ASCII code: lowercase and uppercase differ by the 5th bit (0 for uppercase, 1 for lower case—values differ by 32)

Run strace to find syscalls of a program

For an exit (or exit\_group) syscall: the program stops at that point

Page size: 4096 bytes

ldd <executable> shows which dynamic libraries an executable uses

objdump -T <lib> shows the symbols (often just function names) that are in the library

An update to the dynamic library can easily cause an executable using it to crash, e.g. if a struct with multiple public fields is reordered in the dynamic libraries but the executable still uses the old offsets

* Note: this is okay if the dynamic library never exposes the fields of a struct.

valgrind: trace malloc and free

*longopts* is a pointer to the first element of an array of *struct option* declared in *<*[*getopt.h*](https://linux.die.net/include/getopt.h)*>* as

struct option {

const char \*name;

int has\_arg;

int \*flag;

int val;

};

The meanings of the different fields are:

*name*

is the name of the long option.

*has\_arg*

is: **no\_argument** (or 0) if the option does not take an argument; **required\_argument** (or 1) if the option requires an argument; or **optional\_argument** (or 2) if the option takes an optional argument.

*flag*

specifies how results are returned for a long option. If *flag* is NULL, then **getopt\_long**() returns *val*. (For example, the calling program may set *val* to the equivalent short option character.) Otherwise, **getopt\_long**() returns 0, and *flag* points to a variable which is set to *val* if the option is found, but left unchanged if the option is not found.

*val*

is the value to return, or to load into the variable pointed to by *flag*.

The last element of the array has to be filled with zeros.

If *longindex* is not NULL, it points to a variable which is set to the index of the long option relative to *longopts*.

**getopt\_long\_only**() is like **getopt\_long**(), but '-' as well as "--" can indicate a long option. If an option that starts with '-' (not "--") doesn't match a long option, but does match a short option, it is parsed as a short option instead.

Permission: 0644 (0 denotes octal number instead of decimal number)

* 4’s place: r
* 2’s place: w
* 1’s place: x
* 0644: 6 for user, 5 for group, o for others, a for all (equiv. to ugo)

**ssize\_t read(int** *fd***, void \****buf***, size\_t** *count***);**

fd should always be 0

**read**() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

The **dup**() system call creates a copy of the file descriptor *oldfd*, using the lowest-numbered unused file descriptor for the new descriptor.

* opening the new input/output file
* closing the file descriptor (0, 1, 2) to be replaced
* duplicate the new input/output file to the (newly vacated) file descriptor to be replaced
* close the (now redundant) file descriptor on to which that file was originally opened

|  |  |
| --- | --- |
| **Input redirection**  int ifd = open(newfile, O\_RDONLY);  if (ifd >= 0) {  close(0);  dup(ifd);  close(ifd);  } | **Output redirection**  int ofd = creat(newfile, 0666);  if (ofd >= 0) {  close(1);  dup(ofd);  close(ofd);  } |

The **dup2**() system call performs the same task as **dup**(), but instead of using the lowest-numbered unused file descriptor, it uses the file description number specified in *newfd*. If the file descriptor *newfd* was previously open, it is silently closed before being reused.

* The steps of closing and reusing the file descriptor *newfd* are performed *atomically*.
  + No race condition—better than dup()!

We can safely use signal() as here it calls sigaction because we’re making a single-threaded program

printf uses some global variables to control things and access the heap, so if there is a signal, and there is a printf inside the signal handler, then the buffer may be damaged because it’s not safe to call it within that thread again

getopt will commute all non-option ARGV elements to the end after parsing

*// print any command line arguments that are not options.*

**if** (optind < argc)

{}

## April 8

fork starts a new process which is a copy of the one that calls it, while exec replaces the current process image with another (different) one.

[TERMIOS(3)](http://man7.org/linux/man-pages/man3/termios.3.html)

int tcsetattr(int *fildes*, int *optional\_actions*, const struct termios \**termios\_p*);

* CSANOW: The change occurs immediately.
* TCSADRAIN: The change occurs after all output written to the file descriptor has been transmitted. This action should be used when changing parameters that affect output.
* TCSAFLUSH: The change occurs after all output written to the file descriptor has been transmitted, and all input so far received but not read is discarded before the change is made.

Returns successfully if it was able to perform any of the requested actions, even if some of the requested actions could not be performed. It sets all the attributes that are supported for the device as requested, and leaves all attributes not supported unchanged. If no part of the request can be completed, tcsetattr() returns -1 and sets errno to EINVAL. A subsequent call to [tcgetattr()](https://www.mkssoftware.com/docs/man3/tcgetattr.3.asp) returns the actual state of the terminal device (reflecting both the changes made and not made in the previous tcsetattr() call). The tcsetattr() function does not change the values in the termios structure whether or not it actually accepts them.

int tcgetattr(int *fildes*, struct termios \**termios\_p*);

The *tcgetattr*() function shall get the parameters associated with the terminal referred to by *fildes* and store them in the **termios** structure referenced by *termios\_p*. The *fildes* argument is an open file descriptor associated with a terminal.

