Linux/UNIX System Programming Fundamentals

Michael Kerrisk
man7.org

NDC TechTown; August 2020

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/`.

Short table of contents

Short table of contents

Detailed table of contents

Detailed table of contents

Detailed table of contents

Detailed table of contents

Detailed table of contents

Linux/UNIX System Programming Fundamentals

Course Introduction

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

Outline

Course prerequisites

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

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...
- For programming 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
- For many exercises, I provide **templates** for the solutions
  - Filenames: `ex.*.c`
  - Look for "FIXMEs" to see what parts you must complete
  - ⚠ You will need to edit the corresponding `Makefile` to add a new target for the executable

Lab sessions

- **Pair programming is strongly encouraged!**
  - Pairs typically get through practical sessions faster
    - ⇒ we will go faster as a group, and cover more topics
- **Read each exercise thoroughly** before starting
  - Past experience has shown me the traps that people often fall into with various exercises
  - ⇒ exercise descriptions often include **important hints**
- Solutions will be mailed out shortly after end of course
- Lab sessions are **not** instructor down time...
  - ⇒ One-on-one questions about course material or exercises
- Looking for homework?
  - ⇒ Chapters usually have additional exercises

Lab sessions: some thoughts on building code

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

```
$ while inotifywait -q . ; do echo; make; done
```

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

Outline

Course materials

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

Other resources

- POSIX.1-2001 / SUSv3: http://www.unix.org/version3/
- POSIX.1-2008 / SUSv4: http://www.unix.org/version4/
- Man pages
  - Section 2: system calls
  - Section 3: library functions
  - Latest version online at http://man7.org/linux/man-pages/
  - Latest tarball downloadable at https://www.kernel.org/doc/man-pages/download.html

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)*

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
- KSE: kernel scheduling entity
- IPC: interprocess communication
- 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
    - Documents kernel-user-space + C library APIs
    - Contributor since 2000 (*man-pages-1.31*)
    - As maintainer: ≈20k commits, 188 releases since 2004
    - Author/coauthor of ≈430 out of ≈1040 man 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, where, ...
- What you do with Linux
- Previous knowledge/experience of course topics
- Any special goals for the course

Linux/UNIX System Programming Fundamentals

Fundamental Concepts

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

Outline

System calls

System call == controlled entry point into kernel code

- Request to kernel to perform some task on caller's behalf
- *syscalls(2)* man page lists (nearly) all system calls
- Documented in Section 2 of man pages (notation: *stat(2)*)

[TLPI §3.1]

Steps in the execution of a system call

1. Program calls wrapper function in C library
2. Wrapper function packages syscall arguments into hardware registers
3. Wrapper function puts syscall number into a register
   - Each syscall has a unique number
4. Wrapper function traps to kernel mode
   - e.g., `syscall` instruction on x86-64 (or `sysenter` for 32-bit)
5. Kernel then executes syscall handler:
   - Checks validity of syscall number
   - Invokes **service routine** corresponding to syscall number
     - Checks arguments, **does real work**, returns a result status
   - Places syscall return value in a register
   - Switches back to user mode, passing control back to wrapper
     - E.g., `sysret` instruction on x86-64
6. Wrapper function examines syscall return value; on error, copies return value to *errno*

System calls are expensive!

10^9 calls to...

simple user-space function returning *int* ⇒ 1.5 seconds

getppid() system call ⇒ ~~41~~ 340 seconds

(The page table isolation patches to mitigate Spectre, Meltdown, etc. have caused a real performance hit on system calls)

(*getppid()*, which returns process ID of caller's parent, is one of the simplest system calls)

(Linux 5.4, x86-64; Intel Core i7-8850H; `progconc/syscall_speed.c`)

Library functions

- Library function == one of multitude of functions in Standard C Library
- Diverse range of tasks:
  - I/O
  - Dynamic memory allocation
  - Math
  - String processing
  - etc.
- Documented in Section 3 of man pages (notation: *exit(3)*)
- Some library functions employ system calls
- Many library functions make no use of system calls
- C library provides (simple) wrapper functions for most system calls

[TLPI §3.2]

The C library

- Each C environment has its own implementation of standard C library
- Linux has multiple implementations
- **GNU C library (glibc)** is most widely used
  - Full implementation of POSIX APIs, plus many extensions
  - http://www.gnu.org/software/libc/

[TLPI §3.3]

The C library

- Other Linux C libraries target embedded platforms or the creation of small binaries:
  - musl ("mussel") libc (http://www.musl-libc.org/)
    - Highly active (release 1.0 in 2014)
    - http://wiki.musl-libc.org/wiki/Functional_differences_from_glibc
  - uclibc (http://www.uclibc.org/) [less active?]
  - dietlibc (http://www.fefe.de/dietlibc/) [inactive?]
    - A comparison: http://www.etalabs.net/compare_libcs.html
- (C library on Android is Bionic)
- We'll presume the use of glibc

Outline

Error handling

- Most system calls and library functions return a status indicating success or failure
- Most system calls:
  - Return –1 to indicate error
  - Place integer 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 man 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)

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

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);
}
```


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 $\in$ 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, integer system data types are *long* or smaller
  - ⇒ Promote to *long* when printing system data types

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

- Exception is *off_t*... 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

Example code `lib/` subdirectory

- `lib/` subdirectory contains code of a few functions commonly used in examples
- *camelCase* function name?
  - ⇒ 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
  - Defines `FALSE` and `TRUE`
  - 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);
}
```

- To save some effort, 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 derivatives: *libcap-dev*, *libacl1-dev*, *libselinux1-dev*
    - RPM-based systems: *libcap-devel*, *libacl-devel*, *libselinux-devel*
  - If you can't install these packages, then:

```
cd lib
sh Build_lib.sh      # Ignore any errors you see
```

    and then do *make* in individual directories as needed

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**: 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
- **Method 2**: Add your program at right location in a `Makefile`, and build using `make`

Use of *getopt()* in example programs

- Some example programs use *getopt(3)* to process command-line options
  - *getopt(3)* man page has details, with example of use
  - See also TLPI Appendix B

Notes

Linux/UNIX System Programming Fundamentals

File I/O and Files

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

Outline

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

Created 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
  - We'll see other flags later

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_SYNC`: make file writes synchronous
  - `O_NONBLOCK`: nonblocking I/O
  - More on these later!

open() examples

- Open existing file for reading:

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

- Open new file for read-write, ensuring we are creator:

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

- Open for writing, create if necessary, truncate, always append writes:

```
fd = open("app.log",
      O_WRONLY | O_CREAT | O_TRUNC | O_APPEND,
      S_IRUSR | S_IWUSR);
```

  - (`O_TRUNC` plus `O_APPEND` could be useful if another process is also doing writes at the end of the file)

read(): reading from a file

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

- Arguments:
  - *fd*: file descriptor
  - *buffer*: pointer to buffer to store data
    - ⚠ No terminating null byte is placed at end of buffer
  - *count*: number of bytes to read
    - (*buffer* must be at least this big)
    - (*size_t* and *ssize_t* are 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

write(): writing to a file

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

- Arguments:
  - *fd*: file descriptor
  - *buffer*: pointer to data to be written
  - *count*: number of bytes to write
- Returns:
  - Number of bytes written
    - May be less than *count* (e.g., device full, 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)* man page

Example: copy.c

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

Example: fileio/copy.c (snippet)

Always remember to handle errors!

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

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

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

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; cat f &          # FIFO
$ ./copy mylist f
```

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

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_SOLNS_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:
   - Build `../libtlpi.a` by doing *make* in source code root directory!
   - After first doing some simple command-line testing, test using the unit test in the `Makefile`: `make tee_test`.
   - Remember that you will need to add a target in the `Makefile`!
   - Standard input & output are automatically opened for a process.
   - 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; echo; make; done`
     - You may need to install the *inotify-tools* package
   - Command-line options can be parsed using *getopt(3)*.

Outline

The file offset

Every open file has a **file offset**:

- Location at which next read or write will occur
- Set to byte zero on *open()*
- Automatically updated by *read()*, *write()*, etc.
- Synonyms: read-write offset, file pointer

lseek(): randomly accessing a file

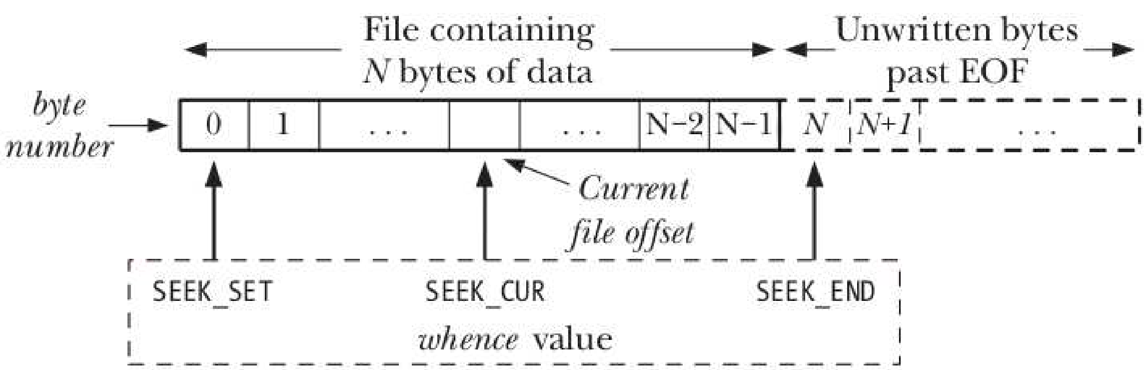
```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

- Adjusts offset for open file referred to by *fd*
  - Some file types not seekable (pipes, sockets, etc.)
- *offset* and *whence* determine new position
  - (*off_t* is an integer type)
- **Returns new file offset** (counted from start of file)

[TLPI §4.7]

lseek(): randomly accessing a file

- *offset*: new offset (byte position)
- *whence*: how to interpret *offset*:
  - `SEEK_SET`: relative start of file
  - `SEEK_CUR`: relative to current position
  - `SEEK_END`: relative to next byte after EOF
- *offset* can be negative for `SEEK_CUR` and `SEEK_END`



lseek() examples

```
lseek(fd, 0, SEEK_SET);
          /* Start of file */
lseek(fd, 1000, SEEK_SET);
          /* Byte 1000 */

lseek(fd, 0, SEEK_END);
          /* First byte past EOF */
lseek(fd, -1, SEEK_END);
          /* Last byte of file */

curr = lseek(fd, 0, SEEK_CUR);
          /* Useful! */
```


Outline

Relationship between file descriptors and open files

- Multiple file descriptors can refer to same open file
- 3 kernel data structures describe relationship:

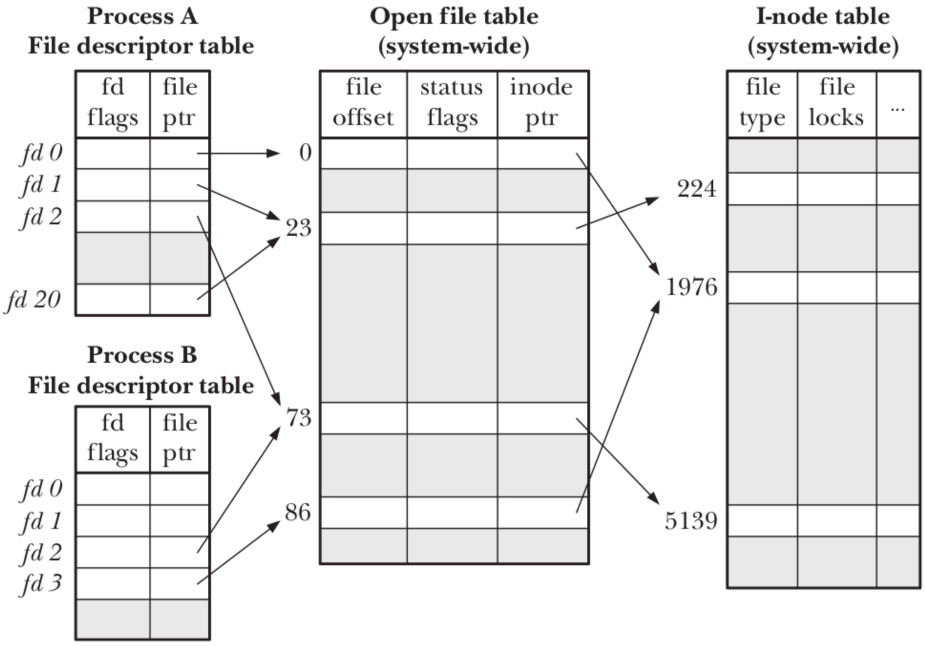


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

File descriptor table

Per-process table with one entry for each FD opened by process:

- Flags controlling operation of FD (close-on-exec flag)
- Reference to open file description
- *struct fdtable* in `include/linux/fdtable.h`

able of open file descriptions (open file table)

System-wide table, one entry for each open file on system:

- File offset
- File access mode (R / W / R-W, from *open()*)
- File status flags (from *open()*)
- Reference to inode object for file
- *struct file* in `include/linux/fs.h`

Following terms are commonly treated as synonyms:

- **open file description (OFD)** (POSIX)
- **open file table entry** or **open file handle**
  - ⚠ Ambiguous terms; POSIX terminology is preferable

(In-memory) inode table

System-wide table drawn from file inode information in filesystem:

- File type (regular file, FIFO, socket, . . .)
- File permissions
- Other file properties (size, timestamps, . . .)
- *struct inode* in `include/linux/fs.h`

Duplicated file descriptors (intraprocess)

A process may have multiple FDs referring to same OFD

- Achieved using *dup()* or *dup2()*

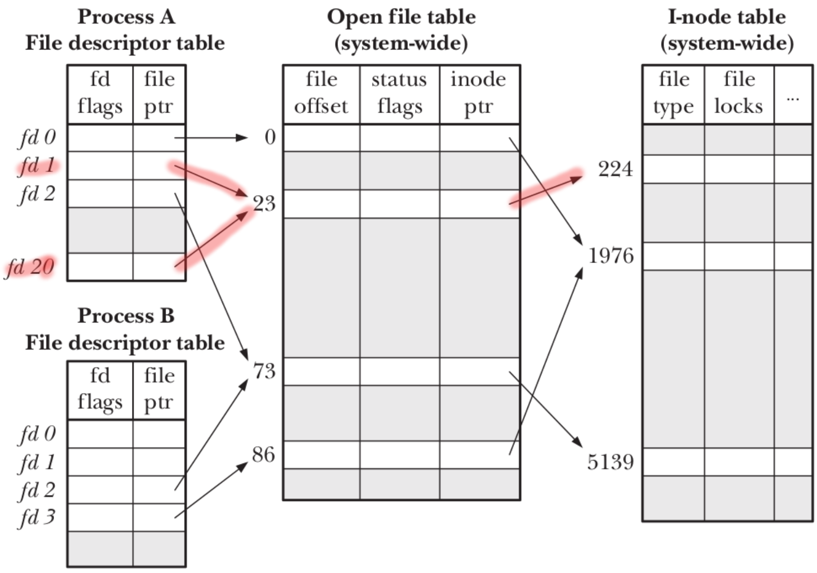


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Duplicated file descriptors (between processes)

Two processes may have FDs referring to same OFD

- Can occur as a result of *fork()*

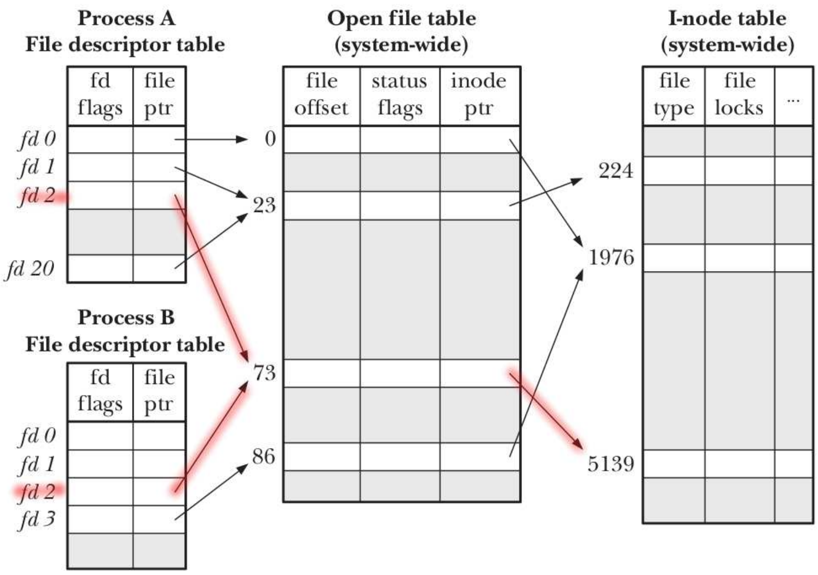


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Distinct open file table entries referring to same file

Two processes may have FDs referring to distinct OFDs that refer to same inode

- Two processes independently *open()*ed same file

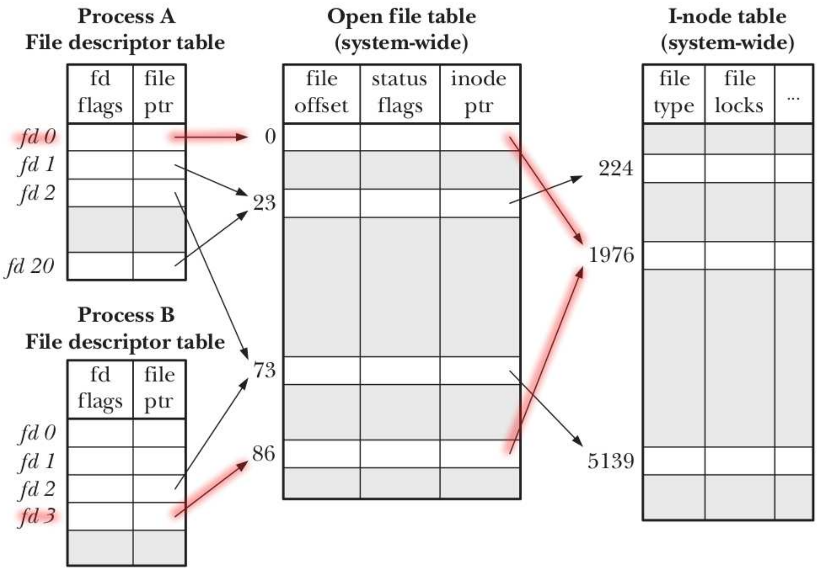


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Why does this matter?

