have a *systemd*-based system where the *--user* flag is employed), you may find that the orphaned child is reparented to a process other than PID 1. Find out what program is running in that process, by using the command *ps <pid>*.

Outline

The `SIGCHLD` signal

- `SIGCHLD` is generated for a parent when a child terminates
- Ignored by default
- Catching `SIGCHLD` allows us to be asynchronously notified of child's termination
  - Can be more convenient than synchronous or nonblocking *waitpid()* calls
- Within `SIGCHLD` handler, we "wait" to reap zombie child

[TLPI §26.3]

A SIGCHLD handler

```
void grimReaper(int sig) {
    int savedErrno = errno;
    while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
    errno = savedErrno;
}
```

- Each *waitpid()* call reaps one terminated child
- while loop handles possibility that multiple children terminated while SIGCHLD was blocked
  - e.g., during earlier invocation of handler
- WNOHANG ensures handler does not block if there are no more terminated children
- Loop terminates when *waitpid()* returns:
  - 0, meaning no more *terminated* children
  - −1, probably with *errno == ECHILD*, meaning no more children
- Save and restore *errno*, so that handler is reentrant (TLPI p427)

SIGCHLD for stopped and continued children

- SIGCHLD is also generated when a child stops or continues
- To prevent this, specify SA_NOCLDSTOP in *sa_flags* when establishing SIGCHLD handler with *sigaction()*

[TLPI §26.3.2]

Outline

Executing a new program

execve() loads a new program into caller process's memory

- Old program, stack, data, and heap are discarded
- After executing run-time start-up code, execution commences in new program's *main()*
- Various functions layered on top of *execve()*:
  - Provide variations on functionality of *execve()*
  - Collectively termed "*exec()*"
    - See *exec(3)* manual page

[TLPI §27.1]

Executing a new program with *execve()*

```
#include <unistd.h>
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

- *execve()* loads program at *pathname* into caller's memory
- *pathname* is an absolute or relative pathname

Executing a new program with *execve()*

```
#include <unistd.h>
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

- *argv* specifies command-line arguments for new program
  - Defines *argv* argument for *main()* in new program
  - `NULL`-terminated array of pointers to strings
- *argv[0]* is command name
  - Typically, same as (basename part of) *pathname*
  - Program can vary its behavior, depending on value of *argv[0]* (e.g., *busybox*)
  - See example programs
    - `procexec/launch_shell.c` (value in *argv[0]* triggers login shell behavior)
    - `procexec/execve_argv_expt.c`

Executing a new program with *execve()*

```
#include <unistd.h>
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

- *envp* specifies environment list for new program
  - Defines *environ* in new program
  - `NULL`-terminated array of pointers to strings

Executing a new program with *execve()*

```
#include <unistd.h>
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

- Successful *execve()* does not return
- If *execve()* returns, it failed; no need to check return value:

```
execve(pathname, argv, envp);
perror("execve");
exit(EXIT_FAILURE);
```

Example: procexec/exec_status.c

```
./exec_status command [args...]
```

- Create a child process
- Child executes *command* with supplied command-line arguments
- Parent waits for child to terminate, and reports wait status

Example: procexec/exec_status.c

```
extern char **environ;
int main(int argc, char *argv[]) {
    pid_t childPid, wpid;
    int wstatus;
    ...
    switch (childPid = fork()) {
    case -1: errExit("fork");

    case 0:      /* Child */
        printf("PID of child: %ld\n", (long) getpid());
        char **nextArgv = &argv[1];          // argv for next program
        char *progname = nextArgv[0];
        execve(progName, nextArgv, environ);
        errExit("execve");

    default:     /* Parent */
        wpid = waitpid(childPid, &wstatus, 0);
        if (wpid == -1) errExit("waitpid");
        printf("Wait returned PID %ld\n", (long) wpid);
        printWaitStatus("        ", wstatus);
    }
    exit(EXIT_SUCCESS);
}
```

