Background topics

System Programming Essentials

Michael Kerrisk, man7.org © 2019

mtk@man7.org

December 2019

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 FD is to be inherited by child process that also writes to 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)
  - -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("couldn't write whole buffer");
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 the system calls *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 the file named in its command-line argument. 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` command-line option, which appends (`O_APPEND`) output to the file if it already exists, rather than truncating the file. To test the program, use the test target in the `Makefile`: `make tee_test`

   Some hints:
   - 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

Process credentials

Each process has several user IDs (UIDs) and group IDs (GIDs):

- **Real** UID + real GID
- **Effective** UID + effective GID
- **Saved** set-user-ID + saved set-group-ID
- **Supplementary GIDs**
- *credentials(7)* man page

Real UID and GID

- Real UID and GID identify who a process belongs to
- Login shell sets these from fields 3 and 4 in `/etc/passwd`
- New process inherits copies of its parent's real IDs

Effective UID and GID

- Determine permissions for performing various operations (in conjunction with supplementary GIDs)
- E.g., files have:
  1. an associated user and group, and
  2. RWX permissions for user/group/other
- New process inherits parent's effective IDs
- Effective UID 0 is special: normally has all privileges
  - AKA *root* or superuser
- Normally, effective IDs have same values as corresponding real IDs
- Can differ when set-user-ID or set-group-ID program is executed (later...)

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

- Used in set-UID and set-GID programs
- More later...

Supplementary GIDs

- Additional groups to which a process belongs
- Used in conjunction with effective GID to check group permissions on files and other objects
- Login shell obtains IDs from `/etc/group`
- New process inherits IDs from parent

APIs for retrieving process credentials

- *getuid()*, *getgid()*: get real IDs
- *geteuid()*, *getegid()*: get effective IDs
- *getresuid(&ruid, &euid, &suid)*,
  getresgid(&rgid, &egid, &sgid): retrieve real, effective and saved set IDs
- *getgroups(size, grouplist)*: retrieve supplementary GID list

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")

[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`)
  - Input became available on a file descriptor (`SIGIO`)
  - 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*: additional signals to block during handler invocation
- *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

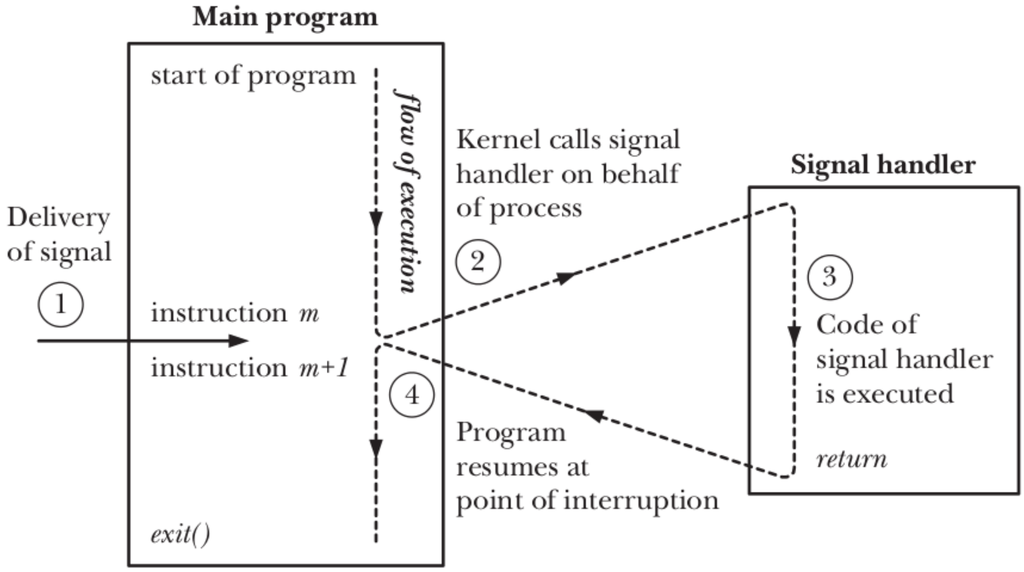
- Programmer-defined function
- Called with one integer argument: number of signal
  - ⇒ handler installed for multiple signals can differentiate...
- Returns `void`

```
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` (snippet)

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 */
}
```