- Two different FDs referring to same OFD share file offset
  - (File offset == location for next *read()*/*write()*)
  - Changes (*read()*, *write()*, *lseek()*) via one FD visible via other FD
  - Applies to both intraprocess & interprocess sharing of OFD
- Similar scope rules for status flags (`O_APPEND`, `O_SYNC`, ...)
  - Changes via one FD are visible via other FD
    - (`fcntl(F_SETFL)` and `fcntl(F_GETFL)`)
- Conversely, changes to FD flags (held in FD table) are private to each process and FD
- *kcmp(2)* `KCMP_FILE` operation can be used to test if two FDs refer to same OFD
  - Linux-specific

[TLPI §5.4]

Outline

A problem

```
./myprog > output.log 2>&1
```

- What does the shell syntax, `2>&1`, do?
- How does the shell do it?
- Open file twice, once on FD 1, and once on FD 2?
  - FDs would have separate OFDs with distinct file offsets ⇒ standard output and error would overwrite
  - File may not even be *open()*-able:
    - e.g., `./myprog 2>&1 | less`
- Need a way to create duplicate FD that refers to same OFD

[TLPI §5.5]

Duplicating file descriptors

```
#include <unistd.h>
int dup(int oldfd);
```

- Arguments:
  - *oldfd*: an existing file descriptor
- Returns new file descriptor (on success)
- **New file descriptor is guaranteed to be lowest available**

Duplicating file descriptors

- FDs 0, 1, and 2 are normally always open, so shell can achieve 2>&1 redirection by:

```
close(STDERR_FILENO);          /* Frees FD 2 */
newfd = dup(STDOUT_FILENO);    /* Reuses FD 2 */
```

- But what if FD 0 had been closed beforehand?
  - We need a better API...

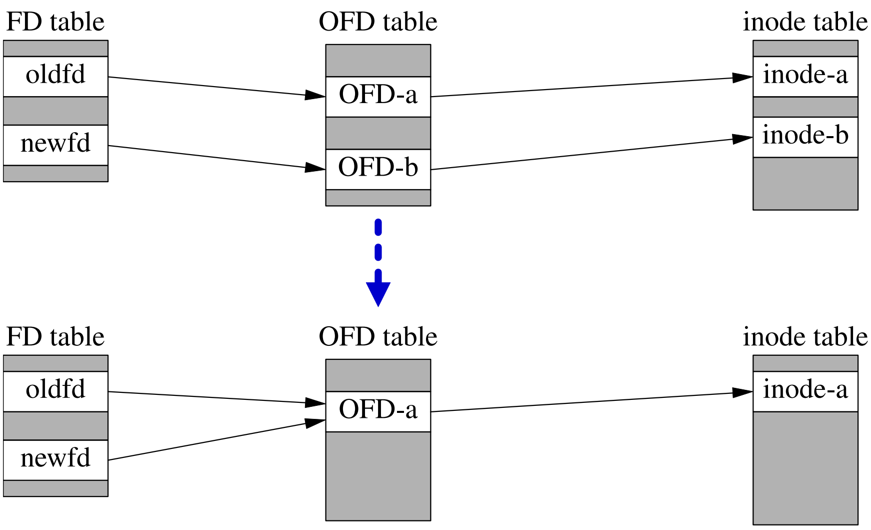
Duplicating file descriptors

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

- Like *dup()*, but uses *newfd* for the duplicate FD
  - **Silently** closes *newfd* if it was open
  - Closing + reusing *newfd* is done atomically
    - Important: otherwise *newfd* might be re-used in between
  - Does nothing if *newfd* == *oldfd*
  - Returns new file descriptor (i.e., *newfd*) on success
- `dup2(STDOUT_FILENO, STDERR_FILENO);`
- See *dup2(2)* man page for more details

[TLPI §5.5]

Understanding *dup2(oldfd, newfd)*



After the *dup2()*:

- If *newfd* was an open FD, OFD-b will be released if *newfd* was the last FD that referred to it
- *oldfd* and *newfd* share file offset and file status flags

Outline

File status flags

- Control semantics of I/O on a file
  - (`O_APPEND`, `O_NONBLOCK`, `O_SYNC`, ...)
- Associated with open file description
- Set when file is opened
- Can be retrieved and modified using *fcntl()*

[TLPI §5.3]

fcntl(): file control operations

```
#include <fcntl.h>
int fcntl(int fd, int cmd /* , arg */ );
```

Performs control operations on an open file

- Arguments:
  - *fd*: file descriptor
  - *cmd*: the desired operation
  - *arg*: optional, type depends on *cmd*
- Return on success depends on *cmd*; –1 returned on error
- Many types of operation
  - file locking, signal-driven I/O, file descriptor flags ...

Retrieving file status flags and access mode

- Retrieving flags (both access mode and status flags)

```
flags = fcntl(fd, F_GETFL);
```

- Check access mode

```
amode = flags & O_ACCMODE;
if (amode == O_RDONLY || amode == O_RDWR)
    printf("File is readable\n");
```

- ⚠ 'read' and 'write' are not separate bits!

```
if (flags & O_RDONLY)       /* Wrong!! */
    printf("File is readable\n");
```

  - Access mode is a 2-bit field that is an enumeration:
    - 00 == `O_RDONLY`
    - 01 == `O_WRONLY`
    - 10 == `O_RDWR`
- Access mode can't be changed after file is opened

Retrieving and modifying file status flags

- Retrieving file status flags

```
flags = fcntl(fd, F_GETFL);
if (flags & O_NONBLOCK)
    printf("Nonblocking I/O is in effect\n");
```

- Setting a file status flag

```
flags = fcntl(fd, F_GETFL);      /* Retrieve flags */
flags |= O_APPEND;               /* Set "append" bit */
fcntl(fd, F_SETFL, flags);       /* Modify flags */
```

  - ⚠ Not thread-safe...
    - (But in many cases, flags can be set when FD is created, e.g., by *open()*)

- Clearing a file status flag

```
flags = fcntl(fd, F_GETFL);      /* Retrieve flags */
flags &= ~O_APPEND;              /* Clear "append" bit */
fcntl(fd, F_SETFL, flags);       /* Modify flags */
```


Exercise

1. Show that duplicate file descriptors share file offset and file status flags by writing a program (**[template: `fileio/ex.fd_sharing.c`]**) that:
   - Opens an existing file (supplied as *argv[1]*) and duplicates (*dup()*) the resulting file descriptor, to create a second file descriptor.
   - Displays the file offset and the state of the `O_APPEND` file status flag via the first file descriptor.
     - Initially the file offset will be zero, and the `O_APPEND` flag will not be set
   - Changes the file offset (*lseek()*, slide 3-23) and enables (turns on) the `O_APPEND` file status flag (*fcntl()*, slide 3-45) via the second file descriptor.
   - Displays the file offset and the state of the `O_APPEND` file status flag via the first file descriptor.

   Hints:
   - Remember to update the `Makefile`!
   - `while inotifywait -q . ; do echo; echo; make; done`

Exercise

2. The program `fileio/fd_overwrite.c` can be used to demonstrate that if a program opens the same file twice, the two file descriptors do not share a file offset, and thus writes via one file descriptor will overwrite writes via the other file descriptor. By contrast, if a program opens the file and duplicates the resulting file descriptor, then the two file descriptors do share a file offset, and writes via one file descriptor will not overwrite writes via the other file descriptor. The program is used with a command-line as follows:

```
$ ./fd_overwrite [-d] <file> <string>...
```

By default, the program *open()*s the specified file twice, but if the *–d* option is specified, then it *open()*s the file once and duplicates the resulting file descriptor. The remaining arguments are strings that are alternately written to the two file descriptors (thus, the first string is written to FD 1, the second to FD 2, the third to FD1, and so on). Run the program with the following two command lines, and explain the output that appears in the two files:

Exercise

```
$ ./fd_overwrite_test    x a A b B c C
$ ./fd_overwrite_test -d y a A b B c C
```

3. Read about the `KCMP_FILE` operation in the *kcmp(2)* man page. Extend the program created in the first exercise to use this operation to verify that the two file descriptors refer to the same open file description (i.e., use *kcmp(getpid(), getpid(), KCMP_FILE, fd1, fd2)*). Note: because there is currently no *kcmp()* wrapper function in glibc, you will have to write one yourself using *syscall(2)*:

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/kcmp.h>

static int kcmp(pid_t pid1, pid_t pid2, int type,
                unsigned long idx1, unsigned long idx2)
{
    return syscall(SYS_kcmp, pid1, pid2, type,
                   idx1, idx2);
}
```

Outline

Retrieving file information: *stat()*

```
#include <sys/stat.h>
int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

- Retrieve information about a file ("metadata"), mostly from inode
  - Information placed in *statbuf*
- *stat()*: retrieve info about **filename** identified by *pathname*
- *lstat()*: if *pathname* is a **symbolic link**, retrieve information about link, not file to which it refers
  - (*stat()* dereferences symbolic links)
- *fstat()*: retrieve info about file referred to by **descriptor** *fd*

[TLPI §15.1]

The *stat* structure

```
struct stat {
  dev_t   st_dev;    /* ID of device containing file */
  ino_t   st_ino;    /* Inode number of file */
  mode_t  st_mode;   /* File type and permissions */
  nlink_t st_nlink;  /* # of (hard) links to file */
  uid_t   st_uid;    /* User ID of file owner */
  gid_t   st_gid;    /* Group ID of file owner */
  dev_t   st_rdev;   /* ID for device special files */
  off_t   st_size;   /* File size (bytes) */
  blksize_t st_blksize; /* Optimal I/O block size (B) */
  blkcnt_t  st_blocks;  /* # of 512B blocks allocated */
  time_t  st_atime;  /* Time of last file access */
  time_t  st_mtime;  /* Time of last file modification */
  time_t  st_ctime;  /* Time of last status change */
};
```

- All types above are defined by POSIX (mostly integers)
- Full details on fields can be found in *inode(7)* and *stat(2)*
  - We'll look at details of a **subset** of these fields

The *stat* structure

- *st_dev*: ID of device containing filesystem where device resides
  - Consists of major ID (12 bits) + minor ID (20 bits)
  - *st_dev* value is calculated by kernel (not stored as part of inode)
- *st_ino*: inode number of file
- *st_nlink*: number of (hard) links to file
- *st_size*: nominal file size (bytes) (*ls –l*)
- *st_blocks*: number of 512-byte blocks actually allocated for file (*du –h*)
  - May be < *(st_size / 512)* because of file "holes"

File timestamps

- File timestamps record time since Epoch (00:00:00, 1 Jan 1970, UTC):
  - *st_atime*: time of last access of file data
  - *st_mtime*: time of last modification of file data
  - *st_ctime*: time of last change to inode
- Various system calls update timestamps as expected
  - TLPI Table 15-2
- Timestamps are really *timespec* structures, recording **seconds and nanoseconds**
  - E.g., *st_atim.tv_sec* and *st_atim.tv_nsec*
- Not all FS types support nanosecond timestamps
  - XFS, ext4, and Btrfs do

[TLPI §15.2]

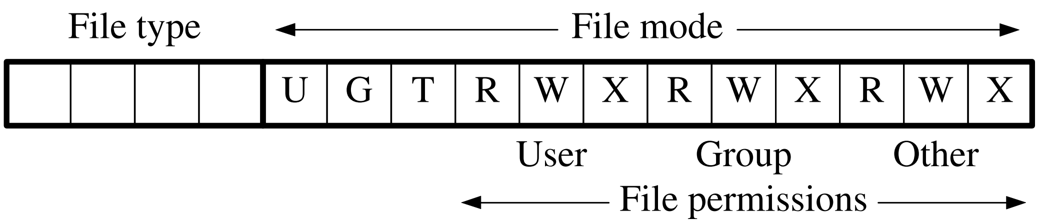
File ownership

- *st_uid* and *st_gid* identify **ownership of file**
  - File UID + GID determine **permissions for file access**
- **UID of new file** == effective UID of creating process
- On most filesystems, **GID of new file** is either:
  - Effective **GID of creating process** (System V semantics)
  - **GID of parent directory** (BSD semantics)
    - Allows creation of subtrees that are always accessible to a particular group
- Choice is determined by whether **parent directory's set-GID bit** is enabled
  - `chmod g+s <dir>` (Propagates directory GID to new files)
  - This use of set-GID bit is a Linux extension

[TLPI §15.3]

File type and mode

- *st_mode* returns two pieces of info:



- Left-most bits give **file type**
- Remaining bits are **file mode**
  - File permissions (9 bits) + set-UID/set-GID/sticky bits

[TLPI §15.1]

File type

- Extract using `statbuf.st_mode & S_IFMT`
- Check using predefined constants and macros:

```
if ((statbuf.st_mode & S_IFMT) == S_IFREG)
    printf("regular file\n");
```

- Common operation, so there are shorthand macros:

```
if (S_ISREG(statbuf.st_mode)) ...
```

Constant	Test macro	File type
`S_IFREG`	`S_ISREG()`	Regular file
`S_IFDIR`	`S_ISDIR()`	Directory
`S_IFCHR`	`S_ISCHR()`	Character device
`S_IFBLK`	`S_ISBLK()`	Block device
`S_IFIFO`	`S_ISFIFO()`	FIFO
`S_IFSOCK`	`S_ISSOCK()`	Socket
`S_IFLNK`	`S_ISLNK()`	Symbolic link

Linux/UNIX System Programming Fundamentals

Directories and Links

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

Outline

Directories

- Stored in same way as a regular file on filesystem
  - But, marked as "directory" in inode
  - (*mkdir()*, *rmdir()*)
- File with a special organization: **table mapping filenames to inode numbers**
  - **Unsorted!** (`ls -U`)

`tmp`	1952
`bin`	6523
`Mail`	224
`init`	1976
`.bashrc`	4594

  - To see inode number: `ls -i <file>`
- **Filenames can be up to 255** bytes on most native Linux filesystems

[TLPI §18.1]

(Hard) Links

- Filename == alias for inode number
- Usual terminology for these aliases is “**links**”
  - Or: “**hard links**” to distinguish from soft/symbolic links
- Multiple filenames can alias to same inode number
  - In same directory or in different directories
- Creating hard link in shell: `ln <old-name> <new-name>`

```
$ mkdir dir1 dir2
$ echo "Hello" > dir1/x
$ ln dir1/x dir2/y
$ echo "World" >> dir2/y
$ cat dir1/x
Hello
World
$ ls -li dir1/x dir2/y
4064456 -rw-r--r--. 2 mtk mtk 12 Nov 14 11:22 dir1/x
4064456 -rw-r--r--. 2 mtk mtk 12 Nov 14 11:22 dir2/y
```

[TLPI §18.1]

(Hard) Links

- All links to a file have equal status
- Each inode has a link count
- File blocks are deallocated only when link count reaches zero
- ⇒ `rm <file>` means:
  - Remove this link to an inode
  - Decrement link counter in inode
  - If link count in inode is now 0, deallocate data blocks and recycle inode slot

Restrictions on creating hard links

- Can't link to a file on another filesystem
  - Inode numbers are unique only within a filesystem
- Can't link to a directory
  - Prevents creation of loops in directory hierarchy
    - Tools that traverse trees could detect such loops, but would need to track/test against inode numbers of visited directory (expensive)
    - Garbage collection would be required for orphaned directories
  - If a directory has multiple links, what should ". ." mean?
    - If several parent directories have links to same child directory, what is ". ." in that directory?
    - What should happen to ". ." if "original" parent is deleted?
- Symbolic links provide a way round these limitations

Outline

Symbolic links

- Symbolic link (AKA "symlink" or "soft link"):
  - File with specially marked inode
  - Content is name of another file (the "target")
- Create from shell: `ln -s target link-name`

[TLPI §18.2]

Interpretation of symbolic links

- System calls **dereference ("resolve", "follow") symbolic links** when interpreting pathnames
  - I.e., symbolic link is replaced with "target"
- Target of link can be a **relative** pathname
  - Interpreted relative to directory containing link
- Target of symlink can be another symlink
  - Kernel will recursively resolve
  - Limit of 40 dereferences in a pathname
    - $>$ 40 dereferences $\Rightarrow$ `ELOOP` error

Interpretation of symbolic links

- All system calls **always** dereference symlinks **in prefix part** ("dirname") of a pathname
  - `/prefix/part/of/path/basename`
- Most system calls also dereference symlinks in **final component** ("basename") of a pathname
- Some system calls don't dereference symlinks in "basename"
  - Certain system calls (by design) work on symlinks rather than their targets; e.g.:
    - "l" syscalls (*lchown()*, *lstat()*, etc.)
    - *unlink()*, *rename()*, ...
- Some system calls fail (by design) if given a symbolic link
  - e.g., *rmdir()*, *open(O_NOFOLLOW)*
  - Prevents "symlink attacks"

Symbolic links

- Hard and soft links are different kinds of aliases:
  - Hard links are **aliases for inode numbers**
  - Symbolic links are **aliases for pathnames**
- Symbolic links are **not** reflected in link count of target file
- If target is deleted (or never existed), symbolic link is **dangling**
  - Attempts to resolve yield `ENOENT` error
- Unlike hard links, symbolic links:
  - Can link across filesystems
  - Can link to directories
    - Programs that scan directory trees know to avoid symbolic link loops
- But there are still use cases for hard links...