Example: procexec/exec_status.c

```
$ ./exec_status /bin/date
PID of child: 4703
Thu Oct 24 13:48:44 NZDT 2013
Wait returned PID 4703
        child exited, status=0
$ ./exec_status /bin/sleep 60 &
[1] 4771
PID of child: 4773
$ kill 4773
Wait returned PID 4773
        child killed by signal 15 (Terminated)
[1]+  Done           ./exec_status /bin/sleep 60
```


Exercise

1. Write a simple shell program. The program should loop, continuously reading shell commands from standard input. Each input line consists of a set of white-space delimited words that are a command and its arguments. Each command should be executed in a new child process (*fork()*) using *execve()*. The parent process (the "shell") should wait on each child and display its wait status (you can use the supplied *printWaitStatus()* function). [template: `procexec/ex.simple_shell.c`]

   Some hints:

   - The space-delimited words in the input line need to be broken down into a set of null-terminated strings pointed to by an *argv*-style array, and that array must end with a `NULL` pointer. The *strtok(3)* library function simplifies this task. (This task is already performed by code in the template.)
   - Because *execve()* is used, you will need to type the full pathname when entering commands to your shell

   Fun facts: the source code of *bash* is around 180k lines (*dash* is around 20k lines)

Exercise

2. Write a program, `procexec/exec_self_pid.c`, that verifies that an exec does not change a process's PID
   - The program should perform the following steps:
     - Print the process's PID.
     - If *argc* is 2, the program exits.
     - Otherwise, the program uses *execl()* to re-execute itself with an additional command-line argument (any string), so that *argc* will be 2.
   - Test the program by running it with no arguments (i.e., *argc* is 1).
3. Write a program ([template: `procexec/ex.make_link.c`]) that takes 2 arguments:

```
make_link target linkpath
```

If invoked with the name *slink*, it creates a symbolic link (*symlink()*) using these pathnames, otherwise it creates a hard link (*link()*). After compiling, create two hard links to the executable, with the names *hlink* and *slink*. Verify that when run with the name *hlink*, the program creates hard links, while when run with the name *slink*, it creates symbolic links.
Hint:

- You will find the *basename()* and *strcmp()* functions useful when inspecting the program name in *argv[0]*.

The *exec()* library functions

```
#include <unistd.h>
int execle(const char *pathname, const char *arg, ...
        /* , (char *) NULL, char *const envp[] */ );
int execlp(const char *filename, const char *arg, ...
        /* , (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
        /* , (char *) NULL */);
int execvpe(const char *filename, const *char argv[],
        char *const envp[]);
```

- Variations on theme of *execve()*
- Like *execve()*, the *exec()* functions return only if they fail
- *execvpe()* is Linux-specific (define `_GNU_SOURCE`)

The *exec()* library functions

Vary theme of *execve()* with 2 choices in each of 3 dimensions:

- How are command-line arguments of new program specified?
- How is the executable specified?
- How is environment of new program specified?

Final letters in name of each function are clue about behavior

Function	**Specification of arguments (v, l)**	**Specification of executable file (-, p)**	**Source of environment (e, -)**
execve()	array	pathname	*envp* argument
execle()	list	pathname	*envp* argument
execlp()	list	filename + `PATH`	caller's *environ*
execvp()	array	filename + `PATH`	caller's *environ*
execv()	array	pathname	caller's *environ*
execl()	list	pathname	caller's *environ*
execvpe()	array	filename + `PATH`	*envp* argument

Notes

Linux System Programming Essentials

System Call Tracing with *strace*

Michael Kerrisk, man7.org © 2024

January 2024

mtk@man7.org

Outline

Rev: # 9feddaa40cef

Outline

strace(1)

- A tool to trace system calls made by a user-space process
  - Implemented via *ptrace(2)*
- Or: a debugging tool for tracing **complete conversation between application and kernel**
  - Application source code is not required
- Answer questions like:
  - What system calls are employed by application?
  - Which files does application touch?
  - What arguments are being passed to each system call?
  - Which system calls are failing, and why (*errno*)?

