Final project!

https://stevens.netmeister.org/631/f19-sish.html
In a nutshell: the "what"

$ ls /bin
[  csh    ed    ls    pwd    sleep
cat    date    expr    mkdir    rcmd    stty
chio    dd    hostname    mt    rcp    sync
chmod    df    kill    mv    rm    systrace
cp    domainname    ksh    pax    rmdir    tar
cpio    echo    ln    ps    sh    test
$

See also:

https://www.cs.stevens.edu/~jschauma/631/#source-code
$ grep "(int" /usr/include/sys/socket.h
int accept(int, struct sockaddr *__restrict, socklen_t *__restrict);
int bind(int, const struct sockaddr *, socklen_t);
int connect(int, const struct sockaddr *, socklen_t);
int getsockopt(int, int, int, void *__restrict, socklen_t *__restrict);
int listen(int, int);
ssize_t recv(int, void *, size_t, int);
ssize_t recvfrom(int, void *__restrict, size_t, int,
ssize_t recvmsg(int, struct msghdr *, int);
ssize_t send(int, const void *, size_t, int);
ssize_t sendto(int, const void *,
ssize_t sendmsg(int, const struct msghdr *, int);
int setsockopt(int, int, int, const void *, socklen_t);
int socket(int, int, int);
int socketpair(int, int, int, int *);
$
In a nutshell: the "what"

- gain an understanding of the UNIX operating systems
- gain (systems) programming experience
- understand fundamental OS concepts (with focus on UNIX family):
  - multi-user concepts
  - basic and advanced I/O
  - process relationships
  - interprocess communication
  - basic network programming using a client/server model
In a nutshell

The "why":

- understanding how UNIX works gives you insights in other OS concepts
- system level programming experience is invaluable as it forms the basis for most other programming and even *use* of the system
- system level programming in C helps you understand general programming concepts
- most higher level programming languages (eventually) call (or implement themselves) standard C library functions
UNIX Basics: Architecture
UNIX Basics: Pipelines

Say "Thank you, Douglas McIlroy!"

http://is.gd/vGH09J
Important ANSI C Features, Error Handling

- Important ANSI C Features:
  - function prototypes
  - generic pointers (`void *`)
  - abstract data types (e.g. `pid_t`, `size_t`)

- Error Handling:
  - meaningful return values
  - `errno` variable
  - look up constant error values via two functions:

```c
#include <string.h>
char *strerror(int errnum)  // Returns: pointer to message string

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

Lecture 14: Review
December 2, 2019
Lecture 02

File I/O, File Sharing
File Descriptors

- A file descriptor (or file handle) is a small, non-negative integer which identifies a file to the kernel.
- Traditionally, stdin, stdout and stderr are 0, 1 and 2 respectively.
- Relying on “magic numbers” is Bad™. Use STDIN_FILENO, STDOUT_FILENO and STDERR_FILENO.
Standard I/O

Basic File I/O: almost all UNIX file I/O can be performed using these five functions:

- open(2)
- close(2)
- lseek(2)
- read(2)
- write(2)

Processes may want to share resources. This requires us to look at:

- atomicity of these operations
- file sharing
- manipulation of file descriptors
open(2)

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

Returns: file descriptor if OK, -1 on error

**oflag** must be one (and only one) of:

- **O_RDONLY** – Open for reading only
- **O_WRONLY** – Open for writing only
- **O_RDWR** – Open for reading and writing

and may be OR’d with any of these:

- **O_APPEND** – Append to end of file for each write
- **O_CREAT** – Create the file if it doesn’t exist. Requires *mode* argument
- **O_EXCL** – Generate error if **O_CREAT** and file already exists. (atomic)
- **O_TRUNC** – If file exists and successfully open in **O_WRONLY** or **O_RDWR**, make length = 0
- **O_NOCTTY** – If pathname refers to a terminal device, do not allocate the device as a controlling terminal
- **O_NONBLOCK** – If pathname refers to a FIFO, block special, or char special, set nonblocking mode (open and I/O)
- **O_SYNC** – Each write waits for physical I/O to complete
open(2) variants

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

Returns: file descriptor if OK, -1 on error

On some platforms `oflag` may also be one of:

- `O_EXEC` – Open for execute only
- `O_SEARCH` – Open for search only (applies to directories)

and may be OR’d with any of these:

- `O_DIRECTORY` – If path resolves to a non-directory file, fail and set errno to ENOTDIR.
- `O_DSYNC` – Wait for physical I/O for data, except file attributes
- `O_RSYNC` – Block read operations on any pending writes.
- `O_PATH` – Obtain a file descriptor purely for fd-level operations. (Linux >2.6.36 only)

`openat(2)` is used to handle relative pathnames from different working directories in an atomic fashion.
close(2)

```
#include <unistd.h>

int close(int fd);
```

Returns: 0 if OK, -1 on error

- Closing a file descriptor releases any record locks on that file (more on that in future lectures)
- File descriptors not explicitly closed are closed by the kernel when the process terminates.
read(2)

```c
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes);
```

Returns: number of bytes read, 0 if end of file, -1 on error

There can be several cases where `read` returns less than the number of bytes requested:

- EOF reached before requested number of bytes have been read
- Reading from a terminal device, one "line" read at a time
- Reading from a network, buffering can cause delays in arrival of data
- Record-oriented devices (magtape) may return data one record at a time
- Interruption by a signal

`read` begins reading at the current offset, and increments the offset by the number of bytes actually read.
write(2)

```c
#include <unistd.h>

ssize_t write(int filedes, void *buff, size_t nbytes);

Returns: number of bytes written if OK, -1 on error
```

- **write** returns **nbytes** or an error has occurred (disk full, file size limit exceeded, ...)
- for regular files, **write** begins writing at the current offset (unless **O_APPEND** has been specified, in which case the offset is first set to the end of the file)
- after the write, the offset is adjusted by the number of bytes actually written
lseek(2)

```c
#include <sys/types.h>
#include <fcntl.h>

off_t lseek(int filedes, off_t offset, int whence);
```

Returns: new file offset if OK, -1 on error

The value of whence determines how offset is used:
- SEEK_SET bytes from the beginning of the file
- SEEK_CUR bytes from the current file position
- SEEK_END bytes from the end of the file

“Weird” things you can do using lseek(2):
- seek to a negative offset
- seek 0 bytes from the current position
- seek past the end of the file
File Sharing

Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files. (See: Stevens, pp 70 ff)

- each process table entry has a table of file descriptors, which contain
  - the file descriptor flags (ie FD_CLOEXEC, see fcntl(2))
  - a pointer to a file table entry

- the kernel maintains a file table; each entry contains
  - file status flags (O_APPEND, O_SYNC, O_RDONLY, etc.)
  - current offset
  - pointer to a vnode table entry

- a vnode structure contains
  - vnode information
  - inode information (such as current file size)
File Sharing
File Sharing
File Sharing

Knowing this, here's what happens with each of the calls we discussed earlier:

- after each `write` completes, the current file offset in the file table entry is incremented. (If current_file_offset > current_file_size, change current file size in i-node table entry.)

- If file was opened `O_APPEND` set corresponding flag in file status flags in file table. For each `write`, current file offset is first set to current file size from the i-node entry.

- `lseek` simply adjusts current file offset in file table entry

- to `lseek` to the end of a file, just copy current file size into current file offset.
dup(2) and dup2(2)