So then, why use hard links?

- Symlinks add layer of indirection (extra accesses in FS)
- Hard link pins file into existence; a symlink does not:

```
$ ln -s file slink
$ ln file hlink
$ rm file           # Renaming file would also be a problem...
$ cat slink
cat: slink: No such file or directory
```

  - But `hlink` still refers to original file...
- Hard links are needed to implement ". ."
- Hard links remain valid inside *chroot* environment
- And there are other use cases

Outline

Creating a (hard) link: *link()*

```
include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

- Creates new (hard) link, *newpath*, to an existing file, *oldpath*
- *newpath* must not exist before call (EEXIST)

[TLPI §18.3]

Removing a link: *unlink()*

```
#include <unistd.h>
int unlink(const char *pathname);
```

- Removes the link *pathname*
  - Can't *unlink()* a directory (use *rmdir()* or *remove()*)
- Subtracts 1 from link count in file's inode
- If link count is now 0, file is deleted
- If *pathname* is a symlink, the link itself is removed
  - (**Not** the target of the symlink)

[TLPI §18.3]

unlink() and open files

- The kernel counts open file descriptions (OFDs) referring to a file
- A file's contents are deleted only when
  - link count is 0 **and**
  - all OFDs are closed
- Uses:
  - Can *unlink()* a file without worrying if open in another process
  - Can open a temporary file, and immediately unlink its name
    - Filename disappears immediately
    - File content disappears when file is closed

Removing a file or directory: *remove()*

```
#include <stdio.h>
int remove(const char *pathname);
```

- Removes a file or a directory
  - Calls *unlink()* on files
  - Calls *rmdir()* on directories

[TLPI §18.7]

Renaming a file: *rename()*

```
#include <stdio.h>
int rename(const char *oldpath, const char *newpath);
```

- Renames the file *oldpath* to *newpath*
- *rename()* simply manipulates entries in directories:
  - ⇒ *oldpath* and *newpath* **must be on same filesystem**
  - How does *mv(1)* move files between filesystems?
    - Write a copy of file, and delete original
- More details (and rules) in TLPI §18.4 and *rename(2)*

[TLPI §18.4]

Outline

Creating a symbolic link: *symlink()*

```
#include <unistd.h>
int symlink(const char *target, const char *linkpath);
```

- Creates a new symbolic link, *linkpath*, with content *target*
  - (A symlink can be **removed** with *unlink()*)
- *target* can be up to `PATH_MAX` bytes (including terminating `NULL` byte)
- *target* need not exist at time of call ⇒ **dangling link**

[TLPI §18.5]

Inspecting a symbolic link: *readlink()*

```
#include <unistd.h>
ssize_t readlink(const char *pathname, char *buffer,
                 size_t bufsiz);
```

- Retrieves content (i.e., target) of symlink in (final component of) *pathname*
- Content is placed in *buffer*
  - ⚠ No null terminator added
- *bufsiz* specifies number of bytes available in *buffer*
- Returns number of bytes placed in *buffer*, or –1 on error
- ⚠ **If *bufsiz* is too small**, value placed in *buffer* is **silently truncated**
  - Make sure *bufsiz* is **bigger** than needed
  - Check that return value $<$ *bufsiz*

[TLPI §18.5]

Outline

The current working directory

- Each process has a current working directory (CWD)
- Location from which relative pathnames are interpreted
  - (i.e., pathnames that do not start with "/")

[TLPI §18.10]

Retrieving the current working directory: *getcwd()*

```
#include <unistd.h>
char *getcwd(char *cwdbuf, size_t size);
```

- Places null-terminated absolute pathname of CWD in *cwdbuf*
- *size* specifies number of bytes available in *cwdbuf*
- Returns *cwdbuf* on success, or `NULL` on error
  - `ERANGE` error means *size* was not big enough
- **Glibc extension**: if *cwdbuf* is `NULL` and *size* is 0, *getcwd()* allocates buffer that is large enough and returns pointer to it
  - Caller must *free()* buffer

[TLPI §18.10]

Changing the current working directory

```
#include <unistd.h>
int chdir(const char *pathname);
int fchdir(int fd);
```

- *chdir()* changes CWD to *pathname*
- *fchdir()* changes CWD to directory referred to by **file descriptor** *fd*
  - Obtain *fd* by *open()*-ing a directory for reading

```
int fd;
fd = open(".", O_RDONLY);     /* Remember where we are */
chdir("/tmp");                     /* Go somewhere else */
...                   /* Do something in that directory */
fchdir(fd);                  /* Return to previous location */
close(fd);                             /* No longer needed */
```

[TLPI §18.10]

Outline

openat()

```
#include <fcntl.h>
int openat(int dirfd, const char *pathname, int oflag,
           ...);
```

- Similar to *open()*, but has extra argument *dirfd*
  - File descriptor that refers to a directory
- Example of one of several APIs that support following cases:
  - *pathname* is absolute ⇒ *dirfd* is ignored; behavior exactly like *open()*
  - *pathname* is relative, *dirfd* is `AT_FDCWD` ⇒ *pathname* is interpreted in usual fashion (i.e., like *open()*)
  - *pathname* is relative, *dirfd* refers to directory ⇒ ***pathname* is interpreted relative to *dirfd*** (instead of CWD)

The **at()* functions

- Many similar APIs added to Linux in 2.6.16; others added later
  - *execveat(2)*, *faccessat(2)*, *fanotify_mark(2)*, *fchmodat(2)*, *fchownat(2)*, *fstatat(2)*, *futimesat(2)*, *linkat(2)*, *mkdirat(2)*, *mknodat(2)*, *name_to_handle_at(2)*, *openat2(2)*, *readlinkat(2)*, *renameat(2)*, *statx(2)*, *symlinkat(2)*, *unlinkat(2)*, *utimensat(2)*, *mkfifoat(3)*, *scandirat(3)*
- Some standardized in POSIX.1-2008

Rationale for the **at()* functions

- Address problems in many traditional APIs
- First: useful in multithreaded applications
  - "Current working directory" is a process-global attribute
  - **at()* functions allow threads to maintain **per-thread working directory**

Rationale for the **at()* functions

- Example usage of "thread current directory"

```
/* Obtain file descriptor that refers to a directory */

dirfd = open("/path/to/dir", O_RDONLY);

/* Perform operations on relative pathnames */

fstatat(dirfd, "somefile", &statbuf);
fd = openat(dirfd, "anotherfile", O_CREAT|O_RDWR, mode);

/* Change thread "current directory" to a subdirectory
   under 'dirfd' */

newdirfd = openat(dirfd, "subdir", O_RDONLY);
if (newdirfd != -1) {
    close(dirfd);
    dirfd = newdirfd;
}
```


Rationale for the **at()* functions

- Second: avoid race conditions that can occur when operating files in location other than CWD
- Problem: a symlink in dirname of pathname changes as we perform operations related to pathname; example:
  1. Check (*stat()*) attributes of `/dir1/dir2/file`
  2. Target of `dir1` or `dir2` symlink changes
  3. Create (*open()*) `/dir1/dir2/file.dep`
- Solution: open an FD referring to target directory and employ **at()* calls

Rationale for the **at()* functions

```
dirfd = open("/dir1/dir2", O_RDONLY);

fstatat(dirfd, "file", &statbuf);
/* Perform a check using returned stat buffer */

fd = openat(dirfd, "file.dep", O_CREAT...);
```

- *open()* is being used only to obtain a reference to directory
  - We can't *read()* from *dirfd*
  - See also: discussion of `O_PATH` flag in *open(2)*

Rationale for the **at()* functions

```
dirfd = open("/dir1/dir2", O_RDONLY);

fstatat(dirfd, "file", &statbuf);
/* Perform a check using returned stat buffer */

fd = openat(dirfd, "file.dep", O_CREAT...);
```

- *dirfd* remains a stable reference to directory, regardless of subsequent changes to symlinks in `/dir1/dir2`
- *dirfd* has other useful properties:
  - *dirfd* is stable even if original directory is renamed
    - (If directory is *deleted*, attempts to create files give `ENOENT`)
  - Open *dirfd* prevents filesystem being unmounted
    - Like traditional CWD
  - (Solutions based on initially resolving symlinks in pathname by use of *realpath(3)* would not have these properties)

Exercises

1. The goal of this exercise is to show one of the reasons that the **at()* functions (in this case, *openat()*) can be useful: to obtain a reference to a directory that remains stable even if symlink components in the directory pathname are modified.

   Write a program (**[template:** **`dirs_links/ex.openat_expmt.c`]**) that takes one argument, which is a pathname. The final component of the prefix (dirname) of this pathname is expected to be a symbolic link that refers to a directory. The suffix component (basename) of the pathname is a filename inside that directory. (To split a pathname into dirname and basename components, use *dirname(3)* and *basename(3)*.)

   The program should perform the following steps:

   - Open a (read-only) file descriptor referring to the dirname component of the argument.
   - Fetch (*readlink()*, slide 4-22) the target of the symbolic link referred to by the dirname component and print it. (Remember: *readlink()* does **not** null-terminate the returned buffer.)
   - Sleep for 15 seconds
   - Once more fetch and display the target of the symbolic link
   - Use *open()* to open the file, using the full pathname specified on the command line. Read and display the contents of the file.
     [Exercise continues on the next slide]

Exercises

- Use *openat()* (slide 4-28) to open the file named in the command-line argument, using the directory file descriptor obtained in the first step plus the basename component of the argument. Read (*read()*) and display the contents of the file. (Remember: *read()* does **not** null-terminate the returned buffer.)
- Set up a test environment for the program as follows:

```
$ mkdir xxx
$ echo "hello" > xxx/f
$ mkdir yyy
$ echo "world" > yyy/f
$ ln -s xxx testdir
```

- Run the program, specifying `testdir/f` as the argument and, while the program is sleeping, execute the following command:

```
$ rm testdir; ln -s yyy testdir
```

- Explain the program output, which should be:

```
world
hello
```

Exercises

2. (Kernel hacking exercise) Although "at" versions of many historical UNIX APIs have been implemented on Linux, there are still a few APIs that do not yet have "at" equivalents. Notably, the *bindat()* and *connectat()* APIs are not implemented on Linux (or specified in POSIX). These APIs work with UNIX domain sockets, which employ pathnames to identify sockets.

   These APIs *are* implemented on FreeBSD. Read the FreeBSD man pages for these APIS, and implement the equivalent system calls on Linux. Obviously, it will be helpful to also look at the Linux kernel source code that implements the existing "at" system calls, and read their manual pages. (From time to time, the topic of implementing these system calls has been raised on the Linux Kernel Mailing List, and it would be worth hunting down those threads to CC interested people on any patches.)

Outline

Scanning directories

Task: scan the contents of a directory list, or traverse all files in a directory subtree

- A single directory can be opened and scanned using *opendir(3)* and *readdir(3)*
  - Yields a list of filenames and inode numbers
  - See TLPI §18.8 for details
- An entire directory tree can be traversed using *nftw(3)*
  - *nftw()* is implemented using *opendir()* and *readdir()*
  - See TLPI §18.8 for details

Notes

Linux/UNIX System Programming Fundamentals

Processes

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

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
    - Kernel 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]

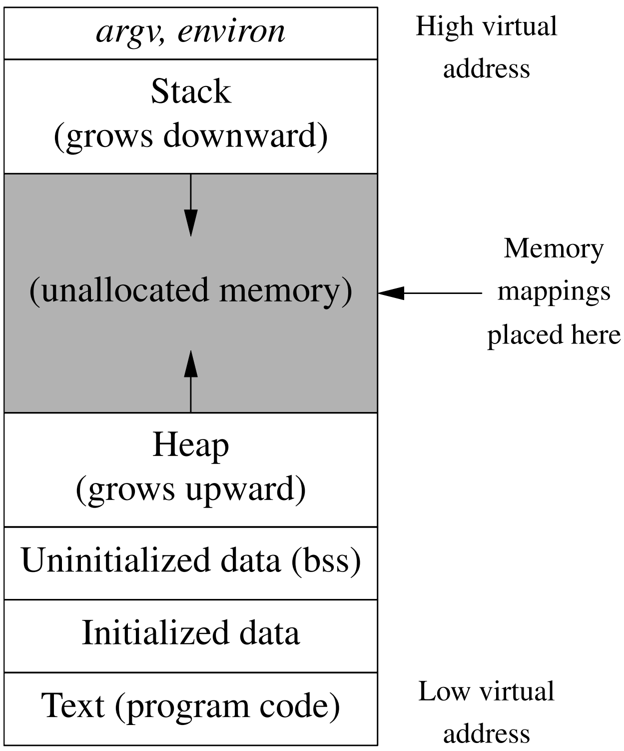
Outline

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)

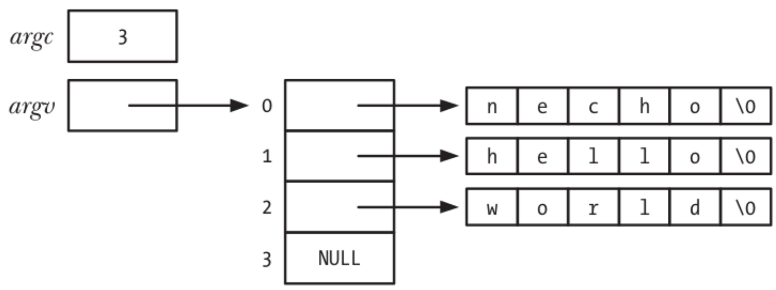


[TLPI §6.3]

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 (*environron*)

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
$ export MANWIDTH
$ man getpid
```

- 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
```

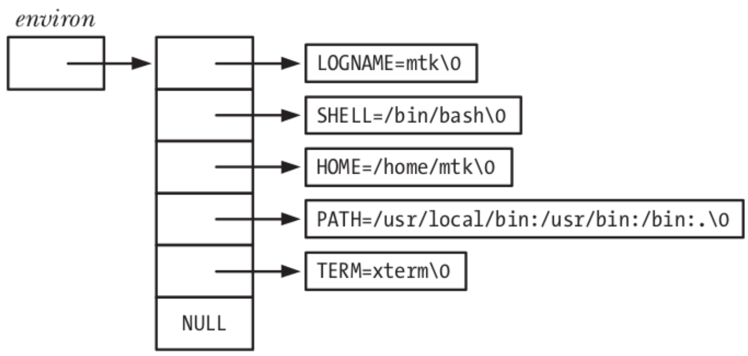
- To list all environment variables, use *env(1)* or *printenv(1)*

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/environment` can be used (with suitable permissions) to view environment of another process
- See man 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)* man 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/UNIX System Programming Fundamentals

Signals: Introduction

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

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 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**: 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)* man 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*
  - But, a signal handler can emulate these behaviors
- Can't change disposition of `SIGKILL` or `SIGSTOP` (`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*
  - Can be `NULL` for no change
- *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
- Other fields only significant when establishing a signal handler, but must be properly initialized here

Outline

Signal handlers

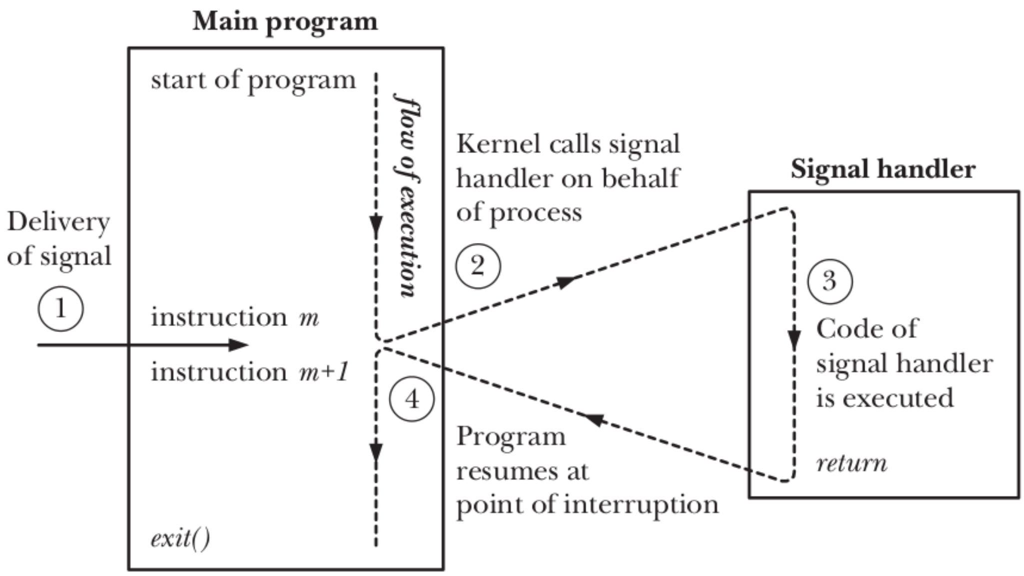
- Programmer-defined function
- Called with one integer argument: number of signal
    - ⇒ handler installed for multiple signals can differentiate...
- Returns `void`

