Recitation 9: Tshlab + VM

Instructor: TAs
29 October 2018
Outline

- Labs
- Signals
- IO
- Virtual Memory
tshlab and malloclаб

- tshlab due Tuesday

- malloclаб is released immediately after
  - Start early
  - Do the checkpoint first, don’t immediately go for the final
  - Expect a recitation next week
    - Working for several hours will improve the value significantly
Signals

- Parent process sends SIGINT to a child process. What is the behavior of the child?
- What is the default?
- What else could the child do?
More Signals

- Parent process sends SIGKILL to a child process. What is the behavior of the child?

- What is the default?

- What else could the child do?
Sending Signals

- Parent sends SIGKILL to a child process.

...  

```c
pid_t pid = ...; // child pid
kill(pid, SIGKILL);
// At this point, what could have
// happened to the child process?
```
Blocking Signals

- The shell is currently running its handler for SIGCHLD.

- What signals can it receive?
- What signals can it not receive (i.e., blocked)?
Errno

- Included from <errno.h>
- Global integer variable – usually 0
- When a system call fails (usually indicated by returning -1), it also will set errno to a value describing what went wrong
- Example: let’s assume there is no “foo.txt” in our path
  ```c
  int fd = open(“foo.txt”, O_RDONLY);
  if(fd < 0) printf(“%d\n”, errno);
  ```
  The code above will print 2 – in the man pages, we can see that 2 is ENOENT “No such file or directory”
- In shell lab, your signal handlers must preserve errno
IO functions

Needed for tshlab

- int open(const char *pathname, int flags);
  - Some important flags:
    - O_CREAT – creates file if needed, opens for read/write
    - O_RDWR – opens for read/write
    - O_RDONLY – opens for read only
    - Various permission modes

- int close(int fd);

- int dup2(int oldfd, int newfd);

Needed for life

- ssize_t read(int fd, void *buf, size_t count);
- ssize_t write(int fd, const void *buf, size_t count);
- off_t lseek(int fd, off_t offset, int whence);
More on open

- int open(const char *pathname, int flags, mode_t mode);

- For *flags*, you can pass a bitwise-OR of one or more flags

- Three kinds of flags (we only discuss the important ones)
  - Access modes (one of them must be included):
    - O_RDONLY, O_WRONLY, O_RDWR
  - File creation flags:
    - O_CREAT, O_TRUNC, etc.
  - File status flags
Access mode flags and file creation flags

- **O_RDONLY**
  - Open the file read-only.

- **O_WRONLY**
  - Open the file write-only.

- **O_RDWR**
  - Open the file read/write.

- **O_CREAT**
  - If the provided *pathname* does not exist, create it as a regular file.

- **O_TRUNC**
  - If the file already exists and if the access mode allows writing (i.e. is **O_RDWR** or **O_WRONLY**), then the file will be truncated to length 0.
More on open

- `int open(const char *pathname, int flags, mode_t mode);`
- For `mode`, you can pass a bitwise-OR of one or more constants
- Specifies, when creating a file, what permission the file will be created with
- Only useful when `flags` contain O_CREAT (or O_TMPFILE)
Every file and directory has permission information

You’ve seen it before

- `ls -l` prints the permissions for each file/directory like:
  - `-rw-r--r-- ...  drwxr-xr-x ...`
- `chmod` changes the permissions for files/directories
  - `$ chmod -R 777 /`

There are read (R), write (W) and executable (X) permissions for user (USR), group (GRP) and other (OTH)
Specify permissions in open()

<table>
<thead>
<tr>
<th></th>
<th>Read (R)</th>
<th>Write (W)</th>
<th>Executable (X)</th>
<th>All (RWX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>User (USR)</td>
<td>S_IRUSR</td>
<td>S_IWUSR</td>
<td>S_IXUSR</td>
<td>S_IRWXU</td>
</tr>
<tr>
<td>Group (GRP)</td>
<td>S_IRGRP</td>
<td>S_IWGRP</td>
<td>S_IXGRP</td>
<td>S_IRWXG</td>
</tr>
<tr>
<td>Other (OTH)</td>
<td>S_IROTH</td>
<td>S_IWOTH</td>
<td>S_IXOTH</td>
<td>S_IRWXO</td>
</tr>
</tbody>
</table>

- These constants can be bitwise-OR’d and passed to the third argument of open()
- What does S_IRWXG | S_IXUSR | S_IXOTH mean?
- How to create a file which everyone can read from but only the user can write to it or execute it?
File descriptors

<table>
<thead>
<tr>
<th>fd</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
</table>

stdin, stdout, stderr are opened automatically and closed by normal termination or exit()
open(“foo.txt”)

<table>
<thead>
<tr>
<th>fd</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>open file table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard input</td>
</tr>
<tr>
<td>Standard output</td>
</tr>
<tr>
<td>Standard error</td>
</tr>
<tr>
<td>foo.txt</td>
</tr>
</tbody>
</table>
open("foo.txt")

Each call to open() creates a new open file description

<table>
<thead>
<tr>
<th>fd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

inode table

- foo.txt

open file table

- Standard input
- Standard output
- Standard error
- foo.txt
- foo.txt
**dup2(STDOUT_FILENO, 3)**

<table>
<thead>
<tr>
<th>fd</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>

**open file table**
- Standard input
- Standard output
- Standard error
- foo.txt

Closed silently
IO and Fork()

- File descriptor management can be tricky.
- How many file descriptors are open in the parent process at the indicated point?
- How many does each child have open at the call to execve?

```c
int main(int argc, char** argv)
{
    int i;
    for (i = 0; i < 4; i++)
    {
        int fd = open("foo", O_RDONLY);
        pid_t pid = fork();
        if (pid == 0)
        {
            int ofd = open("bar", O_RDONLY);
            execve(...);
        }
    }
    // How many file descriptors are open in the parent?
```
Redirecting IO