The *termios\_p* argument is a pointer to a **termios** structure.

The *tcgetattr*() operation is allowed from any process.

## April 10

Core: physical concept—one core can run a process at one time

Multiple cores: can run multiple processes simultaneously

Dereferencing NULL pointer in Java: nullptr exception

read() is synchronous

What are the major types of systems:

* Time sharing
* Batch
* Real-time

## Discussion: April 12

Canonical mode: the operating system provides input editing facilities

1. Queue
2. Echo
3. Erase (backspace), kill

Non-canonical mode:

Everything is read as input, including ^C

[Non-canonical input](https://www.gnu.org/software/libc/manual/html_node/Noncanonical-Input.html#Noncanonical-Input)

The MIN and TIME values interact to determine the criterion for when read should return; their precise meanings depend on which of them are nonzero. There are four possible cases:

1. Both TIME and MIN are nonzero.

In this case, TIME specifies how long to wait after each input character to see if more input arrives. After the first character received, read keeps waiting until either MIN bytes have arrived in all, or TIME elapses with no further input.

read always blocks until the first character arrives, even if TIME elapses first.  read can return more than MIN characters if more than MIN happen to be in the queue.

1. Both MIN and TIME are zero.

In this case, read always returns immediately with as many characters as are available in the queue, up to the number requested. If no input is immediately available, read returns a value of zero.

1. MIN is zero but TIME has a nonzero value.

In this case, read waits for time TIME for input to become available; the availability of a single byte is enough to satisfy the read request and cause read to return. When it returns, it returns as many characters as are available, up to the number requested. If no input is available before the timer expires, read returns a value of zero.

1. TIME is zero but MIN has a nonzero value.

In this case, read waits until at least MIN bytes are available in the queue. At that time, read returns as many characters as are available, up to the number requested.  read can return more than MIN characters if more than MIN happen to be in the queue.

For our purposes, 1 and 4 are the same. 1 is the default, so we don’t have to change anything

[**wait(2) and waitpid(2)**](http://man7.org/linux/man-pages/man2/waitpid.2.html)

The **wait**() *system call* suspends execution of the calling thread untilone of its children terminates. The call *wait(&wstatus)* is equivalent to:

waitpid(-1, &wstatus, 0);

The **waitpid**() system call suspends execution of the calling thread until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

* < -1 meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
* -1 meaning wait for any child process.
* meaning wait for any child process whose process group ID is equal to that of the calling process.
* > 0 meaning wait for the child whose process ID is equal to the value of *pid*.

Return values:

**wait**(): on success, returns the process ID of the terminated child; on error, -1 is returned.

**waitpid**(): on success, returns the process ID of the child whose state has changed. On error, -1 is returned.

**WIFEXITED(***wstatus***)**: returns true if the child terminated normally, that is, by calling [exit(3)](http://man7.org/linux/man-pages/man3/exit.3.html) or [\_exit(2)](http://man7.org/linux/man-pages/man2/_exit.2.html), or by returning from main().

**WEXITSTATUS(***wstatus***)**: returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to [exit(3)](http://man7.org/linux/man-pages/man3/exit.3.html) or [\_exit(2)](http://man7.org/linux/man-pages/man2/_exit.2.html) or as the argument for a return statement in main(). This macro should be employed only if **WIFEXITED** returned true.

What happens to the signal handler after execvp?

* The signal handler is a user-space code, and after execvp everything in the .code section will be replaced, so the signal handler won’t work anymore

[**pipe(2)**](http://man7.org/linux/man-pages/man2/pipe.2.html)

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

**pipe**() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe. *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

pipe puts values into the pipefd array!

Parent process: close the write end

Child process: close(p1[]); execvp();

* Close 4 file descriptors: 2 belongs to pipe, p1’s write end, and p2’s read end
* Dup2(stdin) //IOE redirection
* Dup2(stdout)
* Dup2(stderr)
* Close remaining pipes: p1 read, p2 write
* We only want 0,1,2 in the end since bash only knows about these 3

[**poll(2)**](http://man7.org/linux/man-pages/man2/poll.2.html)

**poll**() performs a similar task to [select(2)](http://man7.org/linux/man-pages/man2/select.2.html): it waits for one of a set of file descriptors to become ready to perform I/O.

The set of file descriptors to be monitored is specified in the *fds* argument, which is an array of structures of the following form:

struct pollfd {

int fd; /\* file descriptor \*/

short events; /\* requested events \*/

short revents; /\* returned events \*/

};

The caller should specify the number of items in the *fds* array in *nfds*.

[**kill(2)**](http://man7.org/linux/man-pages/man2/kill.2.html)

The **kill**() system call can be used to send any signal to any process group or process.

* If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*.
* If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process.
* If *pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.
* If *pid* is less than -1, then *sig* is sent to every process in the process group whose ID is *-pid*.

Return value: On success (at least one signal was sent), zero is returned. On error, -1 is returned, and [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) is set appropriately.