```
#include <unistd.h>
int dup(int oldd);
int dup2(int oldd, int newd);
```

Both return new file descriptor if OK, -1 on error.

An existing file descriptor can be duplicated with `dup(2)` or duplicated to a particular file descriptor value with `dup2(2)`. As with `open(2)`, `dup(2)` returns the lowest numbered unused file descriptor.

Note the difference in scope of the file *descriptor* flags and the file *status* flags compared to distinct processes.
Lecture 03

Files and Directories
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *path, struct stat *sb);
in lstat(const char *path, struct stat *sb);
in fstat(int fd, struct stat *sb);

Returns: 0 if OK, -1 on error

All these functions return extended attributes about the referenced file (in the case of symbolic links, lstat(2) returns attributes of the link, others return stats of the referenced file).

struct stat {
    dev_t st_dev;    /* device number (filesystem) */
    ino_t st_ino;    /* i-node number (serial number) */
    mode_t st_mode;  /* file type & mode (permissions) */
    dev_t st_rdev;   /* device number for special files */
    nlink_t st_nlink; /* number of links */
    uid_t st_uid;    /* user ID of owner */
    gid_t st_gid;    /* group ID of owner */
    off_t st_size;   /* size in bytes, for regular files */
    time_t st_atime; /* time of last access */
    time_t st_mtime; /* time of last modification */
    time_t st_ctime; /* time of last file status change */
    long st_blocks;  /* number of 512-byte* blocks allocated */
    long st_blksize; /* best I/O block size */
};
struct stat: st_mode

The st_mode field of the struct stat encodes the type of file:

- **regular** – most common, interpretation of data is up to application
- **directory** – contains names of other files and pointer to information on those files. Any process can read, only kernel can write.
- **character special** – used for certain types of devices
- **block special** – used for disk devices (typically). All devices are either character or block special.
- **FIFO** – used for interprocess communication (sometimes called named pipe)
- **socket** – used for network communication and non-network communication (same host).
- **symbolic link** – Points to another file.

Find out more in <sys/stat.h>.
struct stat: `st_mode`, `st_uid` and `st_gid`

Every process has six or more IDs associated with it:

<table>
<thead>
<tr>
<th>ID Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>real user ID, real group ID</td>
<td>who we really are</td>
</tr>
<tr>
<td>effective user ID, effective group ID, supplementary group IDs</td>
<td>used for file access permission checks</td>
</tr>
<tr>
<td>saved set-user-ID, saved set-group-ID</td>
<td>saved by <code>exec</code> functions</td>
</tr>
</tbody>
</table>

Whenever a file is `setuid`, set the effective user ID to `st_uid`. Whenever a file is `setgid`, set the effective group ID to `st_gid`. `st_uid` and `st_gid` always specify the owner and group owner of a file, regardless of whether it is setuid/setgid.
**setuid(2)/seteuid(2)**

```c
#include <unistd.h>

int seteuid(uid_t uid);
int setuid(uid_t euid);

Returns: 0 if OK, -1 on error

uid_t geteuid(void);
uid_t getuid(void);

Returns: uid_t; no error
```

See also: `getresuid(2)` (if `_GNU_SOURCE`)

*setuid* programs should only use elevated privileges *when needed*. Note: after using `setuid(2)`, you *cannot* regain elevated privileges. This is by design!
struct stat: st_mode

st_mode also encodes the file access permissions (S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH). Uses of the permissions are summarized as follows:

- To open a file, need execute permission on each directory component of the path
- To open a file with O_RDONLY or O_RDWR, need read permission
- To open a file with O_WRONLY or O_RDWR, need write permission
- To use O_TRUNC, must have write permission
- To create a new file, must have write+execute permission for the directory
- To delete a file, need write+execute on directory, file doesn’t matter
- To execute a file (via exec family), need execute permission
struct stat: st_mode

Which permission set to use is determined (in order listed):

1. If effective-uid == 0, grant access
2. If effective-uid == st_uid
   2.1. if appropriate user permission bit is set, grant access
   2.2. else, deny access
3. If effective-gid == st_gid
   3.1. if appropriate group permission bit is set, grant access
   3.2. else, deny access
4. If appropriate other permission bit is set, grant access, else deny access
struct stat: st_mode

Ownership of new files and directories:

- st_uid = effective-uid
- st_gid = ...either:
  - effective-gid of process
  - gid of directory in which it is being created
umask(2) sets the file creation mode mask. Any bits that are on in the file creation mask are turned off in the file’s mode.

Important because a user can set a default umask. If a program needs to be able to insure certain permissions on a file, it may need to turn off (or modify) the umask, which affects only the current process.
`chmod(2), lchmod(2) and fchmod(2)`

```c
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int lchmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Returns: 0 if OK, -1 on error

Changes the permission bits on the file. Must be either superuser or `effective uid == st.uid`. `mode` can be any of the bits from our discussion of `st_mode` as well as:

- `S_ISUID` – setuid
- `S_ISGID` – setgid
- `S_ISVTX` – sticky bit (aka “saved text”)
- `S_IRWXU` – user read, write and execute
- `S_IRWXG` – group read, write and execute
- `S_IRWXO` – other read, write and execute
chown(2), lchown(2) and fchown(2)

```c
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Returns: 0 if OK, -1 on error

Changes st_uid and st_gid for a file. For BSD, must be superuser. Some SVR4’s let users chown files they own. POSIX.1 allows either depending on _POSIX_CHOWN_RESTRICTED (a kernel constant).

owner or group can be -1 to indicate that it should remain the same. Non-superusers can change the st_gid field if both:

- effective-user ID == st_uid and
- owner == file’s user ID and group == effective-group ID (or one of the supplementary group IDs)

chown and friends clear all setuid or setgid bits.
Lecture 04

File Systems, System Data Files, Time & Date
File Systems

- a disk can be divided into logical *partitions*
- each logical *partition* may be further divided into *file systems* containing *cylinder groups*
- each *cylinder group* contains a list of *inodes (i-list)* as well as the actual *directory- and data blocks*
File Systems

- A disk can be divided into logical partitions.
- Each logical partition may be further divided into file systems containing cylinder groups.
- Each cylinder group contains a list of inodes (i-list) as well as the actual directory- and data blocks.
File Systems

- A disk can be divided into logical *partitions*.
- Each logical *partition* may be further divided into *file systems* containing *cylinder groups*.
- Each *cylinder group* contains a list of *inodes* (*i-list*) as well as the actual *directory- and data blocks*.
- A directory entry is really just a *hard link* mapping a “filename” to an inode.
File Systems

- A disk can be divided into logical partitions.
- Each logical partition may be further divided into file systems containing cylinder groups.
- Each cylinder group contains a list of inodes (i-list) as well as the actual directory- and data blocks.
- A directory entry is really just a hard link mapping a “filename” to an inode.
- You can have many such mappings to the same file.
Directories

- directories are special "files" containing hardlinks
Directories

- directories are special “files” containing hardlinks
- each directory contains at least two entries:
  - . (this directory)
  - .. (the parent directory)
Directories

- directories are special “files” containing hardlinks
- each directory contains at least two entries:
  - . (this directory)
  - .. (the parent directory)
- the link count (st_nlink) of a directory is at least 2
Inodes

- The *inode* contains most of information found in the *stat* structure.
- Every *inode* has a *link count* (*st_nlink*): it shows how many “things” point to this inode. Only if this *link count* is 0 (and no process has the file open) are the *data blocks* freed.
- *Inode* number in a directory entry must point to an *inode* on the same file system (no hardlinks across filesystems)
Inodes

- the *inode* contains most of information found in the *stat* structure.
- every *inode* has a *link count* (*st_nlink*): it shows how many "things" point to this inode. Only if this *link count* is 0 (and no process has the file open) are the *data blocks* freed.
- *inode* number in a directory entry must point to an *inode* on the same file system (no hardlinks across filesystems)
- to move a file within a single filesystem, we can just "move" the directory entry (actually done by creating a new entry, and deleting the old one).
Inodes

- the _inode_ contains most of information found in the _stat_ structure.
- every _inode_ has a _link count_ (\(\text{st}\_\text{nlink}\)): it shows how many “things” point to this inode. Only if this _link count_ is 0 (and no process has the file open) are the _data blocks_ freed.
- _inode_ number in a directory entry must point to an _inode_ on the same file system (no hardlinks across filesystems)
- to move a file within a single filesystem, we can just “move” the directory entry (actually done by creating a new entry, and deleting the old one).
Inodes

- The *inode* contains most of information found in the *stat* structure.
- Every *inode* has a *link count* (st_nlink): it shows how many “things” point to this inode. Only if this *link count* is 0 (and no process has the file open) are the *data blocks* freed.
- *Inode* number in a directory entry must point to an *inode* on the same file system (no hardlinks across filesystems).
- To move a file within a single filesystem, we can just “move” the directory entry (actually done by creating a new entry and deleting the old one).
**link(2) and unlink(2)**

```c
#include <unistd.h>
int link(const char *name1, const char *name2);
```

Returns: 0 if OK, -1 on error

- Creates a link to an existing file (hard link).
- POSIX.1 allows links to cross filesystems, most implementations (SVR4, BSD) don’t.
- only uid(0) can create links to directories (loops in filesystem are bad)

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

Returns: 0 if OK, -1 on error

- removes directory entry and decrements link count of file
- if file link count == 0, free data blocks associated with file (unless processes have the file open)
rename(2)

