CS631 - Advanced Programming in the UNIX Environment

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
http://www.cs.stevens.edu/~jschauma/631/

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
$
```

In a nutshell: the ”what”

```
$ 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 *);
$
```

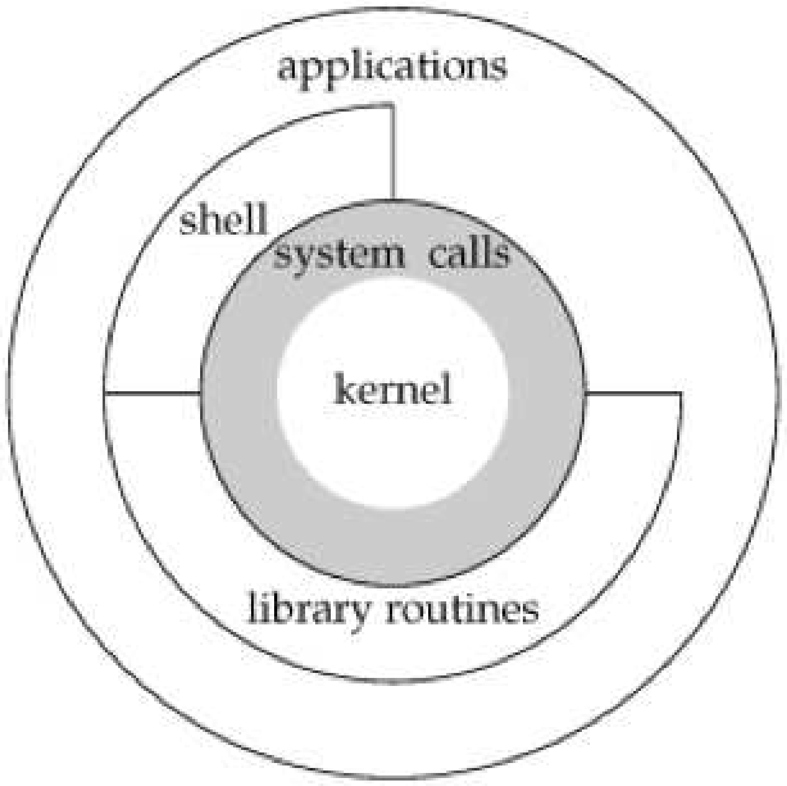
In a nutshell: the ”what”

- 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 ”how”

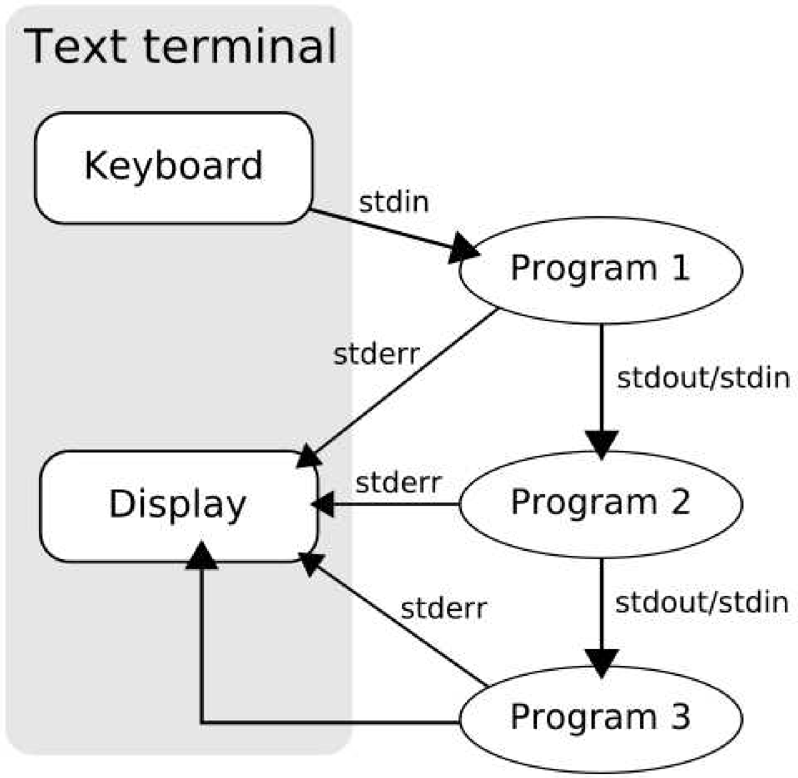
```
static char dot[] = ".", *dotav[] = { dot, NULL };
struct winsize win;
int ch, fts_options;
int kflag = 0;
const char *p;

setprogname(argv[0]);
setlocale(LC_ALL, "");