- File descriptors can be directed to identify different open files.

```c
int main(int argc, char** argv) {
    int i;
    for (i = 0; i < 4; i++)
    {
        int fd = open("foo", O_RDONLY);
        pid_t pid = fork();
        if (pid == 0)
        {
            int ofd = open("bar", O_WRONLY);
            dup2(fd, STDIN_FILENO);
            dup2(ofd, STDOUT_FILENO);
            execve(...);
        }
    }
    // How many file descriptors are open in the parent?
}
```
Redirecting IO

- At the two points (A and B) in main, how many file descriptors are open?

```c
int main(int argc, char** argv)
{
    int i, fd;
    fd = open("foo", O_WRONLY);
    dup2(fd, STDOUT_FILENO);
    // Point A
    close(fd);
    // Point B
    ...
```
Memory Access

- The processor tries to write to a memory address.
- List different steps that are required to complete this operation.
Memory Access

- The processor tries to write to a memory address.
- List different steps that are required to complete this operation. (non exhaustive list)
  - Virtual to physical address conversion (TLB lookup)
  - TLB miss
  - Page fault, page loaded from disk
  - TLB updated, check permissions
  - L1 Cache miss (and L2 ... and)
  - Request sent to memory
  - Memory sends data to processor
  - Cache updated
Address Translation with TLB

- Translate 0x15213, given the contents of the TLB and the first 32 entries of the page table below.

- **1MB Virtual Memory**
  - 256KB Physical Memory
  - 4KB page size

### TLB Contents

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>05</td>
<td>05</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>3F</td>
<td>15</td>
<td>05</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>0F</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>1E</td>
<td>10</td>
<td>0F</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1F</td>
<td>0F</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1F</td>
<td>1F</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>03</td>
<td>2B</td>
<td>1</td>
</tr>
<tr>
<td>1D</td>
<td>23</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Page Table Contents

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>0B</td>
<td>0</td>
</tr>
<tr>
<td>04</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>06</td>
<td>0F</td>
<td>1</td>
</tr>
<tr>
<td>07</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>08</td>
<td>1C</td>
<td>0</td>
</tr>
<tr>
<td>09</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>0B</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>0C</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>0E</td>
<td>0C</td>
<td>0</td>
</tr>
<tr>
<td>0F</td>
<td>2B</td>
<td>1</td>
</tr>
</tbody>
</table>

2-way set associative
If you get stuck on tshlab

- Read the writeup!
- Do manual unit testing before runtrace and sdriver!
- Post private questions on piazza!

- Read the man pages on the syscalls.
  - Especially the error conditions
  - What errors should terminate the shell?
  - What errors should be reported?
man wait

Taken from http://man7.org/linux/man-pages/man2/wait.2.html

WAIT(2)                   Linux Programmer's Manual                  WAIT(2)

NAME

wait, waitpid, waitid - wait for process to change state

SYNOPSIS

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *wstatus);

pid_t waitpid(pid_t pid, int *wstatus, int options);

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

/* This is the glibc and POSIX interface; see 
NOTES for information on the raw system call. */
man pages (probably) cover all you need

- What arguments does the function take?
  - read SYNOPSIS

- What does the function do?
  - read DESCRIPTION

- What does the function return?
  - read RETURN VALUE

- What errors can the function fail with?
  - read ERRORS

- Is there anything I should watch out for?
  - read NOTES

- Different categories for man page entries with the same name

- Looking up man pages online is not an academic integrity violation
Function arguments

- Should I do `dup2(old, new)` or `dup2(new, old)`?
- Read the man page:

```bash
$ man dup2
```

SYNOPSIS

```c
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```
Function behavior

How should I write my format string when I need to print a long double in octals with precision 5 and zero-padded?

Read the man page:

$ man printf

DESCRIPTION

Flag characters

The character % is followed by zero or more of the following flags:

#  The value should be converted...
0  The value should be zero padded...
-  The converted value is to be left adjusted...
  ' ' (a space) A blank should be left before...
+  A sign (+ or -) should always ...
Function return

- What does waitpid() return with and without WNOHANG?
- Read the man page:

$ man waitpid

RETURN VALUE

waitpid(): on success, returns the process ID of the child whose state has changed; if WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.

Each of these calls sets errno to an appropriate value in the case of an error.
Potential errors

- How should I check `waitpid` for errors?
- Read the man page:

$ man waitpid

ERRORS

ECHILD (for `waitpid()` or `waitid()`) The process specified by `pid` (`waitpid()`) or `idtype` and `id` (`waitid()`) does not exist or is not a child of the calling process. (This can happen for one's own child if the action for `SIGCHLD` is set to `SIG_IGN`. See also the Linux Notes section about threads.)

EINTR WNOHANG was not set and an unblocked signal or a `SIGCHLD` was caught; see `signal(7)`.

EINVAL The `options` argument was invalid.
Get advice from the developers

- I sprintf from a string into itself, is this okay?
- Read the man page:

$ man sprintf

NOTES

Some programs imprudently rely on code such as the following

    sprintf(buf, "%s some further text", buf);

to append text to buf. However, the standards explicitly note that
the results are undefined if source and destination buffers overlap
when calling sprintf(), snprintf(), vsprintf(), and vsnprintf().
Depending on the version of gcc(1) used, and the compiler options
employed, calls such as the above will not produce the expected
results.

The glibc implementation of the functions snprintf() and vsnprintf()
conforms to the C99 standard, that is, behaves as described above,
since glibc version 2.1. Until glibc 2.0.6, they would return -1
when the output was truncated.