```
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 */
    /* Don't block additional signals
       during invocation of handler */
    sigemptyset(&sa.sa_mask);

    if (sigaction(SIGINT, &sa, NULL) == -1)
        errExit("sigaction");

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


Exercise

- While a signal 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

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)

[TLPI §20.14]

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
  - `SIG_UNBLOCK`: **remove** signals in *set* from signal mask
  - `SIG_SETMASK`: **assign** *set* to 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);
```


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
- By contrast, realtime signals *do* queue

Exercises

The goal of this exercise is experiment with signal handlers and the use of the signal mask to block delivery of signals. A template for the complete exercise is provided (**[template: `signals/ex.pending_sig_expt.c`]**)

Hint: don't confuse the *sa_mask* field that is passed to *sigaction()*, which specifies additional signals that should be temporarily blocked while a signal handler is executing, with the use of *sigprocmask()*, which allows a process to directly modify its signal mask.

1. Write a program that:
    - Blocks all signals except `SIGINT` (*sigprocmask()*, slides 6-30 + 6-31).
    - Uses *sigaction()* (slides 6-13 + 6-14) to establish a `SIGINT` handler that does nothing but return.
    - Calls *pause()* to wait for a signal.

[Exercise continues on following slides]

Exercises

- After *pause()* returns, determines the set of pending signals for the process (use *sigpending()*, slide 6-32), tests which signals are in that set (use *sigismember()*, iterating through all signals in the range `1 <= s < NSIG`), and prints their descriptions (*strsignal()*).

Run the program and send it various signals (other than `SIGINT` and signals that are ignored by default) using the *kill* command (`kill -<sig> <pid>`). Then type Control-C to generate `SIGINT` and inspect the list of pending signals.

2. What happens if you send `SIGKILL` to the preceding program? Why?

3. Extend the program created in the preceding exercise so that:
   - Just after installing the handler for `SIGINT`, the program installs an additional handler for `SIGQUIT` (generated when the Control-\ key is pressed). The handler should print a message "SIGQUIT received", and return.

Exercises

- 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?

Linux/UNIX System Programming Fundamentals

Signals: Signal Handlers

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

Outline

Keep it simple

- 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
- ⇒ Avoid using signals if you can
  - ⚠ Don't introduce them as a means of IPC
  - ⚠ Don't use as part of a library design
    - (That would imply a contract with main program about which signals library is allowed to use)
- But, in some cases, we must deal with signals sent by kernel
  - ⇒ Design the handlers to be as simple as possible

Keep it simple

- Some simple signal-handler designs:
  - Set a global flag and return
    - Main program periodically checks (and clears) flag, and takes appropriate action
  - 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

Outline

Reentrancy

- Signal handler can interrupt a program at *any* moment
  - ⇒ handler and main program are *semantically* equivalent to two *simultaneous* flows of execution inside process
    - (Like two "threads", but not the same as POSIX threads)
- A function is **reentrant** if it can safely be simultaneously executed by multiple threads
  - Safe == function achieves same result regardless of state of other threads of execution

[TLPI §21.1.2]

Nonreentrant functions

Functions that update global/static variables are **not** reentrant:

- Some functions by their nature operate on global data
  - e.g., *malloc()* and *free()* maintain a global linked list of free memory blocks
    - Suppose main program is executing *free()* and is interrupted by a signal handler that also calls *free()*...
    - Two "threads" updating linked list at same time ⇒ chaos!
- Functions that return results in statically allocated memory are nonreentrant
  - e.g., *getpwnam()* and many other functions in C library
- Functions that use static data structures for internal bookkeeping are nonreentrant
  - e.g., *stdio* functions do this for buffered I/O

Nonreentrant functions

- C library is rife with nonreentrant functions!
  - Man pages usually note functions that are nonreentrant

Async-signal-safe functions

- An async-signal-safe function is one that can be safely called from a signal handler
- A function can be async-signal-safe because either
  - It is reentrant
  - It is not interruptible by a signal handler
    - (Atomic with respect to signals)
- POSIX specifies a set of functions required to be async-signal-safe
  - See *signal-safety(7)* or TLPI Table 21-1
  - Set is a *small* minority of functions specified in POSIX
- No guarantees about functions not on the list
  - ⚠ *stdio* functions are **not** on the list

[TLPI §21.1.2]

Signal handlers and async-signal-safety

- Executing a function inside a signal handler is unsafe only if handler interrupted execution of an unsafe function
- ⇒ Two choices:
  1. Ensure that signal handler calls only async-signal-safe functions
  2. Main program blocks signals when calling unsafe functions or working with global data also used by handler
- Second choice can be difficult to implement in complex programs
  - ⇒ Simplify rule: call only async-signal-safe functions inside a signal handler

Signal handlers can themselves be nonreentrant

- ⚠ Signal handler can also be nonreentrant if it updates global data used by main program
- A common case: handler calls functions that update *errno*
- Solution:

```
void
handler(int sig)
{
    int savedErrno;
    savedErrno = errno;

    /* Execute functions that might
       modify errno */

    errno = savedErrno;
}
```


The *sig_atomic_t* data type

- Contradiction:
  - Good design: handler sets global flag checked by *main()*
  - Sharing global variables between handler & *main()* is unsafe
    - Because accesses may not be atomic

[TLPI §21.1.3]

The *sig_atomic_t* data type

- POSIX defines an integer data type that can be safely shared between handler and *main()*:
  - *sig_atomic_t*
  - Range: `SIG_ATOMIC_MIN`..`SIG_ATOMIC_MAX` (`<stdint.h>`)
  - Read and write guaranteed atomic
  - ⚠ Other operations (e.g., ++ and `--`) **not** guaranteed atomic (i.e., not safe)
  - Specify `volatile` qualifier to prevent optimizer tricks

```
volatile sig_atomic_t flag;
```


Exercises

1. Examine the source code of the program `signals/unsafe_printf.c`, which can be used to demonstrate that calling *printf()* both from the main program and from a signal handler is unsafe. The program performs the following steps:
   - Establishes a handler for the `SIGINT` signal (the control-C signal). The handler uses *printf()* to print out the string "`sssss\n`".
   - After the main program has established the signal handler, it pauses until control-C is pressed for the first time, and then loops forever using *printf()* to print out the string "`mmmmm\n`"

   Before running the program, start up two shells in separate terminal windows as follows (the *ls* command will display an error until the `out.txt` file is actually created):

```
$ watch ps -C unsafe_printf
```

```
$ cd signals
$ watch ls -l out.txt
```

Exercises

In another terminal window, run the *unsafe_printf* program as follows, and then hold down the control-C key **continuously**:

```
$ cd signals
$ ./unsafe_printf > out.txt
^C^C^C
```

Observe the results from the *watch* commands in the other two terminal windows. After some time, it is likely that you will see that the file stops growing in size, and that the program ceases consuming CPU time because of a deadlock in the *stdio* library. Even if this does not happen, after holding the control-C key down for 15 seconds, kill the program using control-\.

Inside the `out.txt` file, there should in theory be only lines that contain "`mmmmm\n`" or "`sssss\n`". However, because of unsafe executions of *printf()*, it is likely that there will be lines containing other strings. Verify this using the following command:

```
$ egrep -n -v '^(mmmmm|sssss)$' < out.txt
```


Exercises

2. Examine the source code of `signals/unsafe_malloc.c`, which can be used to demonstrate that calling *malloc()* and *free()* from both the main program and a signal handler is unsafe. Within this program, a handler for `SIGINT` allocates multiple blocks of memory using *malloc()* and then frees them using *free()*. Similarly, the main program contains a loop that allocates multiple blocks of memory and then frees them.

   In one terminal window, run the following command:

```
$ watch -n 1 ps -C unsafe_malloc
```

   In another terminal window, run the *unsafe_malloc* program, and then hold down the control-C key until either:

   - you see the program crash with a corruption diagnostic from *malloc()* or *free()*; or
   - the *ps* command shows that the amount of CPU time consumed by the process has ceased to increase, indicating that the program has deadlocked inside a call to *malloc()* or *free()*.

Outline

Interrupted system calls

- What if a signal handler interrupts a blocked system call?
- Example:
  - Install handler for (say) `SIGALRM`
  - Perform a *read()* on terminal that blocks, waiting for input
  - `SIGALRM` is delivered
  - What happens when handler returns?
- *read()* fails with `EINTR` ("interrupted system call")
- Can deal with this by manually restarting call:

```
while ((cnt = read(fd, buf, BUF_SIZE)) == -1
            && errno == EINTR)
    continue;    /* Do nothing loop body */
if (cnt == -1)   /* Error other than EINTR */
    errExit("read");
```

[TLPI §21.5]

Automatically restarting system calls: `SA_RESTART`

- Specifying `SA_RESTART` in *sa_flags* when installing a handler causes system calls to *automatically* restart
  - `SA_RESTART` is a per-signal flag
- More convenient than manually restarting, but...
  - Not all system calls automatically restart
  - Set of system calls that restart varies across UNIX systems
  - (Origin of variation is historical)

Automatically restarting system calls: `SA_RESTART`

- Most (all?) modern systems restart at least:
  - *wait()*, *waitpid()*
  - I/O system calls on "slow devices"
    - i.e., devices where I/O can block (pipes, sockets, ...)
    - *read()*, *readv()*, *write()*, *writev()*
- On Linux:
  - Certain other system calls also automatically restart
  - Remaining system calls never restart, regardless of `SA_RESTART`
  - See TLPI §21.5 and *signal(7)* for details
- **Bottom line**: If you need cross-system portability, omit `SA_RESTART` and always manually restart

Outline

Receiving extra signal information: SA_SIGINFO

- Specifying SA_SIGINFO in *sa_flags* argument of *sigaction()* causes signal handler to be invoked with extra arguments
- Handler declared as:

```
void handler(int sig, siginfo_t *siginfo,
             void *ucontext);
```

  - *sig* is the signal number
  - *siginfo* points to structure returning extra info about signal
  - *ucontext* is rarely used (no portable uses)
    - See *getcontext(3)* and *swapcontext(3)*

[TLPI §21.4]

Receiving extra signal information: SA_SIGINFO

- Handler address is passed via *act.sa_sigaction* field (not the usual *act.sa_handler*)

```
struct sigaction act;

sigemptyset(&act.sa_mask);
act.sa_sigaction = handler;
act.sa_flags = SA_SIGINFO;
sigaction(SIGINT, &act, NULL);
```


The *siginfo_t* data type

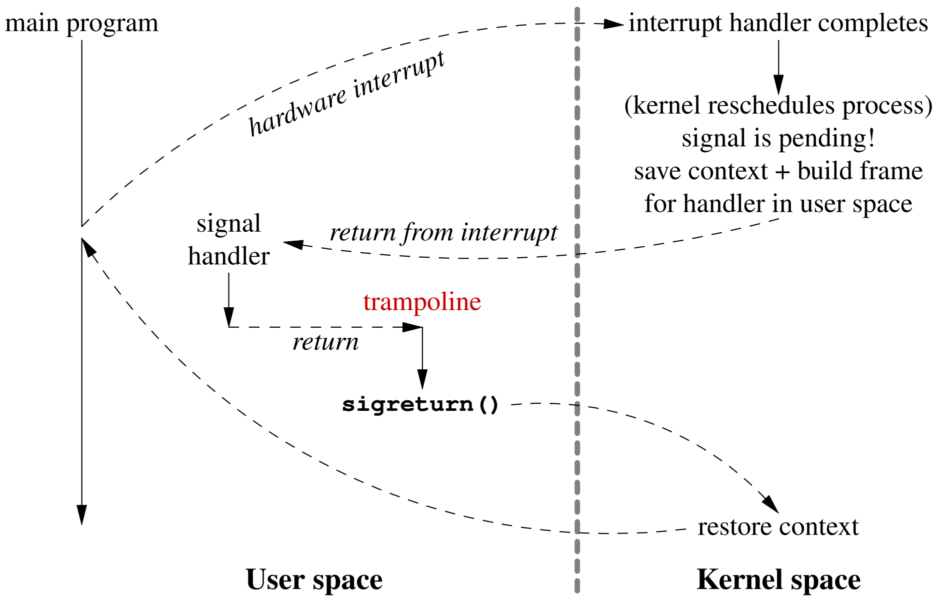
- *siginfo_t* is a structure containing additional info about delivered signal; fields include:
  - *si_signo*: signal number (same as first arg. to handler)
  - *si_code*: additional info about cause of signal
  - *si_pid*: PID of process sending signal (if sent by a process)
  - *si_uid*: real UID of sending process (if sent by a process)
  - *si_value*: data accompanying realtime signal sent with *sigqueue()*
  - And other signal-type-specific fields, such as:
    - *si_addr*: memory location that caused fault; filled in for hardware-generated signals (SIGSEGV, SIGFPE, etc.)
    - *si_fd*: FD that generated a signal (signal-driven I/O)
- See *sigaction(2)* and TLPI §21.4 for more information

Outline

The problem

- Before executing signal handler, kernel must modify some *kernel-maintained* process context
  - Signal mask, signal stack (*sigaltstack()*)
  - (Registers will also be modified during handler execution, and so must be saved)
  - *Easy, because kernel has control at this point*
- Upon return from signal handler, previous context must be restored
  - *But, at this point we are in user mode; kernel has no control*
- **How does kernel regain control in order to restore context?**
  - ⇒ the "signal trampoline"

The "signal trampoline"



The kernel uses the signal trampoline to arrange that control is bounced back to kernel after execution of signal handler

When is a signal delivered?

- In a moment, we consider what's required to execute a signal handler
- But first of all, when is a signal delivered?
  - Signals are asynchronously delivered to process, but...
  - Only on transitions from kernel space back to user space

Steps in the execution of a signal handler

The following steps occur in the execution of a signal handler:

- A hardware interrupt occurs
  - E.g., scheduler timer interrupt, or syscall trap instruction
  - Process is scheduled off CPU
  - Kernel gains control & receives various process context info, which it saves
    - E.g., register values (program counter, stack pointer, etc.)
- Upon completion of interrupt handling, kernel chooses a process to schedule, and discovers it has a pending signal

Steps in the execution of a signal handler

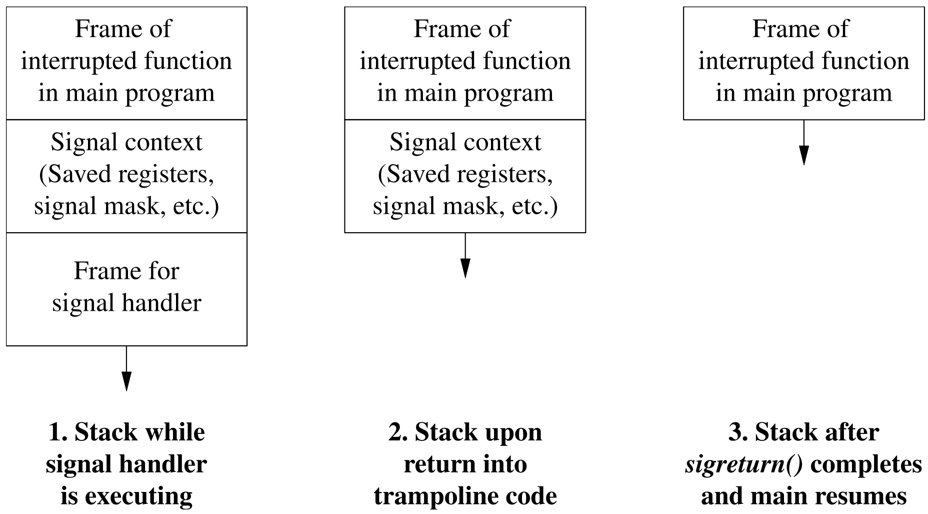
- To allow signal to be handled, the kernel:
  - Saves process context information onto user-space stack
    - Context == CPU registers (PC, SP), signal mask, and more
    - Saved context will be used later by *sigreturn()*...
    - See, e.g., *struct rt_sigframe* definition in `arch/x86/include/asm/sigframe.h`
    - Saved context information is visible via third argument of `SA_SIGINFO` handler, which is really *ucontext_t* *; see also *ucontext_t* definition in `<sys/ucontext.h>`
  - Constructs frame on user-space stack for signal handler
    - Sets return address in frame to point to "signal trampoline"
  - Rearranges trap return address so that upon return to user space, control passes to signal handler
- Control returns to user space
  - Handler is called; handler returns to trampoline

Steps in the execution of a signal handler

- Trampoline code calls *sigreturn(2)*
  - **Now, the kernel once more has control!**
  - *sigreturn()* restores signal context
    - Signal mask, alternate signal stack
  - *sigreturn()* restores saved registers
    - Including program counter ⇒ next return to user space will resume execution where handler interrupted main program
  - Info needed by *sigreturn()* to do its work was saved earlier on user-space stack
    - For example, see code of, and calls to, *setup_sigcontext()* and *restore_sigcontext()* in kernel source file `arch/x86/kernel/signal.c`
  - Trampoline code is in user space (in C library or *vdso(7)*)
    - If in C library, address is made available to kernel via *sa_restorer* field (done by *sigaction()* wrapper function)

sigreturn()

- *sigreturn()*:
  - Special system call used only by signal trampoline
  - Uses saved context to restore state and resume program execution at point where it was interrupted by handler



Notes

Notes

Linux/UNIX System Programming Fundamentals

Process Lifecycle

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

Outline

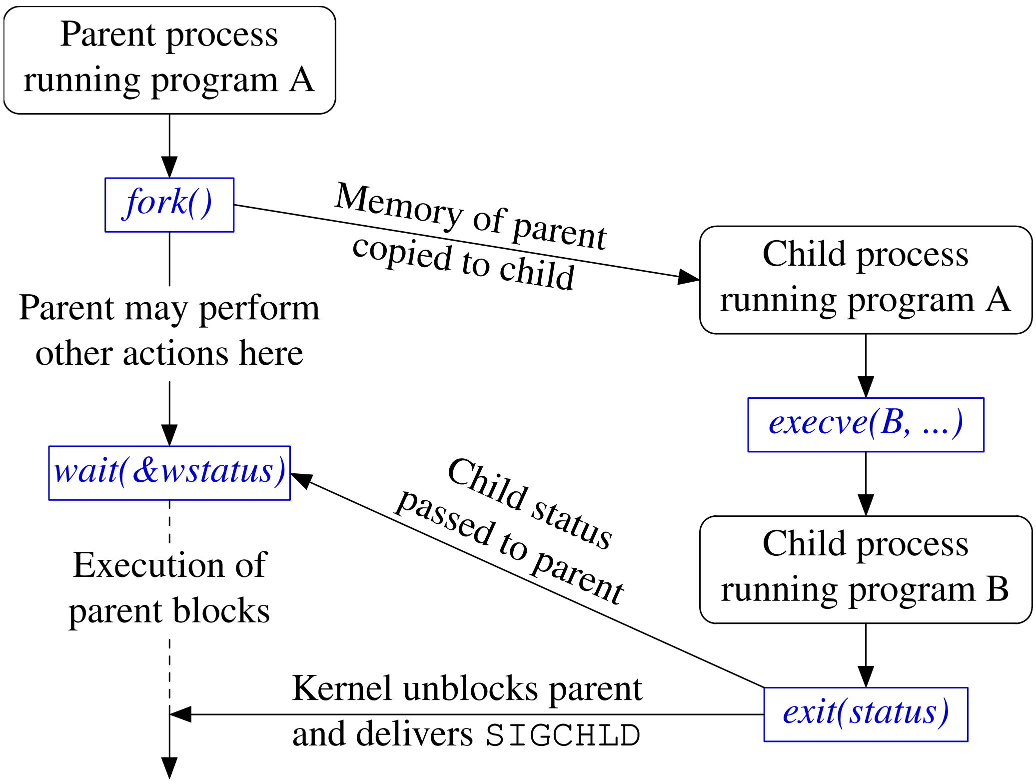
Creating processes and executing programs

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()*