```c
#include <stdio.h>
int rename(const char *from, const char *to);
```

Returns: 0 if OK, -1 on error

If `oldname` refers to a file:
- if `newname` exists and it is not a directory, it’s removed and `oldname` is renamed `newname`
- if `newname` exists and it is a directory, an error results
- must have w+x perms for the directories containing `old/newname`

If `oldname` refers to a directory:
- if `newname` exists and is an empty directory (contains only . and ..), it is removed; `oldname` is renamed `newname`
- if `newname` exists and is a file, an error results
- if `oldname` is a prefix of `newname` an error results
- must have w+x perms for the directories containing `old/newname`
Symbolic Links

```c
#include <unistd.h>

int symlink(const char *name1, const char *name2);

Returns: 0 if OK, -1 on error
```

- file whose "data" is a path to another file
- anyone can create symlinks to directories or files
- certain functions dereference the link, others operate on the link

```c
#include <unistd.h>

int readlink(const char *path, char *buf, size_t bufsize);

Returns: number of bytes placed into buffer if OK, -1 on error
```

This function combines the actions of open, read, and close. Note: `buf` is not NUL terminated.
#include <sys/types.h>

int utimes(const char *path, const struct timeval times[2]);
int lutimes(const char *path, const struct timeval times[2]);
int futimes(int fd, const struct timeval times[2]);

Returns: 0 if OK, -1 on error

If `times` is NULL, access time and modification time are set to the current time (must be owner of file or have write permission). If `times` is non-NULL, then times are set according to the `timeval` struct array. For this, you must be the owner of the file (write permission not enough).

Note that `st_ctime` is set to the current time in both cases.

For the effect of various functions on the access, modification and changes-status times see Stevens, p. 117.

Note: some systems implement `lutimes(3)` (library call) via `utimes(2)` syscalls.
mkdir(2) and rmdir(2)

```c
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

Returns: 0 if OK, -1 on error

Creates a new, empty (except for . and .. entries) directory. Access permissions specified by mode and restricted by the umask(2) of the calling process.

```c
#include <unistd.h>

int rmdir(const char *path);
```

Returns: 0 if OK, -1 on error

If the link count is 0 (after this call), and no other process has the directory open, directory is removed. Directory must be empty (only . and .. remaining)
Reading Directories

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *filename);
Returns: pointer if OK, NULL on error

struct dirent *readdir(DIR *dp);
Returns: pointer if OK, NULL at end of dir or on error

void rewinddir(DIR *dp);

int closedir(DIR *dp);
Returns: 0 if OK, -1 on error
```

- read by anyone with read permission on the directory
- format of directory is implementation dependent (always use readdir and friends)

`opendir`, `readdir` and `closedir` should be familiar from our small `ls` clone. `rewinddir` resets an open directory to the beginning so `readdir` will again return the first entry.

For directory traversal, consider `fts` (3) (not available on all UNIX versions).
Moving around directories

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

Returns: buf if OK, NULL on error
```

Get the kernel’s idea of our process’s current working directory.

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

Returns: 0 if OK, -1 on error
```

Allows a process to change its current working directory. Note that `chdir` and `fchdir` affect only the current process.
Password File

Called a *user database* by POSIX and usually found in `/etc/passwd`, the password file contains the following fields:

<table>
<thead>
<tr>
<th>Description</th>
<th>struct passwd member</th>
<th>POSIX.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>char *pw_name</td>
<td>X</td>
</tr>
<tr>
<td>encrypted passwd</td>
<td>char *pw_passwd</td>
<td></td>
</tr>
<tr>
<td>numerical user id</td>
<td>uid_t pw_uid</td>
<td>X</td>
</tr>
<tr>
<td>numerical group id</td>
<td>gid_t pw_gid</td>
<td>X</td>
</tr>
<tr>
<td>comment field</td>
<td>char *pw_gecos</td>
<td></td>
</tr>
<tr>
<td>initial working directory</td>
<td>char *pw_dir</td>
<td>X</td>
</tr>
<tr>
<td>initial shell</td>
<td>char *pw_shell</td>
<td>X</td>
</tr>
</tbody>
</table>

Encrypted password field is a one-way hash of the users password. (Always maps to 13 characters from [a-zA-Z0-9./].)

Some fields can be empty:

- password empty implies *no password*
- shell empty implies `/bin/sh`
Password File

```c
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);

Returns: pointer if OK, NULL on error
```

```c
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwent(void);
void setpwent(void);
void endpwent(void);

Returns: pointer if OK, NULL on error
```

- **getpwent** returns next password entry in file each time it’s called, no order
- **setpwent** rewinds to ”beginning” of entries
- **endpwent** closes the file(s)

See also: **getspnam(3)/getspent(3)** (where available)
**Group File**

Called a *group database* by POSIX and usually found in `/etc/group`, the group file contains the following fields:

<table>
<thead>
<tr>
<th>Description</th>
<th>struct group member</th>
<th>POSIX.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupname</td>
<td>char *gr_name</td>
<td>X</td>
</tr>
<tr>
<td>encrypted passwd</td>
<td>char *gr_passwd</td>
<td></td>
</tr>
<tr>
<td>numerical group id</td>
<td>uid_t gr_uid</td>
<td>X</td>
</tr>
<tr>
<td>array of pointers to user names</td>
<td>char **gr_mem</td>
<td>X</td>
</tr>
</tbody>
</table>

The `gr_mem` array is terminated by a NULL pointer.
Group File

```c
#include <sys/types.h>
#include <grp.h>

struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);

Returns: pointer if OK, NULL on error
```

These allow us to look up an entry given a user’s group name or numerical GID. What if we need to go through the group file entry by entry? Nothing in POSIX.1, but SVR4 and BSD give us:

```c
#include <sys/types.h>
#include <grp.h>

struct group *getgrent(void);

Returns: pointer if OK, NULL on error

void setgrent(void);
void endgrent(void);
```

- `getgrent` returns next group entry in file each time it’s called, no order
- `setgrent` rewinds to ”beginning” of entries
- `endgrent` closes the file(s)
Supplementary Groups and other data files

```
#include <sys/types.h>
#include <unistd.h>

int getgroups(int gidsetsize, gid_t *grouplist);

Returns: returns number of suppl. groups if OK, -1 on error
```

Note: if `gidsetsize == 0`, `getgroups(2)` returns number of groups without modifying `grouplist`. 
System Identification

```
#include <sys/utsname.h>
int uname(struct utsname *name);

Returns: nonnegative value if OK, -1 on error
```

- Pass a pointer to a `utsname` struct. This struct contains fields like oopsys name, version, release, architecture, etc.
- This function used by the `uname(1)` command (try `uname -a`)
- Not that the size of the fields in the `utsname` struct may not be large enough to id a host on a network

To get just a hostname that will identify you on a TCP/IP network, use the Berkeley-dervied:

```
#include <unistd.h>
int gethostname(char *name, int namelen);

Returns: 0 if OK, -1 on error
```
#include <time.h>
time_t time(time_t *tloc);
Returns: value of time if OK, -1 on error

- Time is kept in UTC
- Time conversions (timezone, daylight savings time) handled "automatically"
- Time and date kept in a single quantity (time_t)

We can break this time_t value into its components with either of the following:

#include <time.h>

struct tm *gmtime(const time_t *calptr);
struct tm *localtime(const time_t *calptr);
Returns: pointer to broken down time
Time and Date

```
#include <time.h>
time_t mktime(struct tm *tmptr);
Returns: calendar time if OK, -1 on error
```

`localtime(3)` takes into account daylight savings time and the `TZ` environment variable. The `mktime(3)` function operates in the reverse direction. To output human readable results, use:

```
#include <time.h>
char *asctime(const struct tm *tmptr);
char *ctime(const struct tm *tmptr);
Returns: pointer to NULL terminated string
```

Lastly, there is a `printf(3)` like function for times:

```
#include <time.h>
size_t strftime(char *buf, size_t maxsize, const char *restricted format, const struct tm *timeptr);
Returns: number of characters stored in array if room, else 0
```
Time and Date
Lecture 05

UNIX development tools
Software Development Tools

UNIX Userland is an IDE – essential tools that follow the paradigm of “Do one thing, and do it right” can be combined.

The most important tools are:

- $EDITOR
- the compiler toolchain
- gdb(1) – debugging your code
- make(1) – project build management, maintain program dependencies
- diff(1) and patch(1) – report and apply differences between files
- cvs(1), svn(1), git(1) etc. – distributed project management, version control
EDITOR

Know your $EDITOR. Core functionality:

- syntax highlighting
- efficient keyboard maneuvering
- setting markers, using buffers
- copy, yank, fold e.g. blocks
- search and replace
- window splitting
- autocompletion
- jump to definition / manual page
- applying external commands and filters
Examples given using `vim(1)`.

Look-ups:
- `find /usr/src -name '*[ch]' -print | exec ctags -f ~/.ctgs`
- `echo "set tags+=~/.ctags" >> ~/.vimrc`
- `Ctrl+], Ctrl+t` – jump to definition and back
- `K` – jump to manual page
- `Ctrl+N` – autocomplete
Examples given using `vim(1)`.

Integration with compiler, debugger, `make(1)` etc.