/* Terminal defaults to -Cq, non-terminal defaults to -1. */
if (isatty(STDOUT_FILENO)) {
    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &win) == 0 &&
        win.ws_col > 0)
        termwidth = win.ws_col;
    f_column = f_nonprint = 1;
} else
    f_singlecol = 1;

/* Root is -A automatically. */
if (!getuid())
    f_listdot = 1;

fts_options = FTS_PHYSICAL;
while ((ch = getopt(argc, argv, "1ABCFLRSTWabcdfghiklmnopqrstuwx")) != -1) {
    switch (ch) {
    /*
     * The -1, -C, -l, -m and -x options all override each other so
     * shell aliasing works correctly.
     */
    case '1':
        f_singlecol = 1;
```

UNIX Basics: Architecture



UNIX Basics: Pipelines

Say ”Thank you, Douglas McIlroy!”



http://is.gd/vGHO9J

Program Design

”Consistency underlies all principles of quality.”
Frederick P. Brooks, Jr

Program Design

`https://secure.wikimedia.org/wikipedia/en/wiki/Unix_philosophy`

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

Files and Directories

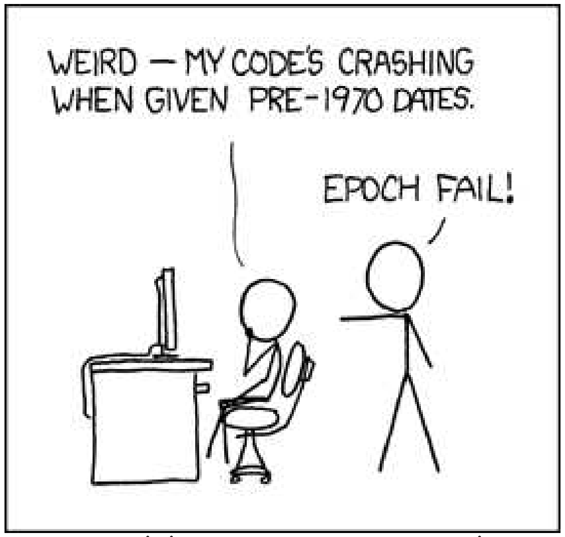
- The UNIX filesystem is a tree structure, with all partitions mounted under the root (/). File names may consist of any character except / and NUL as pathnames are a sequence of zero or more filenames separated by /'s.
- Directories are special ”files” that contain mappings between *inodes* and *filenames*, called directory entries.
- All processes have a current working directory from which all relative paths are specified. (Absolute paths begin with a slash, relative paths do not.)

User Identification

- *User ID*s and *group ID*s are numeric values used to identify users on the system and grant permissions appropriate to them.
- *Group ID*s come in two types; *primary* and *secondary*.

Unix Time Values

Calendar time: measured in seconds since the UNIX epoch (Jan 1, 00:00:00, 1970, GMT). Stored in a variable of type `time_t`.



`https://www.xkcd.com/376/`

`https://secure.wikimedia.org/wikipedia/en/wiki/Year_2038_problem`

Unix Time Values

Process time: central processor resources used by a process. Measured in *clock ticks* (`clock_t`). Three values:

- clock time
- user CPU time
- system CPU time