```
pid_t pid;

pid = fork();

if (pid == -1) {

    /* Handle error */ ;

} else if (pid == 0) {

    /* Code executed by child */

} else {

    /* Code executed by parent */

}
```


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.

Exercise

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 dispostions and alarm timers–are inherited by a child process.

   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 `SIGALARM` 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
    - i.e., *_exit()* does **not** flush *stdio* buffers
  - Calls: `_exit(status)`
- `return n` inside *main()* is equivalent to `exit(n)`

Process teardown

As part of process termination (normal or abnormal), various cleanups are performed:

- 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
- State change is reported in *wstatus* (if non-`NULL`)
  - (details later...)
- Return value:
  - On success: PID of child whose status is being reported
  - On error, −1
    - No more children? ⇒ `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*
- *pid* == *0*: any child in **same process group** as caller
- *pid* < *–1*: any child in **process group whose ID equals *abs(pid)***
  - See *credentials(7)* and *setpgid(2)* for info on process groups

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:

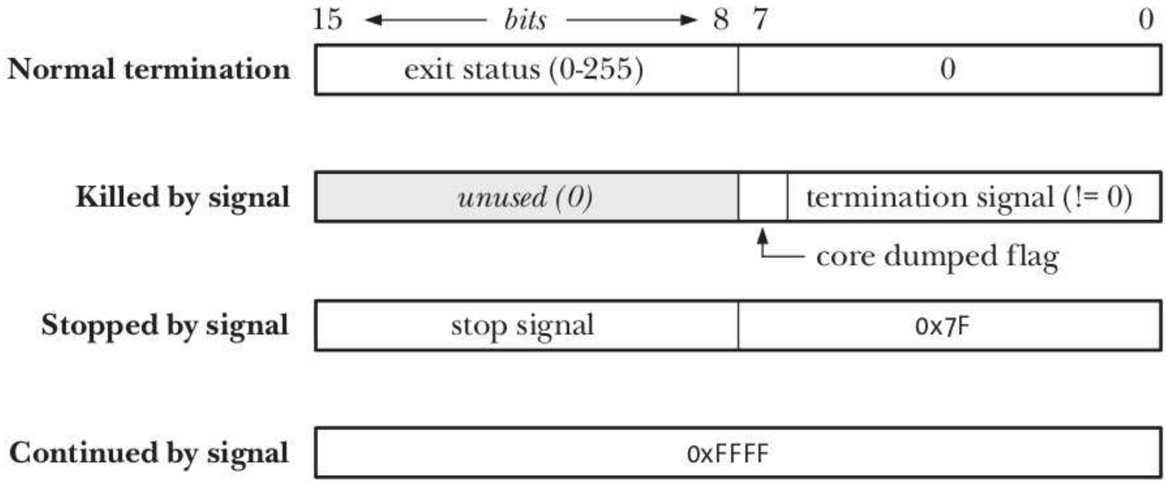
```
for (;;) {
    childPid = waitpid(-1, NULL, 0);
    if (childPid == -1) {
        if (errno == ECHILD) {
            printf("No more children!\n");
            break;
        } else {        /* Unexpected error */
            errExit("wait");
        }
    }

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


The wait status value

- *wstatus* returned by *waitpid()* 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 term **wait status** encompasses all four cases
- The term **termination status** covers the first two cases
  - In the shell, termination status of last command is available via *$?*

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**

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 using *upstart* as *init* system, or *systemd* in some configurations, 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

```
int main(int argc, char *argv[]) {
  int nzombies = (argc > 1) ? atoi(argv[1]) : 1;
  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());
      exit(EXIT_SUCCESS);
    default:          /* Parent continues in loop */
      break;
    }
  }

  sleep(600);    /* 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>
```

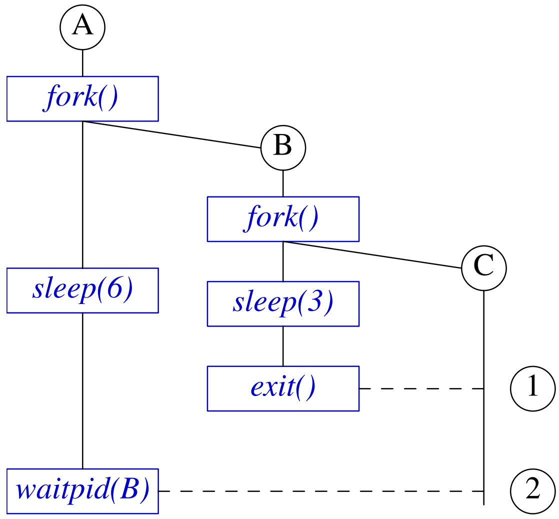
- **Zombies can't be killed** by signals!
  - (Since parent must still be able to "wait")
  - Even silver bullets (`SIGKILL`) don't work

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, parent, and child, 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 to be adopted by *init* (so that *getppid()* in the child returns 1): after the parent terminates or after the grandparent 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.grandchild_zombie.c`]**, to verify the answer.

Note the following points:

- You will probably want to use calls to *sleep()* so that you can more easily observe the steps that occur during execution of the program. For example:
  - The grandchild could loop 10 times, displaying the value returned by *getppid()* and sleeping for 1 second on each loop iteration.
  - The parent could sleep for 3 seconds before terminating.
  - The grandparent could sleep for 6 seconds before calling *wait()*
- Depending on your distribution (e.g., if you are running a Linux distribution that uses *upstart* as the *init* program, or a *systemd*-based system where the *--user* flag is employed), you will see slightly different results from those described above. In particular, the orphaned child is reparented to a process other than PID 1.

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
- Handler saves and restores *errno*, so that it is reentrant

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'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)* man 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
- *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
  - Normally same as basename part of *pathname*
  - Program can vary its behavior, depending on value of *argv[0]*
    - *busybox*

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

```
execve(pathname, argv, envp);
printf("execve() failed\n");
```

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 exit, 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());
        execve(argv[1], &argv[1], 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 specify each command using a (relative or absolute) **pathname**.

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 command-line arguments (i.e., *argc* is 1).

Exercise

3. Write a program (**[template:** `procexec/ex.make_link.c`**]**) that takes two 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

File descriptors and *exec()*

- By default, **file descriptors remain open across *exec()***
- Allows caller of *exec()* to open files for use by new program
  - The shell employs this feature to do I/O redirection
  - E.g., for redirection in this command: `prog > file`

```
fd = open("file", O_CREAT | O_WRONLY | O_TRUNC,
          0666);
dup2(fd, STDOUT_FILENO);
close(fd);
execvp("prog", ...);
```

[TLPI §27.4]

Notes

Notes

Notes

Linux/UNIX System Programming Fundamentals

System Call Tracing with strace

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

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*)?
- There is also a loosely related *ltrace(1)* command
  - Trace library function calls in dynamic shared objects (e.g., libc)
  - We won't cover this tool

strace(1)

- Log 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**
  - ***errno* values** displayed symbolically + matching error text
  - "large" arguments and structures are abbreviated by default

```
fstat(3, {st_dev=makedev(8, 2), st_ino=401567,
  st_mode=S_IFREG|0755, st_nlink=1, st_uid=0, st_gid=0,
  st_blksize=4096, st_blocks=280, st_size=142136,
  st_atime=2015/02/17-17:17:25, st_mtime=2013/12/27-22:19:58,
  st_ctime=2014/04/07-21:44:17}) = 0

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


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 vaue 0 or 1:

```
# echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

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

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 +++
```

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

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)* man 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

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`

```
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

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 -o strace.log -e signal=sigio,sigint ls > /dev/null
$ strace -o strace.log -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 `:errno` 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
  2.86    0.000085          43         2           socket
  2.82    0.000084          42         2         2 connect
...
------ ----------- ----------- --------- --------- --------------
100.00    0.002976                   442        13 total
```

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

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
  - ⚠ ⚠ 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

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

Notes

Linux/UNIX System Programming Fundamentals

Pipes and FIFOs

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

Outline

Pipes and FIFOs

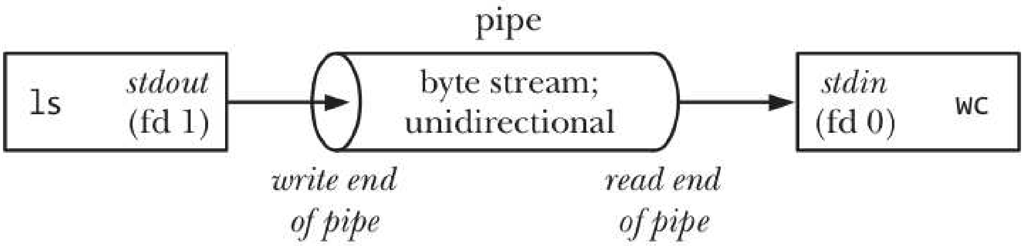
- Mechanisms for exchanging data between processes (IPC)
  - *pipe(7)* man page
- Have generally similar I/O semantics
- Principal difference is accessibility model
  - Pipes: "related" processes
  - FIFOs (named pipes):
    - Have a name in the filesystem
    - Accessibility: user/group ownership + file permissions
- For both mechanisms, data has **process persistence**
  - When all processes close FDs referring to pipe/FIFO, unread data is discarded

Pipes

- Pipes are a commonly used shell feature; e.g.:

```
$ ls | wc -l
```

- To execute this command, the shell:
  - Uses *fork()* to create two processes executing `ls` and `wc`
  - Connects standard output of `ls` and standard input of `wc` to pipe



- Data in pipe is held in kernel memory

Characteristics of pipes

- Pipes are **byte streams**
  - Data is an **undelimited sequence of bytes**
  - **Can read arbitrary blocks** of data, regardless of size of writes
  - Data passes through pipe sequentially (no random access)
- Pipes are **unidirectional**
  - Pipes have a **read end** and a **write end**

[TLPI §44.1]

Characteristics of pipes

- Pipes have a **limited capacity**
  - Limit varies across systems
- Pipe capacity on Linux:
  - Linux <= 2.6.10: 4096 bytes
  - Linux >= 2.6.11: 65,536 bytes
  - *fcntl(fd, F_SETPIPE_SZ, size)* can be used to change pipe capacity (since Linux 2.6.35)
- Applications should be designed not to care about capacity
  - To prevent writer from blocking, ensure that there is always an active reader

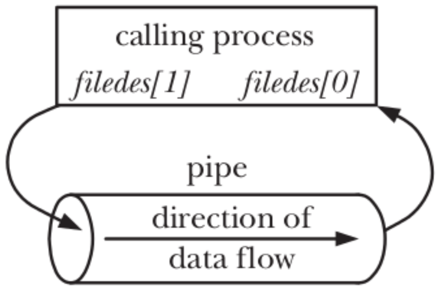
[TLPI §44.1]

Outline

Creating a pipe

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- Creates a new pipe
- Returns two file descriptors in *filedes*:
  - *filedes[0]* refers to read end of pipe
  - *filedes[1]* refers to write end of pipe
  - **Uses lowest free file descriptors**



[TLPI §44.2]

I/O on pipes

- I/O performed as usual, using *read()* and *write()*
  - Calls return number of bytes transferred
- **Reads**:
  - Return *min(# of bytes requested, # of bytes available)*
  - Block until at least one byte of data is available
  - Return end-of-file (0) if write end has been closed
    - (after all outstanding data in pipe has been read)

[TLPI §44.10]

I/O on pipes

Writing to a pipe:

- Insufficient space in pipe? ⇒ *write()* will block
  - Blocked write may be interrupted by a signal handler
    - No bytes yet written? ⇒ −1 return + `EINTR` error
    - Some data written? ⇒ partial write
- **If pipe has no reader**, a writer is informed:
  - *write()* causes generation of `SIGPIPE` signal
    - Default action: terminate process
    - Can make disposition "ignore", in which case...
  - *write()* fails with `EPIPE` error

[TLPI §44.10]

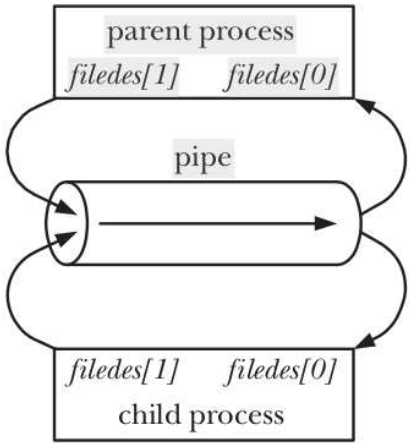
Pipe writes and PIPE_BUF

Writing to a pipe:

- **Writes ≤ `PIPE_BUF` bytes are guaranteed to be atomic**
  - If there is not enough space to write all bytes, none are written
    - Caller blocks until there is space to write all bytes in one operation
  - Bytes won't be intermingled with writes by other processes
  - `PIPE_BUF` == 4096 on Linux; as low as 512 on some systems
- **Writes > `PIPE_BUF` bytes may not be atomic**
  - Data may be broken into/transferred in smaller pieces
    - Possibly less than `PIPE_BUF` bytes (e.g., even single bytes, if that is all there is room for)
    - Pieces may be intermingled if there are multiple writers
  - *write()* completes when all data has been transferred

Connecting processes using pipes

- After creation, only one process knows about pipe
  - Limited uses...
- To connect two processes to pipe, use *fork()*
- Child inherits copies of parent's file descriptors
  - File descriptors refer to same pipe



[TLPI §44.2]

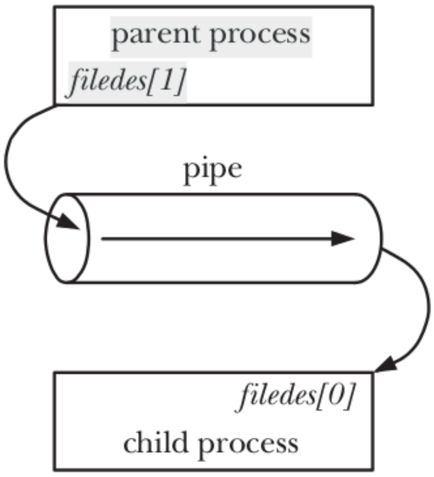
Closing unused descriptors

- Parent and child can now **both** read and write on pipe
- Usually undesirable to have both parent and child each reading and writing on pipe
  - Multiple readers would race for data
  - Multiple writers would have data intermingled
- Instead, data normally flows in one direction
- ⇒ after *fork()*, each process closes unused file descriptors

Closing unused descriptors

Suppose we want to transfer data from parent to child...

- Parent does not need read descriptor and child does not need write descriptor



- **Question**: why is it important to close the unused FDs?

[TLPI §44.2]

Closing unused file descriptors

Suppose we want to transfer data from parent to child:

```
int pfd[2];

pipe(pfd);          /* Create the pipe */

switch (fork()) {
case -1: errExit("fork");
case 0:     /* Child */
  close(pfd[1]);

  /* Child now reads from pipe */
  break;

default:    /* Parent */
  close(pfd[0]);

  /* Parent now writes to pipe */
  break;
}
```

Closing unused file descriptors is essential

- Closing unused descriptors is essential for correct use of pipes
- Reader sees EOF only when **all** write descriptors are closed
  - Instead, *read()* will block, waiting for data
- Writer gets `EPIPE` + `SIGPIPE` only if **all** read descriptors are closed
  - Instead, *write()* will succeed, or block if pipe is full

Example: `pipes/simple_pipe.c`

Parent sends *argv[1]* string to child, via pipe

```
int pfd[2];

pipe(pfd);              /* Create the pipe */

switch (fork()) {
case 0:                 /* Child - reads from pipe */
    close(pfd[1]);
    for (;;) {          /* Read data from pipe, echo on stdout */
        numRead = read(pfd[0], buf, BUF_SIZE);
        if (numRead == 0)
            break;                /* End-of-file */
        write(STDOUT_FILENO, buf, numRead);
    }
    write(STDOUT_FILENO, "\n", 1);
    close(pfd[0]);
    _exit(EXIT_SUCCESS);

default:                /* Parent - writes to pipe */
    close(pfd[0]);
    write(pfd[1], argv[1], strlen(argv[1]));
    close(pfd[1]);                /* Child will see EOF */
    wait(NULL);                   /* Wait for child to finish */
    exit(EXIT_SUCCESS);
}
```

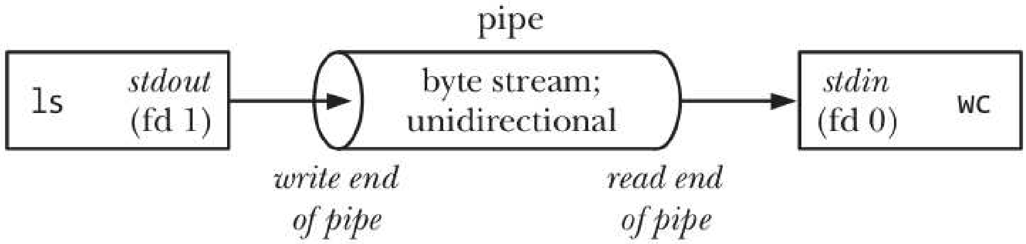
Exercise

1. Write a program (**[template:** **`pipes/ex.pipe_ucase.c`]**) that:
   - Creates a pipe
   - Creates a child process
   - The parent reads input from standard input (until end-of-file) and writes it to the pipe
   - The child reads from the pipe (until end-of-file), upper cases (*toupper(3)*) any letters that it reads, and writes the resulting text to standard output

Outline

Connecting a filter to a pipe: the problem

- Filter == program that reads from *stdin* and/or writes to *stdout*
- Suppose we want filter to read from or write to a pipe...



- What's the problem?
  - Normally, file descriptors 0, 1, and 2 are already in use
  - ⇒ *pipe()* will use 2 other descriptors

Connecting a filter to a pipe: the solution

- Solution is to use *dup()* (or similar)
- e.g., to connect filter to write end of pipe:

```
int pfd[2];
pipe(pfd);          /* Allocates (say) FDs 3 and 4 */
       /* ... Other steps here, e.g., fork() ... */
close(STDOUT_FILENO);                  /* Free FD 1 */
dup(pfd[1]);         /* Uses lowest free FD (FD1) */
```

- And since we no longer need *pfd[1]*, we should close it:

```
close(pfd[1]);
```

  - (Recall that unused descriptors must be closed...)
- But, what if descriptor 0 got closed between *pipe()* and (first) *close()*...?

Connecting a filter to a pipe: refining the solution

- *dup2(oldfd, newfd)* solves the problem:
  - Closes *newfd* if it was open
  - Makes *newfd* a duplicate of *oldfd*
    - (Preceding two steps are atomic; prevents FD races in multithreaded applications)
  - Does nothing if *oldfd* == *newfd*
- ⇒ Replace calls to *close()* and *dup()* with *dup2()*:

```
dup2(pfd[1], STDOUT_FILENO);
        /* Close FD 1, and reopen FD 1 bound
           to write end of pipe */

close(pfd[1]);            /* FD no longer needed */
```


Connecting a filter to a pipe: refining the solution

- But, what if descriptors 0 **and** 1 were closed before *pipe()*:

```
pipe(pfd);                       /* Uses FD 0 and FD 1 */

/* Let's presume write end of pipe used FD 1...*/

dup2(pfd[1], STDOUT_FILENO); /* dup2(1,1) [no-op] */
close(pfd[1]);               /* close(1) [!!] */
```

- ⚠ *dup2()* did nothing, and *close()* closed our only descriptor
- Solution: *dup2()* + *close()* not needed if *pipe()* used the descriptor we want:

```
pipe(pfd);

if (pfd[1] != STDOUT_FILENO) {
    dup2(pfd[1], STDOUT_FILENO);
    close(pfd[1]);
}
```

Example: pipes/pipe_ls_wc_simple.c

Implement `ls | wc -l` (error checking omitted)

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

switch (fork()) {
case 0:      /* Child: exec 'ls' to write to pipe */
    close(pfd[0]);          /* Read end is unused */

    /* Duplicate stdout on write end of pipe */
    if (pfd[1] != STDOUT_FILENO) {
        dup2(pfd[1], STDOUT_FILENO);
        close(pfd[1]);
    }
    execlp("ls", "ls", (char *) NULL);
    errExit("execlp ls");

default:     /* Parent: exec 'wc -l' to read from pipe */
    close(pfd[1]);          /* Write end is unused */

    /* Duplicate stdin on read end of pipe */
    if (pfd[0] != STDIN_FILENO) {
        dup2(pfd[0], STDIN_FILENO);
        close(pfd[0]);
    }
    execlp("wc", "wc", "-l", (char *) NULL);
    errExit("execlp wc");
}
```


Example: pipe/pipe_ls_wc_simple.c

Implement `ls | wc -l` (error checking omitted)

```
$ ./pipe_ls_wc_simple
61
$ ls | wc -l
61
```

Exercises

1. Create a program, (**[template: `pipes/ex.unique_tokens.c`]**), that takes **one filename argument** and uses *fork()*, *exec()*, *dup2()*, and *pipe()* to implement the following pipeline:

```
tr ' \t' '\012' < filename | sort -u
```

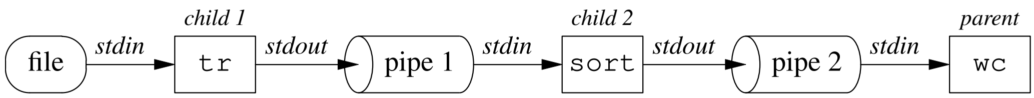
- The *tr* command converts spaces and tabs into newlines. (Input redirection is needed because *tr* doesn't take filename arguments.)
- The program `pipes/pipe_ls_wc_simple.c` provides a useful example for the solution of this problem.
- If you want to write debugging output, write it to standard error.
- To make *tr* read from `filename`, simply *open()* the file and duplicate (*dup2()*) the resulting FD onto `STDIN_FILENO`.

The `Makefile` provides a test: `make test_unique_tokens`

2. Extend the previous program to create a new program, `pipes/ex.count_unique_tokens.c`, that takes one filename argument and implements the following pipeline:

Exercises

```
tr ' \t' '\012' < filename | sort -u | wc -w
```



You can use `tokens.txt` as a test file. It contains 20 unique tokens.
The `Makefile` provides a test: `make test_count_unique_tokens`

3. Generalize the program created in the previous exercise to create a new version (**[template: `pipes/ex.pipeline_builder.c`]**) that implements and uses the following function:

```
int execlPipeline(int infd, bool makePipe, char *arg, ...)
```

This function creates a child process whose standard output is connected to the write end of a pipe created by the function. The child process executes the command specified in the variable-length list of arguments contained in *arg* and subsequent arguments.
[Exercise continues on next slide]

Exercises

Among other things, *execlPipeline()* should do the following:

- Before calling *fork()*, create a pipeline, if *makePipe* is nonzero.
- In the child:
  - Duplicate the file descriptor *infd* to be standard input, so that the child will read from that file descriptor.
  - (If *makePipe* is nonzero) duplicate the write end of the pipe so that it becomes the standard output of the command executed by the child.

As its function result, *execlPipeline()* returns the file descriptor for the read end of the pipe that it creates, or –1 if it did not create a pipe. Before returning, *execlPipeline()* closes *infd*. Using this function, the pipeline could be built using the following code:

```
fd =    open(argv[1], O_RDONLY);
fd =    execlPipeline(fd, true, "tr", " \t", "\\012",
                      (char *) NULL);
fd =    execlPipeline(fd, true, "sort", "-u", (char *) NULL);
(void) execlPipeline(fd, false, "wc", "-l", (char *) NULL);
```


Exercises

Hints:

- (In the child), you will need to make use of the *stdarg(3)* APIs, in order to parse the variable-length argument list. You may find it useful to examine the `procexec/execlp.c` source file for an example of how to build an *argv*-style vector from a variable-length argument list.
- Don't forget to close superfluous pipe file descriptors.

4. Write a program with the following command-line arguments:

```
$ ./pipe_speed num-blocks wblock-size rblock-size
```

The program does the following:

- Creates a pipe.
- Calls *fork()* to create a child process
- The child reads blocks of data of size *rblock-size* from the pipe, until end-of-file.
  [Exercise continues on next slide]

Exercises

- The parent:
  - Writes *num-blocks* blocks of size *wblock-size* to the pipe.
  - Closes the pipe.
  - Waits for the child to terminate.

Time the operation of the program for various values of *num-blocks* and *wblock-size*.

5. The Linux-specific *fcntl(fd, F_SETPIPE_SZ, size)* operation sets the capacity of a pipe to at least *size* bytes, and returns the new capacity. (In the current kernel implementation, the kernel rounds *size* up to the next power-of-two multiple of the page size.)

   Modify the preceding program to allow an optional fourth command-line argument (an integer) that should be used in a `F_SETPIPE_SZ` operation on the pipe. Does making the pipe capacity smaller (say, 4096 bytes) affect the rate of data transfer?

Exercises

6. Read the *sched_setaffinity(2)* man page. Modify the program so that you can choose which CPUs the parent and child run on. Try different combinations of CPUs with "small" block sizes ($\leq$ 1024, say). Do you see any differences in the data transfer rates? If yes, what might be the reason?

Outline

FIFOs

- "First-In First Out"
- Semantically similar to pipes
- Main difference: FIFO has a name in filesystem
  - ⇒ sometimes called "named pipes"
- Any process with permission to open FIFO can perform I/O
- To create in shell: `mkfifo [-m permissions] pathname`

```
$ mkfifo -m u+rw,g=,o= myfifo
$ ls -lF myfifo
prw-------. 1 mtk mtk 0 Oct 31 13:21 myfifo|
```

- To create from a program:

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

- When no longer needed, remove with *unlink()* or *remove()*

[TLPI §44.7]

Opening a FIFO

- Use *open()*:

```
fd = open("myfifo", O_RDONLY); /* Open read end */
fd = open("myfifo", O_WRONLY); /* Open write end */
```

- Opening one end of a FIFO **blocks until another processes opens the other end**
  - If other end is open, *open()* succeeds immediately
  - Opens are synchronized
- Rationale: FIFO is useful only if there is a reader **and** a writer

I/O on FIFOs

- Exactly as for pipes (*read()*, *write()*)
- Note: FIFO data is a buffer **in the kernel**
  - FIFO has filesystem pathname, but that is just a mechanism that allows multiple processes to access same buffer
- If all FDs are closed, unread data is discarded
  - (FIFO **name** persists in filesystem, but **data has process persistence**)

Exercise

1. Try the following with your favorite text file in a shell session:

```
$ mkfifo myfifo
$ tr 'aeiou' 'aieuo' < myfifo &
$ man 2 pipe > myfifo
```


Notes

Notes

Notes

Linux/UNIX System Programming Fundamentals

Alternative I/O Models

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

Outline

The traditional file I/O model

- I/O on **one file at a time**
  - *read()*, *write()*, etc. operate on single descriptor
- **Blocking I/O**
  - I/O not possible ⇒ call blocks until I/O becomes possible
  - Examples:
    - *write()* to pipe blocks if insufficient space
    - *read()* from socket that has no data available
- But sometimes, we want to:
  - **Check if I/O is possible without blocking** if it is not
  - **Monitor multiple file descriptors** to see if I/O is possible on any of them

[TLPI §63.1]

Outline

Nonblocking I/O

- Nonblocking I/O ⇒ **return error instead of blocking**
  - `EAGAIN` error for *read()*, *write()*, and similar
- Enabled via `O_NONBLOCK` file status flag
  - Set during *open()*; can also be enabled via *fcntl()*:

```
flags = fcntl(fd, F_GETFL);
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

  - Recall: file status flags reside in open file description
- Many APIs that create FDs also have a flag that allows nonblocking mode to be set at time FD is created
  - E.g., *eventfd()*, *inotify_init1()*, *open()*, *pipe2()*, *signalfd()*, *socket()*, *timerfd_create()*

EAGAIN vs EWOULDBLOCK

- On BSD, `EWOULDBLOCK` was/is returned instead of `EAGAIN`
- Many modern systems address this portability issue by making `EAGAIN` and `EWOULDBLOCK` synonyms
  - POSIX explicitly permits this
  - Linux does this

Use cases for nonblocking I/O

- Check if I/O is possible without blocking if not ("**polling**")
  - Mark file descriptor nonblocking
  - Perform I/O system call
  - If I/O call fails, try again later
- Perform as much I/O as possible, without blocking on completion
  - Mark file descriptor nonblocking
  - Perform I/O in a loop until `EAGAIN` encountered
- Nonblocking *accept()*
  - Make listening socket nonblocking
  - ⇒ *accept()* returns with `EAGAIN`/`EWOULDBLOCK` if no pending connection
- We'll see some other valid use cases for nonblocking I/O
  - E.g., I/O while employing edge-triggered *epoll* notification

Problems with nonblocking I/O

- Using nonblocking I/O for **repeatedly polling multiple file descriptors is problematic**
  - Frequent polling ⇒ CPU cycles wasted
  - Infrequent polling ⇒ high I/O latency
- We need better techniques...

Better techniques for managing multiple file descriptors

- *poll()*, *select()* ("I/O multiplexing"):
  - **Simultaneously monitor multiple FDs** to see if I/O is possible on any of them
- **Signal-driven I/O**:
  - Kernel sends process a **signal when I/O is possible** on FD
  - Better performance than *select()* / *poll()*
- ***epoll***:
  - Monitor multiple FDs (like *select()* / *poll()*)
  - Better performance and more features than *select()* / *poll()*
  - Simpler to program than signal-driven I/O
  - Linux-specific (since kernel 2.6.0)
- Above techniques only **monitor** FDs to see if I/O is possible
  - Actual I/O is performed using traditional system calls

Outline

Signal-driven I/O

- *Somewhat* portable technique for monitoring multiple FDs
- Process performs following steps:
  - Establish signal handler (default notification signal is `SIGIO`)
  - Mark itself as "owner" of FD (process that is to receive signals)
    - *fcntl(fd, F_SETOWN, pid)* operation
  - Enable signaling when I/O is possible on FD
    - Set `O_ASYNC` flag using *fcntl(fd, F_SETFL, flags)*
  - Carry on to do other tasks
  - When I/O becomes possible, signal handler is invoked
- Can enable I/O signaling on multiple FDs
- Better performance than *poll()*/*select()*
  - (For same reasons as *epoll*, as explained later)

[TLPI §63.3]

Signal-driven I/O

- Fully exploiting signal-driven I/O **requires use of Linux-specific features**
  - Choosing (realtime) signal via *fcntl(fd, F_SETSIG, sig)*
    - Default signal (`SIGIO`) is a **nonqueuing** traditional signal
  - Use `SA_SIGINFO` handler
    - ⇒ obtain file descriptor via *si_fd* field of *siginfo_t* structure
- ***epoll* API is more feature-rich** for task of monitoring multiple FDs
- ⇒ We'll ignore signal-driven I/O
  - (See TLPI §63.3 for more info + example program)

Outline

I/O multiplexing

- Monitor multiple file descriptors to see if I/O is possible on any of them
- Terminology: the FD is "**ready**" for I/O
  - Often, we'll talk of monitoring I/O events, but...
  - Strictly speaking, **these APIs tell us whether an I/O system call would block**
- Two traditional techniques:
  - *select()* (4.2BSD, 1983)
  - *poll()* (System V Release 3, 1986)
  - Both specified in POSIX and widely available
- Can be applied to any file type
  - Pipes, FIFOs, terminals, devices, sockets...
  - Applicable to regular files, but not very useful

[TLPI §63.2]

poll() and *select()*

- *select()* and *poll()* perform same task
- Differ primarily in how FDs are specified:
  - *select()*:
    - Arguments: 3 FD sets for 3 classes of readiness
    - Each FD set contains a set of FDs
  - *poll()*:
    - Argument: list (array) of file descriptors
    - Each array element specifies type of readiness to test

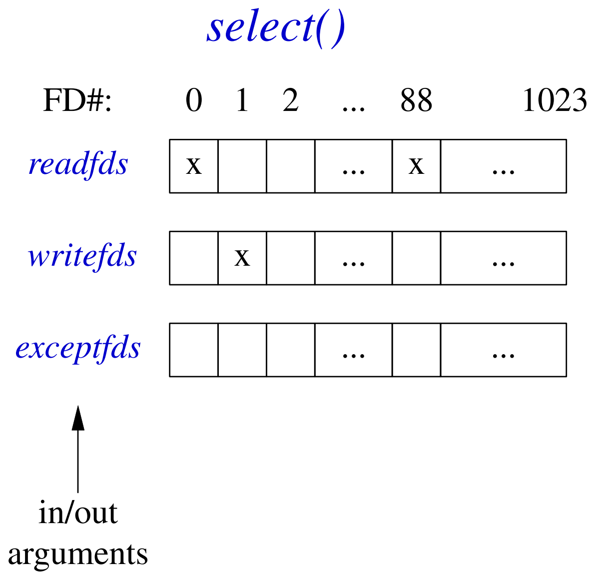
[TLPI §63.2.2]

Arguments of *poll()* and *select()*

poll() pollfds[ ]

fd	*events*	*revents*
0	`POLLIN`	
1	`POLLOUT`	
88	`POLLIN`	

input value — output value



poll() vs *select()*

- *poll()* fixes some of the API problems of *select()*
  - *select()* uses fixed-size FD sets
    - Only FDs $<$ 1024 can be monitored
    - Limitation of glibc, not kernel
  - *select()* uses same arguments for input and output
    - (Must reinitialize on each call inside a loop)
- $\Rightarrow$ We'll focus on *poll()*

[TLPI §63.2.2]

poll()