```bash
vim welcome.c
:make
Ctrl+]
:cnex
...
```

Finally, two of your most powerful Unix IDE integrations are a terminal multiplexer (e.g. `screen(1)` or `tmux(1)`) and copious use of Ctrl+Z (i.e., the shell’s job control mechanisms).
Compilers

A compiler translates *source code* from a high-level programming language into *machine code* for a given architecture by performing a number of steps:

- lexical analysis
- preprocessing
- parsing
- semantic analysis
- code optimization
- code generation
- assembly
- linking
Preprocessing

The compiler chain or driver usually performs preprocessing (e.g. via cpp(1)), compilation (cc(1)), assembly (as(1)) and linking (ld(1)).

```
$ cd compilechain
$ cat hello.c
$ man cpp
$ cpp hello.c hello.i
$ file hello.i
$ man cc
$ cc -v -E hello.c > hello.i
$ more hello.i
$ cc -v -DFOOD="Avocado" -E hello.c > hello.i.2
$ diff -bu hello.i hello.i.2
```
Compilation

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ more hello.i
$ cc -v -S hello.i > hello.s
$ file hello.s
$ more hello.s
```
Assembly

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ as -o hello.o hello.s 
$ file hello.o 
$ cc -v -c hello.s 
$ objdump -d hello.o 
[...]
```
Linking

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ ld hello.o
[...]
$ ld hello.o -lc
[...]
$ cc -v hello.o
[...]
$ ld -dynamic-linker /usr/libexec/ld.elf_so
    /usr/lib/crt0.o /usr/lib/crti.o /usr/lib/crtbegin.o
    hello.o -lc /usr/lib/crtend.o /usr/lib/crtin.o
$ file a.out
$ ./a.out
```
The purpose of a debugger such as `gdb(1)` is to allow you to see what is going on “inside” another program while it executes – or what another program was doing at the moment it crashed. `gdb` allows you to:

- make your program stop on specified conditions (for example by setting `breakpoints`)
- examine what has happened, when your program has stopped (by looking at the `backtrace`, inspecting the value of certain variables)
- inspect control flow (for example by `stepping` through the program)

Other interesting things you can do:

- examine stack frames: `info frame`, `info locals`, `info args`
- examine memory: `x`
- examine assembly: `disassemble func`
$ cc simple-ls.c
$ ./a.out ~/testdir
Memory fault (core dumped)
$ gdb ./a.out
(gdb) run ~/testdir

Program received signal SIGSEGV, Segmentation fault.
0x00000000000400cc7 in main (argc=2, argv=0x7f7fffa71978) at simple-ls-stat.c:48
warning: Source file is more recent than executable.
48     printf("%s (%s)\n", dirp->d_name, pwd->pw_name);

(gdb) bt
(gdb) frame 0
(gdb) li
(gdb) print pwd
**make(1)**

`make(1)` is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs selective rebuilds following a *dependency graph*
- allows simplification of rules through use of *macros* and *suffixes*, some of which are internally defined
- different versions of `make(1)` (BSD make, GNU make, Sys V make, ...) may differ (among other things) in
  - variable assignment and expansion/substitution
  - including other files
  - flow control (for-loops, conditionals etc.)
**diff(1)** and **patch(1)**

**diff(1):**
- compares files line by line
- output may be used to automatically edit a file
- can produce human “readable” output as well as diff entire directory structures
- output called a *patch*
**diff(1) and patch(1)**

**patch(1):**
- applies a `diff(1)` file (aka *patch*) to an original
- may back up original file
- may guess correct format
- ignores leading or trailing “garbage”
- allows for reversing the patch
- may even correct context line numbers
Revision Control

Version control systems allow you to

- collaborate with others
- simultaneously work on a code base
- keep old versions of files
- keep a log of the who, when, what, and why of any changes
- perform release engineering by creating *branches*
Revision Control: Branching

Different strategies:

- trunk / master is fragile
  - trunk is work in progress, may not even compile
  - all work happens in trunk
  - releases are tagged on trunk, then branched

- trunk / master is stable
  - master is always stable
  - all work is done in branches (feature or bugfix)
  - feature branches are deleted after merge
  - releases are made automatically from master

You may combine these as release branching / feature branching / task branching.
Commit Messages

Commit messages are like comments: too often useless and misleading, but critical in understanding human thinking behind the code.

Commit messages should be full sentences in correct and properly formatted English.

Commit messages briefly summarize the what, but provide important historical context as to the how and, more importantly, why.

Commit messages SHOULD reference and integrate with ticket tracking systems.

See also:

- http://is.gd/Wd1LhA
- http://is.gd/CUtwhA
- http://is.gd/rPQj5E
Lecture 06

Process Environment, Process Control
Memory Layout of a C Program

memory-layout.c
The main function

```c
int main(int argc, char **argv);
```

- C program started by kernel (by one of the `exec` functions)
- special startup routine called by kernel which sets up things for `main` (or whatever entrypoint is defined)
- `argc` is a count of the number of command line arguments (including the command itself)
- `argv` is an array of pointers to the arguments
- it is guaranteed by both ANSI C and POSIX.1 that `argv[argc] == NULL`
Process Creation

$ cc -Wall entry.c
$ readelf -h a.out | more

ELF Header:

[...]
Entry point address: 0x400460
Start of program headers: 64 (bytes into file)
Start of section headers: 4432 (bytes into file)

$ objdump -d a.out

[...]
000000000000400460 <_start>:
400460: 31 ed xor %ebp,%ebp
400462: 49 89 d1 mov %rdx,%r9

[...]

$ http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html
Process Creation

$ cc -e foo entry.c
$ ./a.out
Foo for the win!
Memory fault
$ cc -e bar entry.c
$ ./a.out
bar rules!
$ echo $?  
1
$ cc entry.c
$ ./a.out
Hooray main!
$ echo $?  
13
$
Process Termination

There are 8 ways for a process to terminate.

Normal termination:
- return from main
- calling exit
- calling _exit (or _Exit)
- return of last thread from its start routine
- calling pthread_exit from last thread

Abnormal termination:
- calling abort
- terminated by a signal
- response of the last thread to a cancellation request
exit(3) and _exit(2)

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

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

- _exit and _Exit
  - return to the kernel immediately
  - _exit required by POSIX.1
  - _Exit required by ISO C99
  - synonymous on Unix
- exit does some cleanup and then returns
- both take integer argument, aka exit status
atexit(3)

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

- Registers a function with a signature of `void funcname(void)` to be called at exit
- Functions invoked in reverse order of registration
- Same function can be registered more than once
- Extremely useful for cleaning up open files, freeing certain resources, etc.

`exit-handlers.c`
Lifetime of a UNIX Process
Environment List

Environment variables are stored in a global array of pointers:

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

The list is null terminated.

These can also be accessed by:

```c
#include <stdlib.h>

char *getenv(const char *name);
int putenv(const char *string);
int setenv(const char *name, const char *value, int rewrite);
void unsetenv(const char *name);

int main(int argc, char **argv, char **anvp);
```
Memory Allocation

#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
void *alloca(size_t size);
void free(void *ptr);

- **malloc** – initial value is indeterminate.
- **calloc** – initial value set to all zeros.
- **realloc** – changes size of previously allocated area. Initial value of any additional space is indeterminate.
- **alloca** – allocates memory on stack
Process limits

$ ulimit -a

time (cpu-seconds) unlimited
file (blocks) unlimited
coredump (blocks) unlimited
data (kbytes) 262144
stack (kbytes) 2048
lockedmem (kbytes) 249913
memory (kbytes) 749740
nofiles (descriptors) 128
processes 160
vmemory (kbytes) unlimited
sbsize (bytes) unlimited
$

Lecture 14: Review

December 2, 2019
getrlimit(2) and setrlimit(2)

```c
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rtp);
int setrlimit(int resource, const struct rlimit *rtp);
```

Changing resource limits follows these rules:

- **a soft limit** can be changed by any process to a value less than or equal to its hard limit
- any process can lower its **hard limit** greater than or equal to its soft limit
- only superuser can raise **hard limits**
- changes are per process only (which is why `ulimit` is a shell built-in)
Process Identifiers

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

*Process IDs* are guaranteed to be unique and identify a particular executing process with a non-negative integer.