strace(1)

- Trace information is provided in **symbolic form**
  - **System call names** are shown
  - We see **signal names** (not numbers)
  - **Strings** printed as characters (up to 32 bytes, by default)
  - **Bit-mask arguments displayed symbolically**, using corresponding bit flag names ORed together
  - **Structures** displayed with **labeled fields**
  - "Large" arguments are abbreviated by default
    - Use *strace –v* (verbose) to see unabbreviated arguments

strace(1)

```
fstat(3, {st_dev=makedev(0x8, 0x5), st_ino=407279,
  st_mode=S_IFREG|0755, st_nlink=1, st_uid=0, st_gid=0,
  st_blksize=4096, st_blocks=80, st_size=36960, st_atime=1625615479
  /* 2021-07-07T01:51:19.795021222+0200 */, st_atime_nsec=795021222,
  st_mtime=1613345143 /* 2021-02-15T00:25:43+0100 */, st_mtime_nsec=0,
  st_ctime=1616161103 /* 2021-03-19T14:38:23.816838407+0100 */,
  st_ctime_nsec=816838407}) = 0

open("/lib64/liblzma.so.5", O_RDONLY|O_CLOEXEC) = 3

access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
  directory)
```

For each system call, we see:

- Name of system call
- Values passed in/returned via arguments
- System call return value
- Symbolic *errno* value (+ explanatory text) on syscall failures

Simple usage: tracing a command at the command line

- A very simple C program:

```
int main(int argc, char *argv[]) {
#define STR "Hello world\n"
    write(STDOUT_FILENO, STR, strlen(STR));
    exit(EXIT_SUCCESS);
}
```

- Run *strace(1)*, directing logging output (*–o*) to a file:

```
$ strace -o strace.log ./hello_world
Hello world
```

  - (By default, trace output goes to standard error)
- ⚠ On some systems, may first need to to ensure `ptrace_scope` file has value 0 or 1:

```
$ sudo sh -c 'echo 0 > /proc/sys/kernel/yama/ptrace_scope'
```

  - Yama LSM disables *ptrace(2)* to prevent attack escalation; see *ptrace(2)* manual page

Simple usage: tracing a command at the command line

```
$ cat strace.log
execve("./hello_world", ["./hello_world"], [/* 110 vars */]) = 0
...
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=160311, ...}) = 0
mmap(NULL, 160311, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fa5ecfc0000
close(3)                                = 0
open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
...
write(1, "Hello world\n", 12)           = 12
exit_group(0)                           = ?
+++ exited with 0 +++
```

- Even simple programs make lots of system calls!
  - 25 in this case (many have been edited from above output)
- Most output in this trace relates to finding and loading shared libraries
  - First call (*execve()*) was used by shell to load our program
  - Only last two system calls were made by our program

A gotcha...

- The last call in our program was:

```
exit(EXIT_SUCCESS);
```

- But *strace* showed us:

```
exit_group(0)                           = ?
```

- Some detective work:
  - We "know" *exit(3)* is a library function that calls *_exit(2)*
  - But where did *exit_group()* come from?
  - *_exit(2)* manual page tells us:

```
$ man 2 _exit
...
C library/kernel differences
 In glibc up to version 2.3, the _exit() wrapper function
 invoked  the kernel system call of the same name.  Since
 glibc 2.3, the wrapper function  invokes  exit_group(2),
 in order to terminate all of the threads in a process.
```

- ⇒ may need to dig deeper to understand *strace(1)* output

Tracing live processes

- *–p PID*: **trace running process** with specified PID
  - Type *Control-C* to cease tracing
  - To **trace multiple processes**, specify *–p* multiple times
  - Can trace only processes you own
    - (And a process can have only one tracer)
  - ⚠ ⚠ tracing a process can **heavily affect performance**
    - E.g., up to two orders of magnitude slow-down in syscalls
    - ⚠ Think twice before using in a production environment
- *–p PID –f*: will **trace all threads** in specified process