```
#include <poll.h>
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

- *fds*: list of file descriptors to be monitored
- *nfds*: number of elements in *fds*
- *timeout*: timeout if call blocks because no FD is yet ready for I/O

[TLPI §63.2.2]

The *pollfd* array

```
struct pollfd {
    int   fd;       /* File descriptor */
    short events;   /* Requested events bit mask */
    short revents;  /* Returned events bit mask */
};
```

- *fds* argument to *poll()* is list of file descriptors to monitor
- For each list element:
  - *events*: bit mask of **events to monitor** for *fd*
    - Input value, initialized by caller
  - *revents*: returned bit mask of **events that occurred** for *fd*
    - Output value, set by kernel

poll() events bits

Bit	Input in *events*?	Output in *revents*?	Description
POLLIN	•	•	Normal-priority data can be read
POLLPRI	•	•	High-priority data/exceptional condition
POLLRDHUP	•	•	Shutdown on peer socket
POLLOUT	•	•	Data can be written
POLLERR		•	An error has occurred
POLLHUP		•	A hangup occurred
POLLNVAL		•	File descriptor is not open

- Following bits can be specified in *events*; they will be returned in *revents* only if specified in *events*:
  - POLLIN, POLLPRI, and POLLRDHUP indicate **input** events
  - POLLOUT indicates an **output** event
- POLLERR, POLLHUP, and POLLNVAL are returned in *revents* to provide **additional info** about FD
  - Ignored if specified in *events*

poll() events bits

A few *poll()* events bits need some explanation:

- POLLPRI:
  - State change on pseudoterminal master in packet mode
  - Out-of-band data on stream socket
  - (Rarely used)
- POLLHUP:
  - Returned on read end of pipe/FIFO if write end is closed
- POLLERR:
  - Returned on write end of pipe/FIFO if read end is closed
- POLLRDHUP:
  - Stream socket peer has closed (writing half of) connection
  - Linux-specific, since kernel 2.6.17
    - Useful with *epoll* edge-triggered mode (see *epoll_ctl(2)*)
- POSIX is vague on specifics; details vary across systems

[TLPI §63.2.3]

poll() timeout

- *timeout* determines blocking behavior of *poll()*:
  - –1: block indefinitely
  - 0: don't block ("poll" current state of descriptors)
  - > 0: block for up to *timeout* milliseconds
- When blocking, *poll()* waits until either:
  - A file **descriptor becomes ready**
  - A **signal handler interrupts** the call
  - The **timeout** is reached

poll() return value

Return value from *poll()* is one of:

- > 0: number of ready FDs
  - I.e., number of elements in *pollfd* array that have *revents != 0*
- 0: *poll()* timed out without any FD becoming ready
- –1: error

Example: altio/poll_pipes.c

```
./poll_pipes num-pipes [num-writes]
```

- Create *num-pipes* pipes
- Loop *num-writes* times, each time writing a single byte to the write end of a randomly selected pipe
- Employ *poll()* to monitor all of the pipe read ends to see which pipes are readable
- Scan the *pollfd* array returned by *poll()* and print list of readable pipes

Example: altio/poll_pipes.c

```
int numPipes, ready, randPipe, numWrites, j;
struct pollfd *pollFd;
int (*pfds)[2];    /* File descriptors for all pipes */

numPipes = getInt(argv[1], GN_GT_0, "num-pipes");
numWrites = (argc > 2) ?
            getInt(argv[2], GN_GT_0, "num-writes") : 1;

pfds = calloc(numPipes, sizeof(int [2]));
pollFd = calloc(numPipes, sizeof(struct pollfd));
```

- Because number of pipes is selected at run-time, we must allocate structures at run time
- *getInt()* converts string to integer
- Allocate array for pipe pairs
  - *calloc()* == *malloc(nmemb * size)*, and also zeroes memory
- Allocate *pollfd* array

Example: `altio/poll_pipes.c`

```c
for (j = 0; j < numPipes; j++)
    pipe(pfds[j]);

srandom((int) time(NULL));      /* Seed RNG */
for (j = 0; j < numWrites; j++) {
    randPipe = random() % numPipes;
    printf("Writing to fd: %3d (read fd: %3d)\n",
            pfds[randPipe][1], pfds[randPipe][0]);
    write(pfds[randPipe][1], "a", 1);
}
```

- Create pipe pairs
- Loop *num-writes* times, writing a byte to a randomly selected pipe
    - Display FD for write and read end of pipe

Example: `altio/poll_pipes.c`

```c
for (j = 0; j < numPipes; j++) {
    pollFd[j].fd = pfds[j][0];
    pollFd[j].events = POLLIN;
}
ready = poll(pollFd, numPipes, 0);

printf("poll() returned: %d\n", ready);

for (j = 0; j < numPipes; j++)
    if (pollFd[j].revents & POLLIN)
        printf("Readable: %3d\n", pollFd[j].fd);
```

- Build *pollfd* array containing all pipe read ends
    - Monitor to see if input is possible (`POLLIN`)
- Call *poll()* with zero *timeout*
- Return value from *poll()* is number of ready FDs
- Walk through *revents* fields in *pollfd* array, to see which FDs are ready for reading

Exercise

1. Write a program (**[template:** `altio/ex.poll_pipes_write.c`**]**) that has the following command-line syntax:

```
./poll_pipes_write num-pipes [num-writes [block-size]]
```

The program should create *num-pipes* pipes, and make the write ends of each pipe nonblocking (set the `O_NONBLOCK` flag with *fcntl(F_SETFL)*; see slide 11-6).

The program should then loop *num-writes* (default: 1) times, each time writing *block-size* (arbitrary) bytes (default: 100) to a randomly selected pipe. During the loop, the program should count the number of writes that failed because the pipe was full (*write()* failed with `EAGAIN` in *errno*) and the number of partial writes (*write()* wrote fewer bytes than requested).

After the above loop completes, the program should employ a (nonblocking) *poll()* call to monitor all of the pipe **write** ends to see which pipes are still writable, and then report the following:

- A list of the pipes that are writable
- The total number of partial writes
- The total number of times that *write()* failed with `EAGAIN`

Vary the command-line arguments until you see instances of `EAGAIN` errors and partial writes. What is the minimum *block-size* needed in order to see partial writes?

Outline

Problems with *poll()* and *select()*

- *poll()* + *select()* are portable, long-standing, and widely used
- But, there are scalability problems when monitoring many FDs, because, on each call:
  1. Program passes a data structure to kernel describing **all** FDs to be monitored
  2. The kernel must recheck **all** specified FDs for readiness
     - This includes hooking (and subsequently unhooking) all FDs to handle case where it is necessary to block
  3. The kernel passes a modified data structure describing readiness of **all** FDs back to program in user space
  4. After the call, the program must inspect readiness state of **all** FDs in modified data
- ⇒ Cost of *select()* and *poll()* scales with number of FDs being monitored

[TLPI §63.2.5]

Problems with *poll()* and *select()*

- *poll()* and *select()* have a design problem:
  - Typically, set of FDs monitored by application is static
    - (Or set changes only slowly)
  - But, kernel doesn't remember monitored FDs between calls
    - ⇒ Info on all FDs must be copied back & forth on each call
- *epoll* improves performance by fixing this design problem
  - Kernel maintains a persistent set of FDs that application is interested in
  - Application can **incrementally** change "interest list"
- *epoll* cost **scales according to number of I/O events**
  - **Much better performance when monitoring many FDs!**
  - Signal-driven I/O scales similarly, for same reasons

[TLPI §63.4.5]

Outline

Overview

- Like *select()* and *poll()*, *epoll* can monitor multiple FDs
- *epoll* returns readiness information in similar manner to *poll()*
- Two main **advantages**:
  - *epoll* provides **much better performance** when monitoring large numbers of FDs (see TLPI §63.4.5)
  - *epoll* provides two **notification modes**: **level-triggered** and **edge-triggered**
    - Default is level-triggered notification
    - *select()* and *poll()* provide only level-triggered notification
    - (Signal-driven I/O provides only edge-triggered notification)
- Linux-specific, since kernel 2.6.0

[TLPI §63.4]

epoll instances

Central data structure of *epoll* API is an ***epoll* instance**

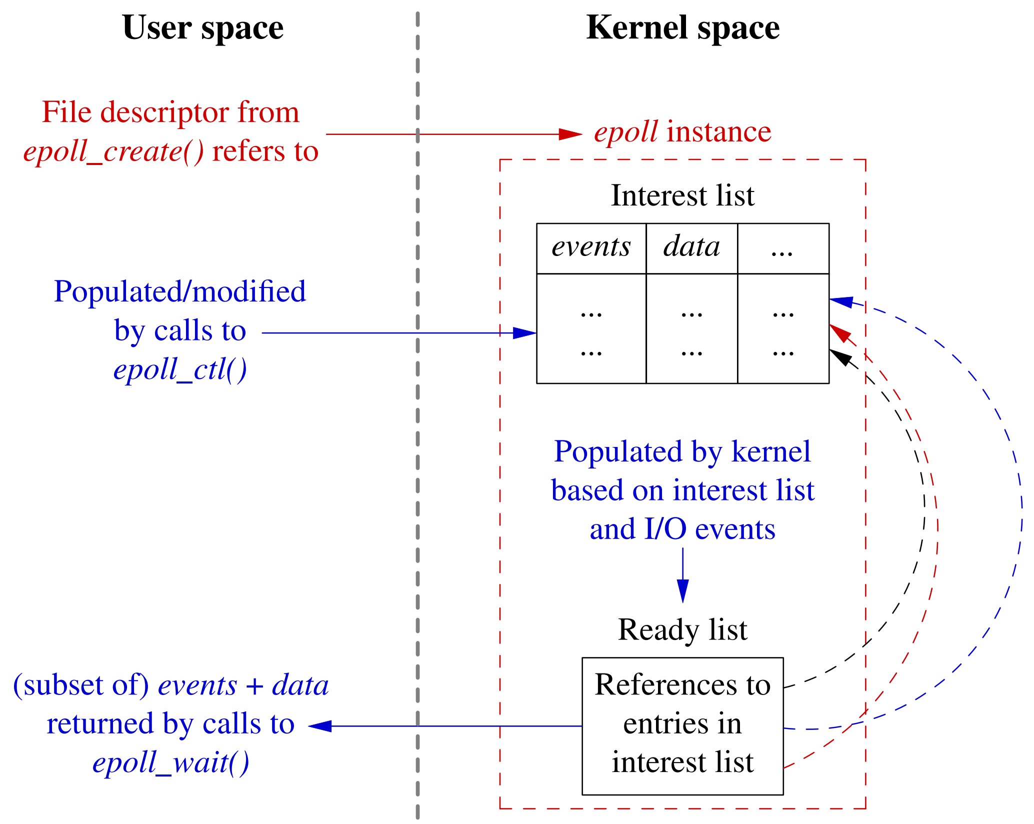
- **Persistent** data structure **maintained in kernel space**
  - Referred to in user space via file descriptor
- Can (abstractly) be considered as container for two lists:
  - **Interest list**: list of FDs to be monitored
  - **Ready list**: list of FDs that are ready for I/O
    - Ready list is (dynamic) subset of interest list

epoll APIs

The key *epoll* APIs are:

- *epoll_create()*: create a new *epoll* instance and return FD referring to instance
  - FD is used in the calls below
- *epoll_ctl()*: modify interest list of *epoll* instance
  - Add FDs to/remove FDs from interest list
  - Modify events mask for FDs currently in interest list
- *epoll_wait()*: return items from ready list of *epoll* instance

epoll kernel data structures and APIs



Creating an *epoll* instance: *epoll_create()*

```
#include <sys/epoll.h>
int epoll_create(int size);
```

- Creates an *epoll* instance
- *size*:
  - Since Linux 2.6.8: serves no purpose, but must be > 0
  - Before Linux 2.6.8: an *estimate* of number of FDs to be monitored via this *epoll* instance
- Returns file descriptor on success, or –1 on error
  - When FD is no longer required, it should be closed via *close()*
- Since Linux 2.6.27, *epoll_create1()* provides improved API
  - See the man page

[TLPI §63.4.1]

Modifying the *epoll* interest list: *epoll_ctl()*

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd,
              struct epoll_event *ev);
```

- Modifies the interest list associated with *epoll* FD, *epfd*
- *fd*: identifies which FD in interest list is to have its settings modified
  - E.g., FD for pipe, FIFO, terminal, socket, POSIX MQ, or even another *epoll* FD
    - (Can't be FD for a regular file or directory)
- *op*: operation to perform on interest list
- *ev*: (Later)

[TLPI §63.4.2]

epoll_ctl() op argument

The *epoll_ctl() op* argument is one of:

- `EPOLL_CTL_ADD`: add *fd* to interest list of *epfd*
  - *ev* specifies events to be monitored for *fd*
  - If *fd* is already in interest list ⇒ `EEXIST`
- `EPOLL_CTL_MOD`: modify settings of *fd* in interest list of *epfd*
  - *ev* specifies new settings to be associated with *fd*
  - If *fd* is not in interest list ⇒ `ENOENT`
- `EPOLL_CTL_DEL`: remove *fd* from interest list of *epfd*
  - Also removes corresponding entry in ready list, if present
  - *ev* is ignored
  - If *fd* is not in interest list ⇒ `ENOENT`
  - **Closing an FD automatically removes it from all *epoll* interest lists**
    - ⚠ But see later! Manual deletion is sometimes required

The *epoll_event* structure

epoll_ctl() ev argument is pointer to an *epoll_event* structure:

```
struct epoll_event {
    uint32_t     events;  /* epoll events (bit mask) */
    epoll_data_t data;    /* User data */
};

typedef union epoll_data {
    void    *ptr;    /* Pointer to user-defined data */
    int      fd;     /* File descriptor */
    uint32_t u32;    /* 32-bit integer */
    uint64_t u64;    /* 64-bit integer */
} epoll_data_t;
```

- *ev.events*: bit mask of events to monitor for *fd*
    - (Similar to *events* mask given to *poll()*)
- *data*: info to be passed back to caller of *epoll_wait()* when *fd* later becomes ready
    - **Union field**: value is specified in *one* of the members

Example: using *epoll_create()* and *epoll_ctl()*

```
int epfd;
struct epoll_event ev;

epfd = epoll_create(5);

ev.data.fd = fd;
ev.events = EPOLLIN;  /* Monitor for input available */
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```

Outline

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- Returns info about ready FDs in interest list of *epoll* instance of *epfd*
- Blocks until at least one FD is ready
- Info about ready FDs is returned in array *evlist*
  - I.e., can get information about multiple ready FDs with one *epoll_wait()* call
  - (Caller allocates the *evlist* array)
- *maxevents*: size of the *evlist* array

[TLPI §63.4.3]

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- *timeout* specifies a timeout for call:
  - –1: block until an FD in interest list becomes ready
  - 0: perform a nonblocking "poll" to see if any FDs in interest list are ready
  - $>$ 0: block for up to *timeout* milliseconds or until an FD in interest list becomes ready
- Return value:
  - $>$ 0: number of items placed in *evlist*
  - 0: no FDs became ready within interval specified by *timeout*
  - –1: an error occurred

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- Info about **multiple** FDs can be returned in the array *evlist*
- Each element of *evlist* returns info about one file descriptor:
  - *events* is a bit mask of events that have occurred for FD
  - *data* is *ev.data* value *currently* associated with FD in the interest list
- **NB**: the FD itself is **not** returned!
  - Instead, we put FD into *ev.data.fd* when calling *epoll_ctl()*, so that it is returned via *epoll_wait()*
    - (Or, put FD into a structure pointed to by *ev.data.ptr*)