Certain processes have fixed, special identifiers. They are:

- **swapper**, process ID 0 – responsible for scheduling
- **init**, process ID 1 – bootstraps a Unix system, owns orphaned processes
- **pagedaemon**, process ID 2 – responsible for the VM system (some Unix systems)
fork(2)

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

fork(2) causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors.
- The child process’ resource utilizations are set to 0.

**Note:** no order of execution between child and parent is guaranteed!
fork(2)
fork(2)
fork(2)

$ cc -Wall forkflush.c
$ ./a.out
a write to stdout
before fork
pid = 12149, glob = 7, var = 89
pid = 12148, glob = 6, var = 88
$ ./a.out | cat
a write to stdout
before fork
pid = 12153, glob = 7, var = 89
before fork
pid = 12151, glob = 6, var = 88
$
The `exec()` family of functions are used to completely replace a running process with a new executable.

- if it has a `v` in its name, `argv`'s are a vector: `const * char argv[]`
- if it has an `l` in its name, `argv`'s are a list: `const char *arg0, ... /* (char *) 0 */`
- if it has an `e` in its name, it takes a `char * const envp[]` array of environment variables
- if it has a `p` in its name, it uses the `PATH` environment variable to search for the file
### wait(2) and waitpid(2)

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

pid_t wait(int *status);
pid_t waitpid(pid_t wpid, int *status, int options);
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage);
```

A parent that calls `wait(2)` or `waitpid(2)` can:

- block (if all of its children are still running)
- return immediately with the termination status of a child
- return immediately with an error
Lecture 07

Process Groups, Sessions, Signals
Login Process

Let’s revisit the process relationships for a login:

- kernel $\Rightarrow$ init(8)  # explicit creation
- init(8) $\Rightarrow$ getty(8)  # fork(2)
- getty(8) $\Rightarrow$ login(1)  # exec(3)
- login(1) $\Rightarrow$ $\$SHELL$  # exec(3)
- $\$SHELL$ $\Rightarrow$ ls(1)  # fork(2) + exec(3)
Login Process

init(8) # PID 1, PPID 0, EUID 0

getty(8) # PID N, PPID 1, EUID 0

login(1) # PID N, PPID 1, EUID 0

$SHELL # PID N, PPID 1, EUID U

ls(1) # PID M, PPID N, EUID U

pstree -hapun | more
Process Groups

```c
#include <unistd.h>

pid_t getpgrp(void);

pid_t getpgid(pid_t pid);

Returns: process group ID if OK, -1 otherwise
```

- In addition to having a PID, each process also belongs to a process group (collection of processes associated with the same job/terminal)
- Each process group has a unique process group ID
- Process group IDs (like PIDs) are positive integers and can be stored in a `pid_t` data type
- Each process group can have a process group leader
  - Leader identified by its process group ID == PID
  - Leader can create a new process group, create processes in the group
- A process can set its (or its children’s) process group using `setpgid(2)`
Process Groups

init ➔ login shell

$ proc1 | proc2 &
[1] 10306
$ proc3 | proc4 | proc5
Process Groups and Sessions

```
#include <unistd.h>

pid_t setsid(void);

Returns: process group ID if OK, -1 otherwise
```

A session is a collection of one or more process groups.

If the calling process is not a process group leader, this function creates a new session. Three things happen:

- the process becomes the session leader of this new session
- the process becomes the process group leader of a new process group
- the process has no controlling terminal
Process Groups

\textit{init} \rightarrow \textit{login shell}

$ \text{proc1 | proc2 &}$

[1] 10306

$ \text{proc3 | proc4 | proc5}$
Process Groups and Sessions

\( \text{init} \Rightarrow \text{login shell} \)

$ \text{proc1} \mid \text{proc2} \&$

[1] 10306

$ \text{proc3} \mid \text{proc4} \mid \text{proc5} $
Process Groups and Sessions

```
$ ps -o pid,ppid,pgid,sess,comm | ./cat1 | ./cat2

<table>
<thead>
<tr>
<th>PID</th>
<th>PPID</th>
<th>PGRP</th>
<th>SESS</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1989</td>
<td>949</td>
<td>7736</td>
<td>949</td>
<td>ps</td>
</tr>
<tr>
<td>1990</td>
<td>949</td>
<td>7736</td>
<td>949</td>
<td>cat1</td>
</tr>
<tr>
<td>1988</td>
<td>949</td>
<td>7736</td>
<td>949</td>
<td>cat2</td>
</tr>
<tr>
<td>949</td>
<td>21401</td>
<td>949</td>
<td>949</td>
<td>sh</td>
</tr>
</tbody>
</table>
```
Job Control

$ ps -o pid,ppid,pgid,sess,comm
  PID  PPID  PGRP  SESS  COMMAND
  24251 24250 24251 24251  ksh
  24620 24251 24620 24251  ps
$ echo $?  
  0
$
Job Control

```bash
$ dd if=/dev/zero of=/dev/null bs=512 count=2048000 >/dev/null 2>&1 &
[1] 24748
$ ps -o pid,ppid,pgrp,sess,comm
  PID  PPID  PGRP  SESS   COMMAND
24251 24250 24251 24251 ksh
24748 24251 24748 24251 dd
24750 24251 24750 24251 ps
$
[1] +  Done  dd if=/dev/zero of=/dev/null bs=512 count=2048000 >/dev/null 2>&1 &
$
Job Control

Diagram showing the relationship between login shell, background process group, foreground process group, terminal driver, and user at a terminal.
Job Control

$ cat >file
Input from terminal,
Output to terminal.
^D
$ cat file
Input from terminal,
Output to terminal.
$ cat >/dev/null
Input from terminal,
Output to /dev/null.
Waiting forever...
Or until we send an interrupt signal.
^C
$
Job Control

$ cat file &
[1] 2056
$ Input from terminal, Output to terminal.

[1] + Done  cat file &
$ stty tostop
$ cat file &
[1] 4655
$

[1] + Stopped(SIGTTOU)  cat file &
$ fg
cat file
Input from terminal, Output to terminal.
$
Signal Concepts

Signals are a way for a process to be notified of asynchronous events. Some examples:

- a timer you set has gone off (SIGALRM)
- some I/O you requested has occurred (SIGIO)
- a user resized the terminal "window" (SIGWINCH)
- a user disconnected from the system (SIGHUP)
- ...

See also: signal(2)/signal(3)/signal(7) (note: these man pages vary significantly across platforms!)
Signal Concepts

Besides the asynchronous events listed previously, there are many ways to generate a signal:

- terminal generated signals (user presses a key combination which causes the terminal driver to generate a signal)
- hardware exceptions (divide by 0, invalid memory references, etc)
- `kill(1)` allows a user to send any signal to any process (if the user is the owner or superuser)
- `kill(2)` (a system call, not the unix command) performs the same task
- software conditions (other side of a pipe no longer exists, urgent data has arrived on a network file descriptor, etc.)
**kill(2) and raise(3)**

```c
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int signo);
int raise(int signo);
```

- **`pid > 0`** – signal is sent to the process whose PID is `pid`
- **`pid == 0`** – signal is sent to all processes whose process group ID equals the process group ID of the sender
- **`pid == -1`** – POSIX.1 leaves this undefined, BSD defines it (see `kill(2)`)
Signal Concepts

Once we get a signal, we can do one of several things:

- Ignore it. (note: there are some signals which we CANNOT or SHOULD NOT ignore)
- Catch it. That is, have the kernel call a function which we define whenever the signal occurs.
- Accept the default. Have the kernel do whatever is defined as the default action for this signal
signal(3)

```c
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);

Returns: previous disposition of signal if OK, SIG_ERR otherwise
```

*func* can be:

- **SIG_IGN** which requests that we ignore the signal *signo*
- **SIG_DFL** which requests that we accept the default action for signal *signo*
- or the address of a function which should catch or handle a signal
Interrupted System Calls

Some system calls can block for long periods of time (or forever). These include things like:

- `read(2)`s from files that can block (pipes, networks, terminals)
- `write(2)` to the same sort of files
- `open(2)` of a device that waits until a condition occurs (for example, a modem)
- `pause(3)`, which purposefully puts a process to sleep until a signal occurs
- certain `ioctl(3)`s
- certain IPC functions

Catching a signal during execution of one of these calls traditionally led to the process being aborted with an `errno` return of `EINTR`. 
Lecture 08

Interprocess Communication
System V IPC

Three types of IPC originating from System V:

- Semaphores
- Shared Memory
- Message Queues

All three use *IPC structures*, referred to by an *identifier* and a *key*; all three are (necessarily) limited to communication between processes on one and the same host.