Outline

Tracing child processes

- By default, *strace* does not trace children of traced process
- *–f* option causes children to be traced
  - Each trace line is prefixed by PID
  - In a program that employs POSIX threads, each line shows kernel thread ID (*gettid()*)

Tracing child processes: `strace/fork_exec.c`

```c
int main(int argc, char *argv[]) {
    pid_t childPid;
    char *newEnv[] = {"ONE=1", "TWO=2", NULL};

    printf("PID of parent: %ld\n", (long) getpid());
    childPid = fork();
    if (childPid == 0) {          /* Child */
        printf("PID of child:  %ld\n", (long) getpid());
        if (argc > 1) {
            execve(argv[1], &argv[1], newEnv);
            errExit("execve");
        }
        exit(EXIT_SUCCESS);
    }
    wait(NULL);          /* Parent waits for child */
    exit(EXIT_SUCCESS);
}
```

```
$ strace -f -o strace.log ./fork_exec
PID of parent: 1939
PID of child:  1940
```


Tracing child processes: `strace/fork_exec.c`

```
$ cat strace.log
1939 execve("./fork_exec", ["./fork_exec"], [/* 110 vars */]) = 0
...
1939 clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
  child_tidptr=0x7fe484b2ea10) = 1940
1939 wait4(-1,  <unfinished ...>
1940 write(1, "PID of child:  1940\n", 21) = 21
1940 exit_group(0)                     = ?
1940 +++ exited with 0 +++
1939 <... wait4 resumed> NULL, 0, NULL) = 1940
1939 --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=1940,
  si_uid=1000, si_status=0, si_utime=0, si_stime=0} ---
1939 exit_group(0)                     = ?
1939 +++ exited with 0 +++
```

- Each line of trace output is prefixed with corresponding PID
- Inside glibc, *fork()* is actually a wrapper that calls *clone(2)*
- *wait()* is a wrapper that calls *wait4(2)*
- We see two lines of output for *wait4()* because call blocks and then resumes
- *strace* shows us that parent received a `SIGCHLD` signal

Exercises

1. Try using *strace* to trace the execution of a program of your choice.
2. Some amusements:
   - `strace -p $$`
   - `strace strace -p $$`

Outline

Selecting system calls to be traced

- *strace –e* can be used to select system calls to be traced
- *–e trace=<syscall>[,<syscall>...]*
  - Specify system call(s) that should be traced
  - Other system calls are ignored

```
$ strace -o strace.log -e trace=open,close ls
```

- *–e trace=!<syscall>[,<syscall>...]*
  - **Exclude** specified system call(s) from tracing
    - Some applications do bizarre things (e.g., calling *gettimeofday()* 1000s of times/sec.)
  - ⚠ "!" needs to be quoted to avoid shell interpretation
- *–e trace=/<regexp>*
  - Trace syscalls whose names match regular expression
    - April 2017; expression will probably need to be quoted...

Selecting system calls by category

- *–e trace=<syscall-category>* trace a category of syscalls
- Categories include:
  - *%file*: trace all syscalls that take a filename as argument
    - *open()*, *stat()*, *truncate()*, *chmod()*, *setxattr()*, *link()*...
  - *%desc*: trace file-descriptor-related syscalls
    - *read()*, *write()*, *open()*, *close()*, *fsetxattr()*, *poll()*, *select()*, *pipe()*, *fcntl()*, *epoll_create()*, *epoll_wait()*...
  - *%process*: trace process management syscalls
    - *fork()*, *clone()*, *exit_group()*, *execve()*, *wait4()*, *unshare()*...
  - *%network*: trace network-related syscalls
    - *socket()*, *bind()*, *listen()*, *connect()*, *sendmsg()*...
  - *%signal*: trace signal-related syscalls
    - *kill()*, *rt_sigaction()*, *rt_sigprocmask()*, *rt_sigqueueinfo()*...
  - *%memory*: trace memory-mapping-related syscalls
    - *mmap()*, *mprotect()*, *mlock()*...