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- If > *maxevents* FDs are ready, successive *epoll_wait()* calls round-robin through FDs
  - Helps prevent file descriptor starvation
- In multithreaded programs:
  - One thread can modify interest list (*epoll_ctl()*) while another thread is blocked in *epoll_wait()*
  - *epoll_wait()* call will return if a newly added FD becomes ready

epoll events

Following table shows:

- Bits given in *ev.events* to *epoll_ctl()*
- Bits returned in *evlist[].events* by *epoll_wait()*

Bit	*epoll_ctl()*?	*epoll_wait()*?	Description
EPOLLIN	•	•	Normal-priority data can be read
EPOLLPRI	•	•	High-priority data can be read
EPOLLRDHUP	•	•	Shutdown on peer socket
EPOLLOUT	•	•	Data can be written
EPOLLONESHOT	•		Disable monitoring after event notification
EPOLLET	•		Employ edge-triggered notification
EPOLLERR		•	An error has occurred
EPOLLHUP		•	A hangup occurred

- Other than `EPOLLONESHOT` and `EPOLLET`, bits have same meaning as similarly named *poll()* bit flags
- `EPOLLIN`, `EPOLLPRI`, `EPOLLRDHUP`, and `EPOLLOUT` are returned by *epoll_wait()* only if specified when adding FD using *epoll_ctl()*

[TLPI §63.4.3]

Example: altio/epoll_input.c

```
./epoll_input file...
```

- Monitors one or more files using *epoll* API to see if input is possible
- Suitable files to give as arguments are:
  - FIFOs
  - Terminal device names
    - (May need to run *sleep* command in FG on the other terminal, to prevent shell stealing input)
  - Standard input
    - /dev/stdin

Example: altio/epoll_input.c (1)

```
#define MAX_BUF     1000    /* Max. bytes for read() */
#define MAX_EVENTS     5
     /* Max. number of events to be returned from
        a single epoll_wait() call */

int epfd, ready, fd, s, j, numOpenFds;
struct epoll_event ev;
struct epoll_event evlist[MAX_EVENTS];
char buf[MAX_BUF];

epfd = epoll_create(argc - 1);
```

- Declarations for various variables
- Create an *epoll* instance, obtaining *epoll* FD

Example: altio/epoll_input.c (2)

```
for (j = 1; j < argc; j++) {
    fd = open(argv[j], O_RDONLY);
    printf("Opened \"%s\" on fd %d\n", argv[j], fd);

    ev.events = EPOLLIN;
    ev.data.fd = fd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
}

numOpenFds = argc - 1;
```

- Open each of the files named on command line
- Each file is monitored for input (EPOLLIN)
- *fd* placed in *ev.data*, so it is returned by *epoll_wait()*
- Add the FD to *epoll* interest list (*epoll_ctl()*)
- Track the number of open FDs

Example: altio/epoll_input.c (3)

```
while (numOpenFds > 0) {
    printf("About to epoll_wait()\n");
    ready = epoll_wait(epfd, evlist, MAX_EVENTS, -1);
    if (ready == -1) {
        if (errno == EINTR)
            continue;       /* Restart if interrupted
                               by signal */
        else
            errExit("epoll_wait");
    }
    printf("Ready: %d\n", ready);
```

- Loop, fetching *epoll* events and analyzing results
- Loop terminates when all FDs has been closed
- *epoll_wait()* call places up to MAX_EVENTS events in *evlist*
  - *timeout* $== -1 \Rightarrow$ infinite timeout
- Return value of *epoll_wait()* is number of ready FDs

Example: altio/epoll_input.c (4)

```
    for (j = 0; j < ready; j++) {
        printf("  fd=%d; events: %s%s%s\n", evlist[j].data.fd,
            (evlist[j].events & EPOLLIN)  ? "EPOLLIN "  : "",
            (evlist[j].events & EPOLLHUP) ? "EPOLLHUP " : "",
            (evlist[j].events & EPOLLERR) ? "EPOLLERR " : "");
        if (evlist[j].events & EPOLLIN) {
            s = read(evlist[j].data.fd, buf, MAX_BUF);
            printf("    read %d bytes: %.*s\n", s, s, buf);
        } else if (evlist[j].events & (EPOLLHUP | EPOLLERR)) {
            printf("    closing fd %d\n", evlist[j].data.fd);
            close(evlist[j].data.fd);
            numOpenFds--;
        }
    }
}
```

- Scan up to *ready* items in *evlist*
- Display *events* bits
- If `EPOLLIN` event occurred, read some input and display it on *stdout*
  - `%.*s` ⇒ print string with field width taken from argument list (*s*)
- Otherwise, if error or hangup, close FD and decrements FD count
- Code correctly handles case where both `EPOLLIN` and `EPOLLHUP` are set in *evlist[j].events*

Exercises

1. Write a client (**[template: `altio/ex.is_chat_cl.c`]**) that communicates with the TCP chat server program, `is_chat_sv.c`. The program should be run with the following command line:

```
./is_chat_cl <host> <port> [<nickname>]
```

The program should create a connection to the server, and then use the *epoll* API to monitor both the terminal and the TCP socket for input. All input that becomes available on the socket should be written to the terminal and vice versa.

- Each time the program sends input from the terminal to the socket, that input should be prepended by the nickname supplied on the command line. If no nickname is supplied, then use the string returned by *getlogin(3)*. (*snprintf(3)* provides an easy way to concatenate the strings.)
- The program should terminate if it detects end-of-file or an error condition on either file descriptor.
- Calling *epoll_wait()* with *maxevents==1* will simplify the code!
- Bonus points if you find a way to crash the server (reproducibly)!

Exercises

2. Write the chat server (**[template: `altio/ex.is_chat_sv.c`]**). Note the following points:
   - The program should take one command-line argument: the port number to which it should bind its listening socket.
   - The program should accept and handle multiple simultaneous client connections. Input read from any client should be broadcast to all other clients.
   - Use the *epoll* API to manage the file descriptors.
   - You should use nonblocking file descriptors to ensure that the server never blocks when accepting connections or when reading or writing to clients.
   - When the server detects end-of file or an error (other than `EAGAIN`) while reading or writing on a client connection, it should close that connection. (Remember that closing a file descriptor automatically removes it from any *epoll* interest lists of which it is a member.)

Exercises

3. Write a program (**[template: `altio/ex.epoll_pipes.c`]**) which performs the same task as the `altio/poll_pipes.c` program, but uses the *epoll* API instead of *poll()*.
   Hints:
   - After writing to the pipes, you will need to call *epoll_wait()* in a loop. The loop should be terminated when *epoll_wait()* indicates that there are no more ready file descriptors.
   - After each call to *epoll_wait()*, you should display each ready pipe read file descriptor and then drain all input from that file descriptor so that it does not indicate as ready in future calls to *epoll_wait()*.
   - In order to drain a pipe without blocking, you will need to make the file descriptor for the read end of the pipe nonblocking.

Outline

Edge-triggered notification

- By default, *epoll* provides **level-triggered** (LT) notification
  - Tells us whether an **I/O operation can be performed on FD without blocking**
  - Like *poll()* and *select()*
- `EPOLLET` provides **edge-triggered** (ET) notification
  - Has I/O **activity occurred since *epoll_wait()* last notified about this FD**?
    - Or, if no *epoll_wait()* since FD was added/modified by *epoll_ctl()*, then: is FD ready?
- Example:

```
struct epoll_event ev;
ev.data.fd = fd
ev.events = EPOLLIN | EPOLLET;
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```

[TLPI §63.4.6]

Edge-triggered notification

- Illustration of difference between LT and ET notification:
  1. Monitoring a socket for input possible (`EPOLLIN`)
  2. Input arrives on socket
  3. We call *epoll_wait()*, which informs us that FD is ready
     - We *perhaps* consume some (**but not all**) available input
     - **No further input arrives on socket**
  4. We call *epoll_wait()* again
- LT notification: second *epoll_wait()* would (again) report FD as ready
  - Because outstanding data is still available for reading
- ET notification: second *epoll_wait()* would **not** report FD as ready
  - Because no I/O activity occurred since previous *epoll_wait()*

Uses for edge-triggered notification

- Can be more efficient: application is not repeatedly reminded that FD is ready
- Example: application that (periodically) generates data to be written to a socket
  - Application does not always have data to write
  - Application monitors socket for writability (`EPOLLOUT`)
    - Application is also monitoring other FDs for I/O possible
  - At some point, socket is full (output not possible)
  - Peer drains some data, socket becomes writable
  - LT notification: every *epoll_wait()* would (immediately) wake and say FD is writable
  - ET notification: only first *epoll_wait()* would say FD is writable
    - Application could cache that info for later action (e.g., when data is generated)

Edge-triggered notification provides an optimization

- Scenario: multiple threads/processes are *epoll_wait()*-ing on same *epoll* FD
  - E.g., *epoll* FD is monitoring listening socket
  - LT notification: **all** waiters are woken up when connection request arrives
  - ET notification: only **one** waiter is woken up
    - Avoids thundering herd problem
  - Code examples: `altio/multithread_epoll_wait.c`, `altio/epoll_flags_fork.c`
  - The `EPOLLEXCLUSIVE` flag provides a similar behavior in some scenarios when using level-triggered notification
    - Since Linux 4.5
    - See *epoll_ctl(2)* and `altio/epoll_flags_fork.c`

Edge-triggered notification and EPOLLONESHOT

- Scenario: monitoring socket for input available with `EPOLLET`
  - Assumption: we want to know when input is available, but don't want to read it **yet**
    - (So, we use `EPOLLET` to avoid repeated notifications)
- New input keeps appearing ⇒ ET notification keeps notifying
  - Really, we needed only **first** notification
- Solution: `EPOLLONESHOT`

One-shot monitoring: `EPOLLONESHOT`

- Specifying `EPOLLONESHOT` in *events* causes FD to be reported just once by *epoll_wait()*
- FD is then marked inactive in interest list
- FD remains in interest list, and can be reactivated using *epoll_ctl(EPOLL_CTL_MOD)*
  - Continuing previous example: reenable notification after draining input from socket

[TLPI §63.4.3]

Using edge-triggered notification

- Normally **employed with nonblocking I/O**
  - Can't monitor "I/O level", so must do nonblocking I/O calls until no more I/O is possible
    - Otherwise: risk blocking when doing I/O
- **Beware of FD starvation**
  - Scenarios where responding to a busy FD leaves other ready FDs starved of attention
  - (Starvation scenarios can also occur with level-triggered notification)
  - See TLPI §63.4.6

Exercises

The altio/i_epoll.c program can be used to perform epoll monitoring and file I/O operations on the objects named in its command-line arguments. The program is interactive, and supports the following commands:

```
p [<timeout>]
        Do epoll_wait() with millisecond timeout (default: 0)
e <fd> [<flags>]
        Modify epoll settings of <fd>; <flags> can include:
        'r' - EPOLLIN
        'w' - EPOLLOUT
        'e' - EPOLLET
        'o' - EPOLLONESHOT
        If no flags are given, disable <fd> in the interest list
r <fd> <size>
        Blocking read of <size> bytes from <fd>
R <fd> <size>
        Nonblocking read of <size> bytes from <fd>
w <fd> <size> [<char>]
        Blocking write of <size> bytes to <fd>; <char> is character
        to write (default: 'x')
W <fd> <size> [<char>]
        Nonblocking write of <size> bytes to <fd>
```

Each command-line argument has the form <path>[:<flags>] (to open a file) or s%<host>%<port>[:<flags>] (to connect a socket to a specified host/port). <flags> is as described above, and defaults to "r". (If testing with sockets, you will find the command ncat -l <port> useful, in order to create a server that you can connect to.)

Exercises

The following exercises are intended to demonstrate the effect of the EPOLLET and EPOLLONESHOT flags.

1. In separate windows, create two FIFOs and use *cat* to write to each FIFO:

```
mkfifo x
cat > x
```

```
mkfifo y
cat > y
```

2. Run the *i_epoll* program, using it to monitor both FIFOs for reading, specifying the EPOLLET flag for the FIFO y; note the file descriptor numbers used for each FIFO:

```
./i_epoll x:r y:re
```

3. Type some input into both *cat* commands, and then use the "p" command to perform an *epoll_wait()*:

```
i_epoll> p
```

You should find that both file descriptors report as ready for reading (EPOLLIN).

Exercises

4. Enter the "p" command again. You should find that only the FIFO `x` reports `EPOLLIN`. (`y` does not report as ready because no new input has appeared on the FIFO.)
5. Type some input into the *cat* command that is writing to the FIFO `y`, and once more use the "p" command to perform an *epoll_wait()*. You should find that both FIFOs report `EPOLLIN`. (`y` reports as ready again because new input has appeared on the FIFO.)
6. Switch the monitoring of the FIFO `y` to use `EPOLLET` and `EPOLLONESHOT` with the command `"e <fd> reo"`.
7. Type some input into the FIFO `y`, and then use the "p" command to perform an *epoll_wait()*. You should find that both `x` and `y` report `EPOLLIN`.
8. Type some more input into the FIFO `y`, and again use the "p" command to perform an *epoll_wait()*. You should find that `y` does not report as ready (because, after it reported as ready in the previous step, it was disabled in the interest list by `EPOLLONESHOT`).
9. Reenable the FIFO `y` in the interest list using the command `"e <fd> re"` and again use the "p" command to perform an *epoll_wait()*. You should find that `y` reports `EPOLLIN`.
10. Try any other experiments you might think of!

Outline

epoll and duplication of file descriptors

- Entries in *epoll* interest list are associated with **combination** of file descriptor (FD) and open file description (OFD)
  - Not just FD alone
- ⚠ Lifetime of interest list entry == lifetime of OFD
  - Can provide some surprises when FDs are duplicated...

[TLPI §63.4.4]

epoll and duplication of file descriptors

- Suppose that *fd* in code below refers to a socket...

```
ev.events = EPOLLIN;
ev.data.fd = fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
newfd = dup(fd);
close(fd);
epoll_wait(epfd, ...);
```

- What happens if some input now arrives on the socket?
- *epoll_wait()* might still return events for registration of *fd*
  - Because open file description is still alive and present in interest list
    - OFD is kept alive by *newfd*
  - ⚠ Notifications return data given in registration of *fd*!!

epoll and duplication of file descriptors

- Analogous scenarios possible with *fork()*:

```
ev.events = EPOLLIN;
ev.data.fd = fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
if (fork() == 0) {
    /* Child continues, does not close 'fd' */
} else {
    close(fd);
    epoll_wait(epfd, ...);
}
```


epoll and duplication of file descriptors

- ⚠ Can't `EPOLL_CTL_DEL` *fd* after *close()*
  - ⇒ `EBADF`
- Must either:
  - Close duplicate FDs
    - ⚠ But you may not know about duplicate if it was created by a library function that used *dup()* or *fork()*
  - Or manually `EPOLL_CTL_DEL` *fd* **before** closing it

Outline

Event-loop programming

- *select()*/*poll()*/*epoll* lend themselves to **event-loop programming**
  - I.e., program just sits in a loop, waiting on events from file descriptors
    - Monitored FDs can include pipes, sockets, terminals, devices, inotify, and even other epoll instances
  - Events are processed synchronously
- Problem: some other events of interest are not (traditionally) synchronous/aren't monitorable via FDs:
  - Signals
  - Timer expirations
  - IPC synchronization events
    - E.g., semaphore is incremented (*sem_post()*)
  - Process state transitions
    - E.g., child process termination

Event-loop programming

- Linux solution: turn those other events into file descriptors:
  - Signals ⇒ *signalfd()*
  - Timers ⇒ *timerfd* (*timerfd_create()*, *timerfd_settime()*, ...)
  - Synchronization ⇒ *eventfd()*
  - Process state transitions ⇒ “PID” file descriptors
    - PID FDs are returned by *pidfd_open()*, *clone()*/*clone3()* `CLONE_PIDFD`
    - Currently (Linux 5.4), only process-termination transitions are notified
- Monitor FDs produced by those mechanisms along with other FDs, using *select()*/*poll()*/*epoll*

Notes

Linux/UNIX System Programming Fundamentals

Wrapup

Michael Kerrisk, man7.org © 2020

mtk@man7.org

NDC TechTown
August 2020

Outline

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

The Linux *man-pages* project

- Your participation in the course indirectly supports work on the Linux *man-pages* project... Thanks!
- The Linux *man-pages* project:
  - Man pages for the Linux user-space APIs (system calls, etc.) and C library APIs
    - Principally, pages in sections 2 and 3
  - http://www.kernel.org/doc/man-pages/
  - Patches and contributions welcome
    - https://www.kernel.org/doc/man-pages/contributing.html
  - Latest man pages online at http://man7.org/linux/man-pages/

Subscribe to LWN!

- LWN (http://lwn.net/) == core online publication of Linux kernel and "plumbing" development community
- Latest info on kernel development and new features
  - + related topics (development process, legal issues facing FOSS, etc.)
- Almost entirely subscriber-funded
  - Please consider subscribing: https://lwn.net/subscribe/Info
- They also seek guest authors
  - https://lwn.net/op/AuthorGuide.lwn

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
  - Creating and using shared libraries
  - System programming for Linux containers
  - Linux/UNIX network programming
  - POSIX Threads programming
  - Subsets/combinations of the above
  - Further courses to be announced: http://man7.org/training/

Thanks!

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

PGP fingerprint: 4096R/3A35CE5E

http://man7.org/training/