Since these structures are not known by name, special system calls (msgget(2), semop(2), shmat(2), etc.) and special userland commands (ipcrm(1), ipcs(1), etc.) are necessary.
POSIX Message Queues

`mq(3)` provides a real-time IPC interface similar to System V message queues. Notably:

- message queues are identified by a named identifier (no `ftok(3)` needed); may or may not be exposed in the file system (e.g. `/dev/mqueue`)
- `mq_send(3)` and `mq_receive(3)` allow both blocking and non-blocking calls
- `mq_send(3)` lets you specify a priority; equal priority messages are queued as a FIFO, but higher priority messages are inserted before those of a lower priority
- `mq(3)` provides an asynchronous notification mechanism: `mq_notify(3)`
Pipes: pipe(2)

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

Returns: 0 if OK, -1 otherwise

- oldest and most common form of UNIX IPC
- half-duplex (on some versions full-duplex)
- can only be used between processes that have a common ancestor
- can have multiple readers/writers (PIPE_BUF bytes are guaranteed to not be interleaved)

Behavior after closing one end:

- **read(2)** from a pipe whose write end has been closed returns 0 after all data has been read
- **write(2)** to a pipe whose read end has been closed generates SIGPIPE signal. If caught or ignored, write(2) returns an error and sets errno to EPIPE.
Pipes: `popen(3)` and `pclose(3)`

```c
#include <stdio.h>
FILE *popen(const char *cmd, const char *type);

Returns: file pointer if OK, NULL otherwise

int pclose(FILE *fp);

Returns: termination status cmd or -1 on error
```

- Historically implemented using unidirectional pipe (nowadays frequently implemented using sockets or full-duplex pipes).
- `type` one of “r” or “w” (or “r+” for bi-directional communication, if available).
- `cmd` passed to `/bin/sh -c`
FIFOs: `mkfifo(2)`

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

Returns: 0 if OK, -1 otherwise

- aka “named pipes”
- allows unrelated processes to communicate
- just a type of file – test for using `S_ISFIFO(st_mode)`
- `mode` same as for `open(2)`
- use regular I/O operations (ie `open(2)`, `read(2)`, `write(2)`, `unlink(2)` etc.)
- used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files
Lecture 09

Interprocess Communication II
Sockets: socketpair(2)

```
#include <sys/socket.h>
int socketpair(int d, int type, int protocol, int *sv);
```

The `socketpair(2)` call creates an unnamed pair of connected sockets in the specified domain `d`, of the specified `type`, and using the optionally specified `protocol`.

The descriptors used in referencing the new sockets are returned in `sv[0]` and `sv[1]`. The two sockets are indistinguishable.

This call is currently implemented only for the UNIX domain.
Sockets: `socket(2)`

```c
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Some of the currently supported domains are:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF_LOCAL</td>
<td>local (previously UNIX) domain protocols</td>
</tr>
<tr>
<td>PF_INET</td>
<td>ARPA Internet protocols</td>
</tr>
<tr>
<td>PF_INET6</td>
<td>ARPA IPv6 (Internet Protocol version 6) protocols</td>
</tr>
<tr>
<td>PF_ARP</td>
<td>RFC 826 Ethernet Address Resolution Protocol</td>
</tr>
</tbody>
</table>

Some of the currently defined types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCK_STREAM</td>
<td>sequenced, reliable, two-way connection based byte streams</td>
</tr>
<tr>
<td>SOCK_DGRAM</td>
<td>connectionless, unreliable messages of a fixed (typically small) maximum length access to internal network protocols and interfaces</td>
</tr>
<tr>
<td>SOCK_RAW</td>
<td></td>
</tr>
</tbody>
</table>

Lecture 14: Review  December 2, 2019
Sockets: Datagrams in the UNIX/LOCAL domain

- create socket using `socket(2)`
- attach to a socket using `bind(2)`
- binding a name in the UNIX domain creates a socket in the file system
- both processes need to agree on the name to use
- these files are only used for rendezvous, not for message delivery
  once a connection has been established
- sockets must be removed using `unlink(2)`
Sockets: Datagrams in the Internet Domain

Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed.

The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard value INADDR_ANY.

“well-known” ports (range 1 - 1023) only available to super-user
request any port by calling bind(2) with a port number of 0

determine used port number (or other information) using getsockname(2)

convert between network byteorder and host byteorder using htons(3) and ntohs(3) (which may be noops)
connections are asymmetrical: one process requests a connection, the other process accepts the request

- one socket is created for each accepted request
- mark socket as willing to accept connections using `listen(2)`
- pending connections are then `accept(2)`ed
- `accept(2)` will block if no connections are available
- `select(2)` to check if connection requests are pending
Lecture 10

HTTP; Code Reading
The Hypertext Transfer Protocol

HTTP is a request/response protocol:

1. client sends a request to the server
   - request method
   - URI
   - protocol version
   - request modifiers
   - client information

2. server responds
   - status line (including success or error code)
   - server information
   - entity metainformation
   - content
The Hypertext Transfer Protocol

Server status codes:

- **1xx** – Informational; Request received, continuing process
- **2xx** – Success; The action was successfully received, understood, and accepted
- **3xx** – Redirection; Further action must be taken in order to complete the request
- **4xx** – Client Error; The request contains bad syntax or cannot be fulfilled
- **5xx** – Server Error; The server failed to fulfill an apparently valid request
HTTP - more than just text

HTTP is a *Transfer Protocol* – serving *data*, not any specific text format.

- **Accept-Encoding** client header can specify different formats such as *gzip*, *Shared Dictionary Compression over HTTP (SDCH)* etc.

- **corresponding server headers**: **Content-Type** and **Content-Encoding**
HTTP - more than just static data

HTTP is a *Transfer Protocol* – what is transferred need not be static; resources may generate different data to return based on many variables.

- CGI – resource is *executed*, needs to generate appropriate response headers
- server-side scripting (ASP, PHP, Perl, ...)
- client-side scripting (JavaScript/ECMAScript/JScript,...)
- applications based on HTTP, using:
  - AJAX
  - RESTful services
  - JSON, XML, YAML to represent state and abstract information
Code Reading

Let's take a look at some sample implementations:

- **mathopd**: [http://www.mathopd.org/download.html](http://www.mathopd.org/download.html)
- **Null httpd**: [http://nullhttpd.sourceforge.net/httpd/](http://nullhttpd.sourceforge.net/httpd/)
- **muhttpd**: [http://inglorion.net/software/muhttpd/](http://inglorion.net/software/muhttpd/)

Walk us through the code:

- networking setup (`socket(2), bind(2), ...`)
- request handling
- header parsing
- CGI execution
Lecture 11

Dæmon Processes, Shared Libraries
Writing a daemon

- fork off the parent process
- change file mode mask (umask)
- create a unique Session ID (SID)
- change the current working directory to a safe place
- close (or redirect) standard file descriptors
- open any logs for writing
- enter actual daemon code
Dæmon conventions

- prevent against multiple instances via a *lockfile*
- allow for easy determination of PID via a *pidfile*
- configuration file convention `/etc/name.conf`
- include a system initialization script (for `/etc/rc.d/` or `/etc/init.d/`)
- re-read configuration file upon SIGHUP
- relay information via *event logging*
Linking and Loading

A linker takes multiple *object files*, resolves symbols to *e.g.*, addresses in *libraries* (possibly relocating them in the process), and produces an *executable*.
Executable and Linkable Format