Filtering signals

- *strace –e signal=set*
  - Trace only specified set of signals
  - "sig" prefix in names is optional; following are equivalent:

```
$ strace -e signal=sigio,sigint ls > /dev/null
$ strace -e signal=io,int ls > /dev/null
```

- *strace –e signal=!set*
  - Exclude specified signals from tracing

Filtering by pathname

- *strace –P pathname*: trace only system calls that access file at *pathname*
  - Specify multiple *–P* options to trace multiple paths
- Example:

```
$ strace -o strace.log -P /lib64/libc.so.6 ls > /dev/null
Requested path '/lib64/libc.so.6' resolved into '/usr/lib64/libc-2.18.so'
$ cat strace.log
open("/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p\36\2\0\0\0\0\0"...,
  832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2093096, ...}) = 0
mmap(NULL, 3920480, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE,
  3, 0) = 0x7f8511fa3000
mmap(0x7f8512356000, 24576, PROT_READ|PROT_WRITE,
  MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b3000) = 0x7f8512356000
close(3)                                 = 0
+++ exited with 0 +++
```

  - *strace* noticed that the specified file was opened on FD 3, and also traced operations on that FD

Mapping file descriptors to pathnames

- *–y* option causes *strace* to display pathnames corresponding to each file descriptor
  - Useful info is also displayed for other types of file descriptors, such as pipes and sockets

```
$ strace -y cat greet
...
openat(AT_FDCWD, "greet", O_RDONLY)    = 3</home/mtk/greet>
fstat(3</home/mtk/greet>, {st_mode=S_IFREG|0644, ...
read(3</home/mtk/greet>, "hello world\n", 131072) = 12
write(1</dev/pts/11>, "hello world\n", 12) = 12
read(3</home/mtk/greet>, "", 131072) = 0
close(3</home/mtk/greet>)    = 0
...
```

- *–yy* is as for *–y* but shows additional protocol-specific info for sockets

```
write(3<TCP:[10.0.20.135:33522->213.131.240.174:80]>,
"GET / HTTP/1.1\r\nUser-Agent: Wget"..., 135) = 135
read(3<TCP:[10.0.20.135:33522->213.131.240.174:80]>,
"HTTP/1.1 200 OK\r\nDate: Thu, 19 J"..., 253) = 253
```


Outline

System call tampering

- *strace* can be used to **modify** behavior of selected syscall(s)
  - Initial feature implementation completed in early 2017
- Various possible effects:
  - Inject delay before/after syscall
  - Generate a signal on syscall
  - Bypass execution of syscall, making it return a "success" value or fail with specified value in *errno* (error injection)
  - (Limited) ability to choose which invocation of syscall will be modified

`strace -e inject` options

- Syntax: `strace -e inject=`*<syscall-set>[:<option>]...*
  - *syscall-set* is set of syscalls whose behavior will be modified
- `:error=`*errnum*: syscall is not executed; returns failure status with *errno* set as specified
- `:retval=`*value*: syscall is not executed; returns specified "success" value
  - Can't specify both `:retval` and `:error` together

`strace -e inject` options

- `:signal=`*sig*: deliver specified signal on entry to syscall
- `:delay_enter=`*usecs*, `:delay_exit=`*usecs*: delay for *usecs* microseconds on entry to/return from syscall
- `:when=`*expr*: specify which invocation(s) to tamper with
  - `:when=`*N*: tamper with invocation *N*
  - `:when=`*N+*: tamper starting at *N*th invocation
  - `:when=`*N+S*: tamper with invocation *N*, and then every *S* invocations
  - Range of *N* and *S* is 1..65535