[**close(2)**](http://man7.org/linux/man-pages/man2/close.2.html)

**close**() closes a file descriptor, so that it no longer refers to any file and may be reused.

[**send(2)**](http://man7.org/linux/man-pages/man2/send.2.html)

**ssize\_t send(int** *sockfd***, const void \****buf***, size\_t** *len***, int** *flags***);**

The **send**() call may be used only when the socket is in a *connected* state (so that the intended recipient is known). The only difference between **send**() and [write(2)](http://man7.org/linux/man-pages/man2/write.2.html) is the presence of *flags*. With a zero *flags* argument, **send**() is equivalent to [write(2)](http://man7.org/linux/man-pages/man2/write.2.html).

The argument *sockfd* is the file descriptor of the sending socket.

[**recv(2)**](http://man7.org/linux/man-pages/man2/recv.2.html)

The **recv**(), **recvfrom**(), and **recvmsg**() calls are used to receive messages from a socket.

The only difference between **recv**() and [read(2)](http://man7.org/linux/man-pages/man2/read.2.html) is the presence of *flags*. With a zero *flags* argument, **recv**() is generally equivalent to [read(2)](http://man7.org/linux/man-pages/man2/read.2.html) (but see NOTES).

SIGPIPE

If a process tries to write to a pipe that has no reader, it will be sent the SIGPIPE signal from the kernel. This is imperative when more than two processes are involved in a pipeline.

[**flock(2)**](http://man7.org/linux/man-pages/man2/flock.2.html)

**int flock(int** *fd***, int** *operation***);**

Apply or remove an advisory lock on the open file specified by *fd*.

The argument *operation* is one of the following:

* **LOCK\_SH** Place a shared lock. More than one process may hold a shared lock for a given file at a given time.
* **LOCK\_EX** Place an exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.
* **LOCK\_UN** Remove an existing lock held by this process.

A call to **flock**() may block if an incompatible lock is held by another process. To make a nonblocking request, include **LOCK\_NB** (by ORing) with any of the above operations.

[**lockf(3)**](http://man7.org/linux/man-pages/man3/lockf.3.html)

**int lockf(int** *fd***, int** *cmd***, off\_t** *len***);**

Apply, test or remove a POSIX lock **on a section of an open file**. The file is specified by *fd*, a file descriptor open for writing, the action by *cmd*, and the section consists of byte positions *pos*..*pos*+*len*-1 if *len* is positive, and *pos*-*len*..*pos*-1 if *len* is negative, where *pos* is the current file position, and if *len* is zero, the section extends from the current file position to infinity, encompassing the present and future end-of-file positions. In allcases, the section may extend past current end-of-file.

Valid operations are given below:

* **F\_LOCK** Set an exclusive lock on the specified section of the file. If (part of) this section is already locked, the call blocks until the previous lock is released. If this section overlaps an earlier locked section, both are merged. File locks are released as soon as the process holding the locks closes some file descriptor for the file. A child process does not inherit these locks.
* **F\_TLOCK** Same as **F\_LOCK** but the call never blocks and returns an error instead if the file is already locked.
* **F\_ULOCK** Unlock the indicated section of the file. This may cause a locked section to be split into two locked sections.
* **F\_TEST** Test the lock: return 0 if the specified section is unlocked or locked by this process; return -1, set [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) to **EAGAIN** (**EACCES** on some other systems), if another process holds a lock.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OS@boot | Hardware | |  | | | | | | | | |
| Initialize trap table |  | |  | | | | | | | | |
|  | Remember addresses of: syscall , timer, illegal mem-access, illegal instruction handlers | | |  | | | | | | | |
| Start interrupt timer |  | |  | | | | | | | | |
|  | Start timer, interrupt after X ms | |  | | | | | | | | |
| Initialize process table & free list |  | |  | | | | | | | | |
| OS@run (kernel mode) | Hardware | | Program (user mode) | | | | | | | | |
| Start process A: allocate entry in process table, allocate memory for process, set base/bounds registers  Return-from-trap (into A) | | | | | | | |  | |  | |
|  | Restore registers of A, move to user mode, jump to A’s (initial) PC | | | | | | | |  | | |
|  |  | | Process A runs: fetch instruction | | | | | | | | |
|  | Translate virtual address & perform fetch | |  | | | | | | | | |
|  |  | | Execute instruction | | | | | | | | |
|  | If explicit load/store: ensure address is in-bounds; translate address and perform load/store | | | |  | | | | | | |
|  | Timer interrupt; move to kernel mode, jump to interrupt handler | | | | | | | | | |  |
| Handle the trap: call switch routine  Save regs(A) to proc-struct(A) (in PCB), incl. b/b  Restore regs(B) from proc-struct(B), incl b/b  Return-from-trap (into B) | |  |  | | | | | | | | |
|  | Restore registers of B, move to user mode, jump to B’s PC | | | | |  | | | | | |
|  |  | | Process B runs: bad load | | | | | | | | |
|  | Load is out-of-bounds; move to kernel mode, jump to trap handler | |  | | | | | | | | |
| Handle the trap, decide to terminate process B, deallocate B’s memory, free B’s entry in process table | | | | | | |  | | | |  |