```bash
$ hexdump -C a.out | head -2
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
00000010 02 00 3e 00 01 00 00 00 e0 07 40 00 00 00 00 00
$ readelf -h a.out
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2’s complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x4007e0
  ...
```
Linking and Loading

A loader copies a program into main memory, possibly invoking the *dynamic linker* or *run-time link editor* to find the right libraries, resolve addresses of symbols, and relocate them.
Executable and Linkable Format

Compilers produce, and linkers and loaders operate on *object files*. Just like other files, they have specific formats such as *e.g.*, assembler output (`a.out`), Common Object File Format (`COFF`), Mach-O, or ELF.

- **executable** – just what it sounds like (*e.g.*, `a.out`)
- **core** – virtual address space and register state of a process; debugging information (`a.out.core`)
- **relocatable file** – can be linked together with others to produce a shared library or an executable (*e.g.*, `foo.o`)
- **shared object file** – position independent code; used by the dynamic linker to create a process image (*e.g.*, `libfoo.so`)
Shared Libraries

What is a shared library, anyway?

- contains a set of callable C functions (i.e., implementation of function prototypes defined in `.h` header files)
- code is position-independent (i.e., code can be executed anywhere in memory)
- shared libraries can be loaded/unloaded at execution time or at will
- libraries may be static or dynamic

```
$ man 3 fprintf
$ grep " fprintf" /usr/include/stdio.h
```
Shared Libraries

How do shared libraries work?

- at *link time*, the linker resolves undefined symbols
- contents of object files and *static* libraries are pulled into the executable at link time
- contents of *dynamic* libraries are used to resolve symbols at *link time*, but loaded at *execution time* by the *dynamic linker*
- contents of *dynamic* libraries may be loaded at *any time* via explicit calls to the dynamic linking loader interface functions
Understanding object files

$ cc -Wall ldtest1.c ldtest2.c main.c
$ nm a.out

U _libc_init
00000000004007a0 T _start
U atexit
0000000000600ea0 B environ
U exit
0000000000400990 T ldtest1
00000000004009b4 T ldtest2
00000000004009d8 T main
U printf

$ ldd a.out
a.out:
-lgcc_s.1 => /usr/lib/libgcc_s.so.1
-lc.12 => /usr/lib/libc.so.12

See also: objdump -x a.out
Shared Libraries

Static libraries:
- created by `ar(1)`
- usually end in `.a`
- contain a symbol table within the archive (see `ranlib(1)`)  

Dynamic libraries:
- created by the compiler/linker (i.e., multiple steps)
- usually end in `.so`
- frequently have multiple levels of symlinks providing backwards compatibility / ABI definitions
Dynamically Linked Shared Libraries

- the path to the link loader is embedded in the binary
- the link loader needs to know where to find all required shared libraries
- the path to the libraries may be
  - configured system wide
  - hard-coded into the link loader
  - influenced by e.g., LD_LIBRARY_PATH
  - embedded in the binary (-Wl,-rpath)
- other environment variables may influence the behavior (LD_PRELOAD)
Lecture 12

Advanced I/O / Encryption in a Nutshell
A central logging facility
syslog(3)

```
#include <syslog.h>
void openlog(const char *ident, int logopt, int facility);
void syslog(int priority, const char *message, ...);
```

`openlog(3)` allows us to set specific options when logging:

- prepend `ident` to each message
- specify logging options (LOG_CONS | LOG_NDELAY | LOG_PERRO | LOG_PID)
- specify a facility (such as LOG_DAEMON, LOG_MAIL etc.)

`syslog(3)` writes a message to the system message logger, tagged with `priority`.

A `priority` is a combination of a `facility` (as above) and a `level` (such as LOG_DEBUG, LOG_WARNING or LOG_EMERG).
Nonblocking I/O

Recall from our lecture on signals that certain system calls can block forever:

- `read(2)` from a particular file, if data isn’t present (pipes, terminals, network devices)
- `write(2)` to the same kind of file
- `open(2)` of a particular file until a specific condition occurs
- `read(2)` and `write(2)` of files that have mandatory locking enabled
- Certain `ioctl(2)`
- Some IPC functions (such as `sendto(2)` or `recv(2)`)  

Nonblocking I/O lets us issue an I/O operation and not have it block forever. If the operation cannot be completed, return is made immediately with an error noting that the operating would have blocked (EWOULDBLOCK or EAGAIN).
# Advisory Locking

```c
#include <fcntl.h>

int flock(int fd, int operation);

Returns: 0 if OK, -1 otherwise
```

- applies or removes an advisory lock on the file associated with the file descriptor `fd`
- `operation` can be `LOCK_NB` and any one of:
  - `LOCK_SH`
  - `LOCK_EX`
  - `LOCK_UN`
- locks entire file
Advisory “Record” locking

```c
#include <unistd.h>

int lockf(int fd, int value, off_t size);
```

Returns: 0 on success, -1 on error

`value` can be:

- `F_ULOCK` – unlock locked sections
- `F_LOCK` – lock a section for exclusive use
- `F_TLOCK` – test and lock a section for exclusive use
- `F_TEST` – test a section for locks by other processes

<table>
<thead>
<tr>
<th>Request for</th>
<th>read lock</th>
<th>write lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>no locks</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>one or more read locks</td>
<td>OK</td>
<td>denied</td>
</tr>
<tr>
<td>one write lock</td>
<td>denied</td>
<td>denied</td>
</tr>
</tbody>
</table>
Advisory “Record” locking

Locks are:
- released if a process terminates
- released if a filedescriptor is closed (!)
- not inherited across `fork(2)`
- inherited across `exec(2)`
- released upon `exec(2)` if close-on-exec is set
Asynchronous I/O

<table>
<thead>
<tr>
<th></th>
<th>Blocking</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>Read/write</td>
<td>Read/write (O_NONBLOCK)</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>i/O multiplexing (select/poll)</td>
<td>AIO</td>
</tr>
</tbody>
</table>

Lecture 14: Review

December 2, 2019
Synchronous blocking I/O
Synchronous non-blocking I/O
"Asynchronous" blocking I/O

- **Application**
  - `Read()`
  - `Select()`
  - `Read()`

- **Kernel**
  - `System call - kernel context switch`
  - `EAGAIN / EWOULDBLOCK`  
  - `initiate read I/O`
  - `System call - kernel context switch`
  - `Data movement from kernel space to user space`
  - `Select - data available (readable)`
  - `Read response`
Asynchronous non-blocking I/O

- `aio_read()`
- System call - kernel context switch
- Data movement from kernel space to user space with signal or callback
- Read response
Asynchronous I/O

- System V derived async I/O
  - limited to STREAMS
  - enabled via `ioctl(2)`
  - uses SIGPOLL

- BSD derived async I/O
  - limited to terminals and networks
  - enabled via `fcntl(2)` (O_ASYNC, F_SETOWN)
  - uses SIGIO and SIGURG

- POSIX `aio(3)`
  - kernel process manages queued I/O requests
  - notification of calling process via signal or `sigevent` callback function
  - calling process can still choose to block/wait
# Memory Mapped I/O

```c
#include <sys/types.h>
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

Returns: pointer to mapped region if OK

Protection specified for a region:
- PROT_READ – region can be read
- PROT_WRITE – region can be written
- PROT_EXEC – region can be executed
- PROT_NONE – region can not be accessed

`flag` needs to be one of
- MAP_SHARED
- MAP_PRIVATE
- MAP_COPY

which may be OR'd with other flags (see `mmap(2)` for details).
Cryptography can provide “security” in the areas of:

- **Authenticity**
  - *Is the party I’m talking to actually who I think it is?*

- **Accuracy or Integrity**
  - *Is the message I received in fact what was sent?*

- **Secrecy or Confidentiality**
  - *Did/could anybody else see (parts of) the message?*
How does encryption work?

**Secrecy**: Make sure that the data can only be read by those intended.
- Alice and Bob agree on a way to transform data
- transformed data is sent over insecure channel
- Alice and Bob are able to get data out of the transformation
Cipher Modes

Encryption entails transformation of input data ("plain" or "clear" text) into encrypted output data ("ciphertext"). Input data is generally transformed in one of two ways:

*Stream Cipher*: each bit on plaintext is combined with a pseudo-random cipher digit stream (or *keystream*)

*Block Cipher*: fixed-length blocks of plaintext are transformed into same-sized blocks of ciphertext; may require padding
Lecture 13

Restricting Processes / Containers
Filesystem access

Limitations of the traditional Unix access semantics:

- a file can only have one group owner
- group membership quickly becomes convoluted
- different systems have different limits on the number of groups a user can be a member of
- any modification of group membership requires the sysadmin to make changes (add/remove members, create new groups, ...)