Example

```
$ strace -y -e close -e inject=close:error=22:when=3 /bin/ls > d
close(3</etc/ld.so.cache>)              = 0
close(3</usr/lib64/libselinux.so.1>)    = 0
close(3</usr/lib64/libcap.so.2.25>)     =  -1 EINVAL (Invalid argument) (INJECTED)
close(3</usr/lib64/libcap.so.2.25>)     = 0
/bin/ls: error while loading shared libraries: libcap.so.2:
cannot close file descriptor: Invalid argument
+++ exited with 127 +++
```

- Use *–y* to show pathnames corresponding to file descriptors
- Inject error 22 (`EINVAL`) on third call to *close()*
- Third *close()* was not executed; an error return was injected
  - (After that, *ls* got sad)

Using system call tampering for error injection

- Success-injection example: make *unlinkat()* succeed, without deleting temporary file that would have been deleted
- Error-injection use case: quick and simple black-box testing
  - Does application fail gracefully when encountering unexpected error?
- But there are alternatives for black-box testing:
  - Preloaded library with interposing wrapper function that spoofs a failure (without calling "real" function)
    - Can be more flexible
    - But can't be used with set-UID/set-GID programs
  - Seccomp (secure computing)
    - Generalized facility to block execution of system calls based on system call number and argument values
    - More powerful, but can't, for example cause Nth call to fail

Outline

Obtaining a system call summary

- *strace –c* counts time, calls, and errors for each system call and reports a summary on program exit

```
$ strace -c who > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- --------------
 21.77    0.000648           9        72           alarm
 14.42    0.000429           9        48           rt_sigaction
 13.34    0.000397           8        48           fcntl
  8.84    0.000263           5        48           read
  7.29    0.000217          13        17         2 kill
  6.79    0.000202           6        33         1 stat
  5.41    0.000161           5        31           mmap
  4.44    0.000132           4        31         6 open
  2.89    0.000086           3        29           close
...
------ ----------- ----------- --------- --------- --------------
100.00    0.002976                   442        13 total
```

- Treat time measurements as indicative only, since *strace* adds overhead to each syscall

Further *strace* options

- *–k*: print a stack trace after each traced syscall
- *sudo strace –u <username> prog*: run program with UID and GIDs of specified user
  - Useful when tracing privileged programs, such as set-UID-*root* programs
    - Normally, privileged programs are **not** run with privilege when executed under *strace*

Further *strace* options

- *–v*: don't abbreviate arguments (structures, etc.)
  - Output can be quite verbose...
- *–s strsize*: maximum number of bytes to display for strings
  - Default is 32 characters
  - Pathnames are always printed in full
- Various options show start time or duration of system calls
  - *–t*, *–tt*: prefix each trace line with wall-clock time
    - *–tt* also adds microseconds
  - *–T*: show time spent in syscall
    - But treat as indications only, since *strace* causes overhead on syscalls

Notes

Linux System Programming Essentials

Wrapup

Michael Kerrisk, man7.org

January 2024

mtk@man7.org

Outline

Rev: # 9feddaa40cef

Outline

Course materials

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

Marketing

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

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

Linux/UNIX System Programming (LUSP01, 5 days)

- **System Programming Fundamentals** (SPINTRO01, 2 days) → **Threads and IPC Programming** (TIPC01, 3 days)

System Programming Fundamentals → **IPC Programming** (IPC02, 3 days)

POSIX Threads (PTHR01, 1 day)

System Programming for Linux Containers (SPLC02, 5 days)

- **System Prog. Essentials** (SPESS01, 1d) → **Linux Security and Isolation APIs** (SECISOL02, 4 days)
  - **Capabilities + Namespaces** (CAPNS01, 2 days)
  - **Seccomp** (SECCOMP01, 1d)
  - **Control Groups** (CGROUPS02, 1d)

Linux Shared Libraries (SHLIB04, 2.5 days)

Linux/UNIX Network Programming (NWP02, 3 days)

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

Thanks!

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

PGP fingerprint: 4096R/3A35CE5E

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

Notes