Outline

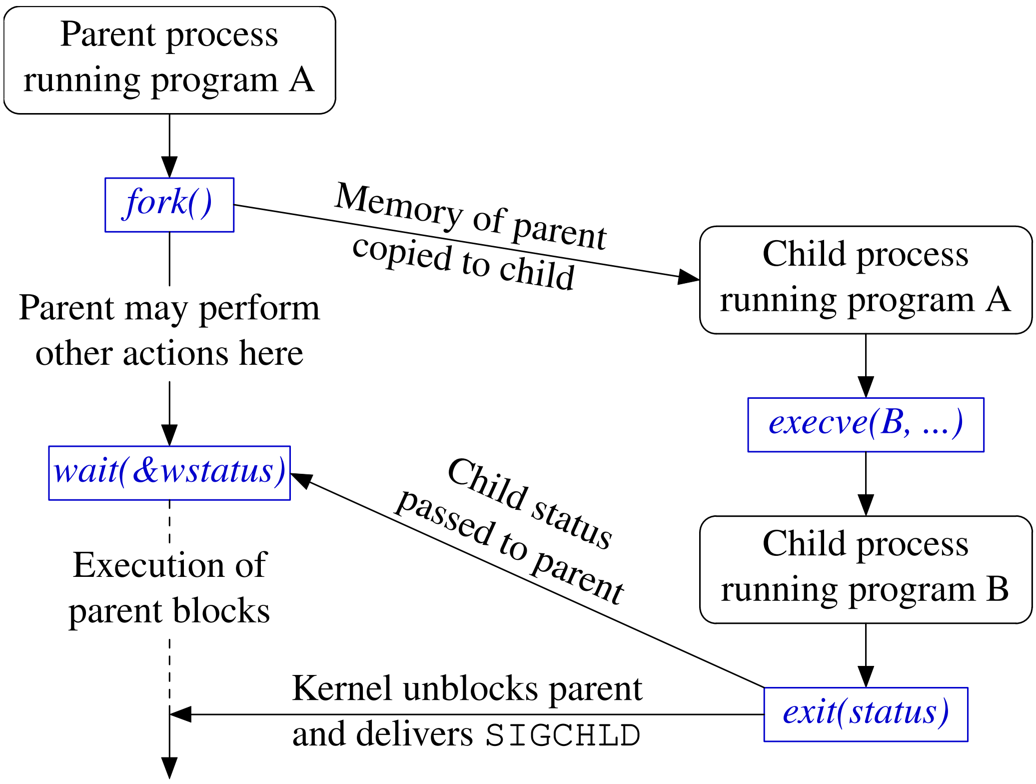
Creating processes and executing programs

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(3)* 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. The function *alarm(secs)* establishes a timer that expires after the specified number of seconds, and notifies the process by delivery of a `SIGALRM` signal. Write a program that performs the following steps in order to determine if a child process inherits alarm timers from the parent [template: `procexec/ex.inherit_alarm.c`]:
   - Establishes a `SIGALRM` handler that prints the process's PID.
   - Starts an alarm timer that expires after two seconds.
   - Creates a child process.
   - Both processes then loop 8 times, displaying the process PID and sleeping for half a second (use *usleep()*).

Outline

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:

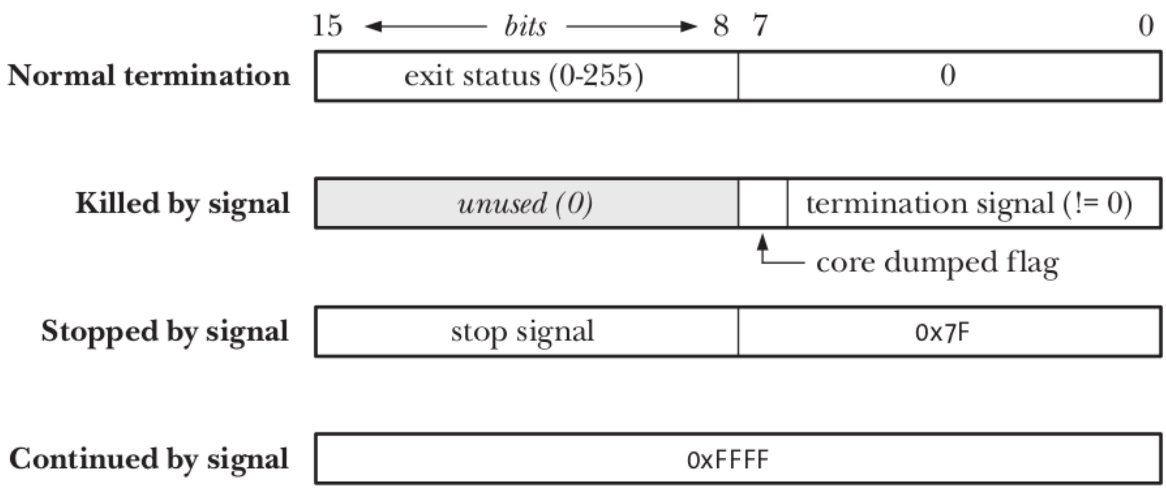
```
for (;;) {
    childPid = waitpid(-1, NULL, 0);
    if (childPid == -1) {
        if (errno == ECHILD) {
            printf("No more children!\n");
            break;
        } else {        /* Unexpected error */
            errExit("wait");
        }
    }

    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

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

```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 ([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]*.

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