Access Control Lists

POSIX.1e Access Control Lists (ACLs) provide more fine-grained access control:

- user can specify individuals or groups with different access
- implemented as 'Extended Attributes' in the filesystem
- `ls(1)` indicates their presence via a '+' at the end of the permissions string

```bash
$ ls -l hole.c; getfacl hole.c
-rw-------+ 1 jschauma professor 984 Nov 27 21:51 # file: hole.c
# file: hole.c
# owner: jschauma
# group: professor
user::rw-
group::---
group:student:r--
mask::r--
other::---
```
Changeing eUID

- Accomplish privilege separation by dropping raised privileges or changing to an unprivileged user via `seteuid(2)`.
- Allow users to completely become another user via `su(1)`.
- Allow users to selectively run commands with another’s eUID via e.g. `sudo(8)`.
Securelevels, mount options, CPU restrictions

Restrict even root from making specific changes:

- use of “file flags” (chflags(1))
- mount file systems e.g., `readonly`, `noexec`, `nosuid`, ...
- raise the securelevel, so change require reboot

CPU restrictions

- limit process utilization via `ulimit(1)` built-in / `setrlimit(2)`
- change process priority via `nice(1)` / `renice(1)`
- restrict processes to certain processors via `cpuset`
Restricted Shells

Allow a user interactive use of the system but restrict specifically which commands they can run.

- prohibit `cd`
- prohibit changing e.g. `PATH` etc.
- prohibit use of commands containing a `/` (i.e., only commands found in the (fixed) `PATH` can be executed)
- redirecting output into files

Beware trivial break-outs via commands that allow invoking other commands!
Chroots and Jails

Chroots:
- restrict a process’s view of the filesystem hierarchy
- restrict commands by only providing needed executables
- must provide full environment, shared libraries, config files, etc.
- combine with null mounts / mount options
- open file descriptors may be brought into the chroot
- processes outside the chroot are visible!

Jails: First sandbox environment / OS-level virtualization:
- per-jail process view
- changing sysctls or securelevels is prohibited
- modifying the network configuration is prohibited, raw sockets are disabled
- mounting and unmounting filesystems is prohibited
On to containers...

Combine all of the above with POSIX “Capabilities”, control groups (cgroups), namespaces to create an isolated execution environment providing lightweight virtualization, i.e., containers:

- use null and union mounts to provide the right environment
- restrict processes in their utilization
- restrict filesystem views
- restrict processes from what they can see
- restrict processes from what they can do
Lecture 14

Review / Coding Guidelines
"Consistency underlies all principles of quality."
Frederick P. Brooks, Jr
Program Design


UNIX programs...
- ...are simple
- ...have a manual page
- ...follow the element of least surprise
- ...accept input from stdin
- ...generate output to stdout
- ...generate meaningful error messages to stderr
- ...have meaningful exit codes
The Unix Philosophy

This is the Unix philosophy:

Write programs that do one thing and do it well.

Write programs to work together.

Write programs to handle text streams, because that is a universal interface.
Unix basics

Write your code and tools such that they work well within the Unix ecosystem:

- write portable code, target different Unix flavors
- use `strerror(3)/perror(3)`
- errors go to `stderr`
- use meaningful return codes
- follow Unix conventions when using e.g. flags, files, config files, passwords, environment variables, ...
The Zen of Python

Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren’t special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you’re Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it’s a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let’s do more of those!
Readability counts

Visual flow:

- use spaces/tabs/indentation *consistently*
- use a standard width terminal (~80 chars)
- refactor if code wraps / trails off right side
- refactor if logic doesn’t fit into about one screen height
- never repeat the same code block
Readability counts

Code is language:

- you are not charged per character
- use descriptive function and variable names
- use comments where necessary; explain why, not what
- don’t use magic numbers
- write boring code
Structure

- "do one thing and do it well" also applies to functions
- eliminate side-effects
- minimize the use of global variables
- keep open(2)/close(2), malloc(3)/free(3), etc. in same (visual/logical) scope
- separating code into multiple files helps clarify your interfaces
Pitfalls

- check the return value of any function that can fail!
- avoid file I/O whenever possible
- avoid using temporary files whenever possible
- don’t assume you can write to the current working directory
- be explicit in setting permissions; set/use `umask(2)`
- use an exit handler to clean up after yourself
- retain idempotency whenever possible
Using Crypto

- don’t write your own crypto code, use existing libraries
- don’t invent your own security protocol, even if you can’t think of a way that you could break it
- don’t invent your own source of entropy
- always seed your PRNG, salt your hashes
- default to reasonable crypto primitives:
  - 2048 bit RSA for asymmetric key cryptography
  - AES256-CBC for symmetric key cryptography
  - HMAC-SHA256 for integrity
Handling Secrets

- use a Key Management System; integrate with common libraries/API
- allow the user different options of providing secrets; see e.g. openssl(1)
  - on the command-line (note: visible in process table!)
  - via the environment (note: possibly visible to other users; often then stored in shell initialization files)
  - from a file (note: ensure correct permissions!)
  - from a file descriptor
  - from stdin
  - prompt from the tty
- sanitize / zero out secrets after use
- don’t log secrets!
Input validation

Never trust anything from outside of your control. This includes:

- data input directly provided by the user
- data indirectly / implicitly controlled by the user (e.g. HTTP headers)
- data read from files you think you control (e.g. config or state files)
- anything from the environment;
  - use `getpwent(3)` instead of e.g. `HOME` or `USER`
  - explicitly set e.g. `PATH`, `LD_LIBRARY_PATH`
  - explicitly unset e.g. `LD_PRELOAD`
Input validation

- length checks (in both directions!)
- range checks on numeric fields, character ranges
- check path names against directory escapes (./.././../)
- prefer whitelists over blacklists
- encode data before validation or use
- use type check assertions
Subprocesses

- don’t use `system(3), popen(3)` with any user provided input
- prefer `fork(2)/exec(3)`
- explicitly set a trusted `PATH, LD_*` etc.
- never invoke commands from a temporary or relative location (e.g. `/tmp/cmd, ./cmd`)
- set a suitable `umask(2)`
Setuid

drop privileges as early as possible

only raise privileges for sections you need

permanently drop privileges if you no longer need them

be aware of which subprocesses might let you break out of your program or which could spawn a shell (e.g. `vi(1)`)  

be aware of which operations are atomic and which aren’t

beware signal and exit handlers
File I/O

- avoid wherever possible
- assert suitable protection on private files (see e.g. ssh(1))
- be careful when opening, unlinking, overwriting files based on user input / user provided pathnames
- don’t use temporary files
  - set a restrictive umask(2)
  - use mktemp(3)
  - unlink via exit handler

https://www.netmeister.org/blog/mktemp.html
Coding techniques

- treat functions as black boxes, minimize side-effects, avoid global variables
- explicitly mark variables / function arguments as 'const' (see also: const.c)
- check your return codes!
- avoid magic numbers
- use `strncpy(3)/strncat(3)` etc. instead of `strcpy(3)/strcat(3)` etc.
- fail early, fail explicitly
- allocate / free resource in same scope
- check the boundaries of your buffers
- use compiler options (e.g. `-fsanitize=address`), debugging libraries, analysis tools (e.g. valgrind)
- understand and resolve all compiler warnings
  (-Wall -Werror -Wpedantic)
Core Principles

- Simplify – don’t write any code you don’t need.
- Minimize your Attack Surface – only expose (interfaces, API functionality, access, ...) what is needed
- Secure Defaults – user and group permissions, umask, PATH, locations, ...
- Assume that Human Behavior Will Introduce Vulnerabilities into Your System
- Know Your Enemy – understand your threat model
Core Principles

- Principle of Least Privilege – only access, use, or accept information/resources that are strictly needed; don’t run unprivileged unless/until privileged mode is needed
- Fail Closed – (unexpected) failure must not lead to e.g. access, information disclosure, increased privileges, ...
- Defense in Depth – any component or tool needs to be safe to use; do not rely on outside mechanisms or protections
- Kerkhoff’s Principle – ”the enemy knows the system”; avoid Security by Obscurity
- Assume a Hostile Environment
  - always use transport encryption
  - always authenticate all parties
  - authentication ≠ authorization
That's all, folks!