```
$ time grep -r _POSIX_SOURCE /usr/include >/dev/null
```

Standard I/O

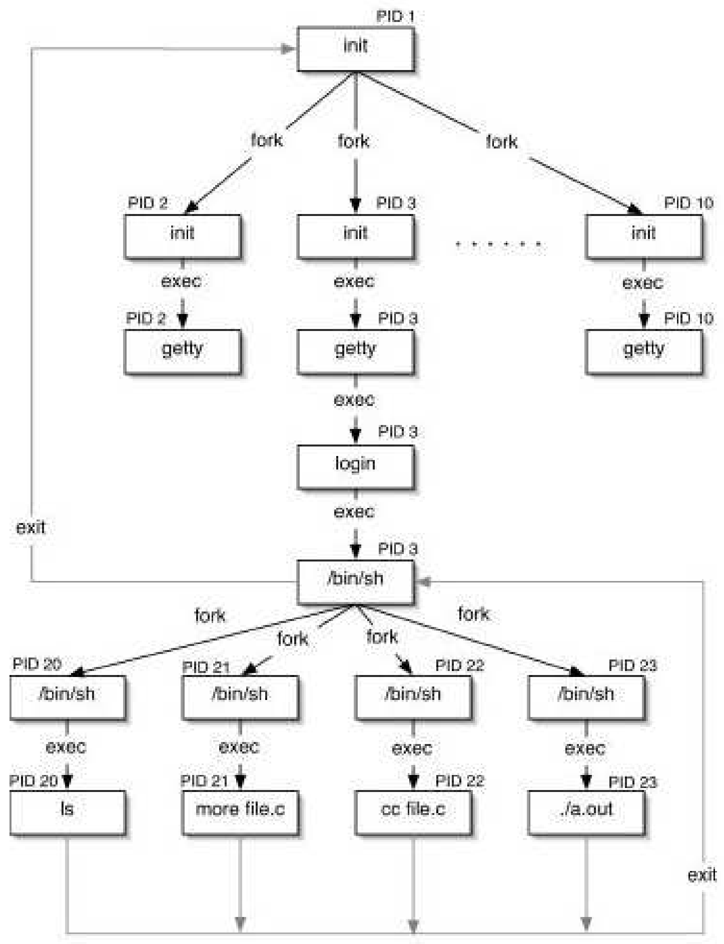
- Standard I/O:
  - **file descriptors**: Small, non-negative integers which identify a file to the kernel. The shell can redirect any file descriptor.
  - kernel provides **unbuffered** I/O through e.g. `open read write lseek close`
  - kernel provides **buffered** I/O through e.g. `getc putc fopen fread fwrite`

Processes

Programs executing in memory are called *processes*.

- Programs are brought into memory via one of the six `exec(3)` functions. Each process is identified by a guaranteed unique non-negative integer called the *processes ID*. New processes can **only** be created via the `fork(2)` system call.
- **process control** is performed mainly by the `fork(2)`, `exec(3)` and `waitpid(2)` functions.

Processes


PID 1
init
fork
fork
fork
PID 2
init
exec
PID 2
getty
PID 3
init
exec
PID 3
getty
exec
PID 3
login
exec
PID 3
/bin/sh
PID 10
init
exec
PID 10
getty
exit
fork
fork
fork
fork
PID 20
/bin/sh
exec
PID 20
ls
PID 21
/bin/sh
exec
PID 21
more file.c
PID 22
/bin/sh
exec
PID 22
cc file.c
PID 23
/bin/sh
exec
PID 23
./a.out
exit

Processes

```
$ pstree -hapun | more
```

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:

```
#include <string.h>
char *strerror(int errnum)
```

Returns: pointer to message string

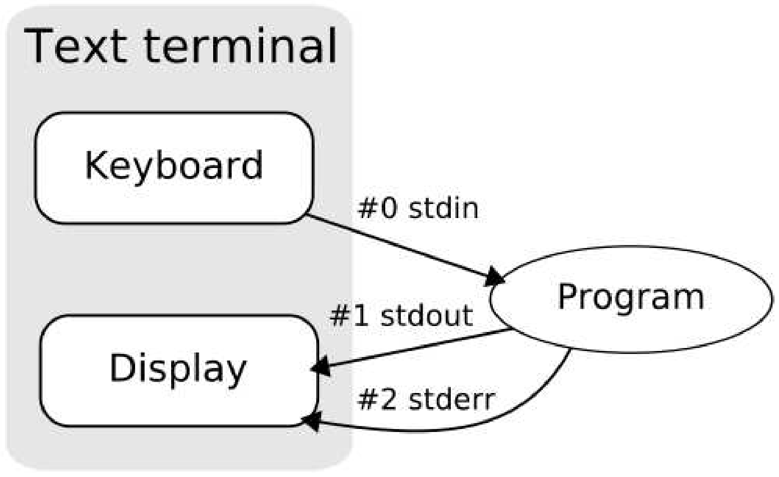
```
#include <stdio.h>
void perror(const char *msg)
```

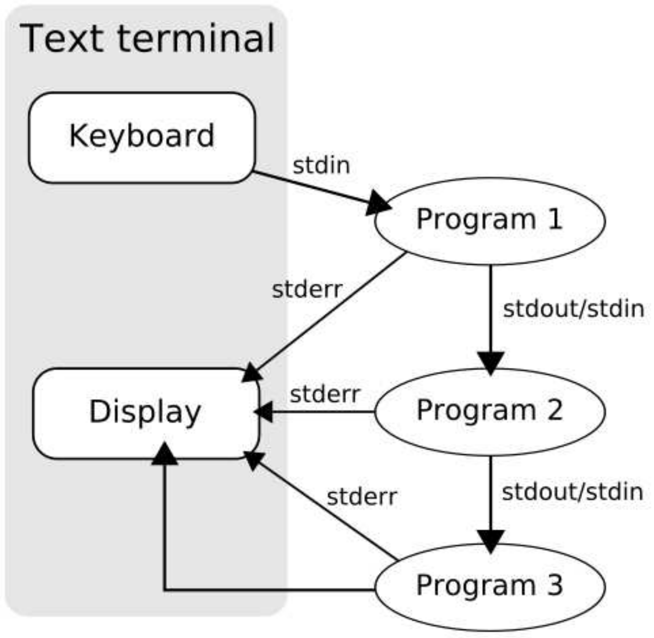
Lecture 02

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.





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^{TM}. Use `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`.

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$^{\text{TM}}$. Use `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`.

See also: `http://en.wikipedia.org/wiki/File_descriptor`

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 recources. This requires us to look at:

- atomicity of these operations
- file sharing
- manipulation of file descriptors

open(2)

```
#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

```
#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 filedescriptor 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)

```
#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)

```
#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)

```
#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)`:

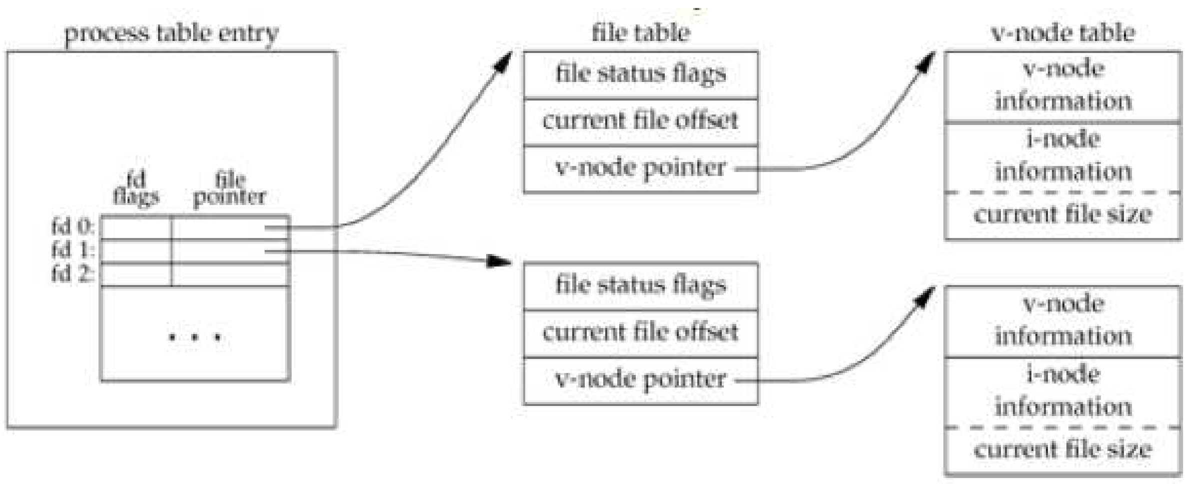
- seek to a negative offset
- seek 0 bytes from the current position
- seek past the end of the file

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

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.

Atomic Operations

In order to ensure consistency across multiple writes, we require *atomicity* in some operations.

An operation is atomic if either *all* of the steps are performed or *none* of the steps are performed.

Suppose UNIX didn't have `O_APPEND` (early versions didn't). To append, you'd have to do this:

```
if (lseek(fd, 0L, 2) < 0) {         /* position to EOF */
    fprintf(stderr, "lseek error\n");
    exit(1);
}

if (write(fd, buff, 100) != 100) { /* ...and write */
    fprintf(stderr, "write error\n");
    exit(1);
}
```

What if another process was doing the same thing to the same file?

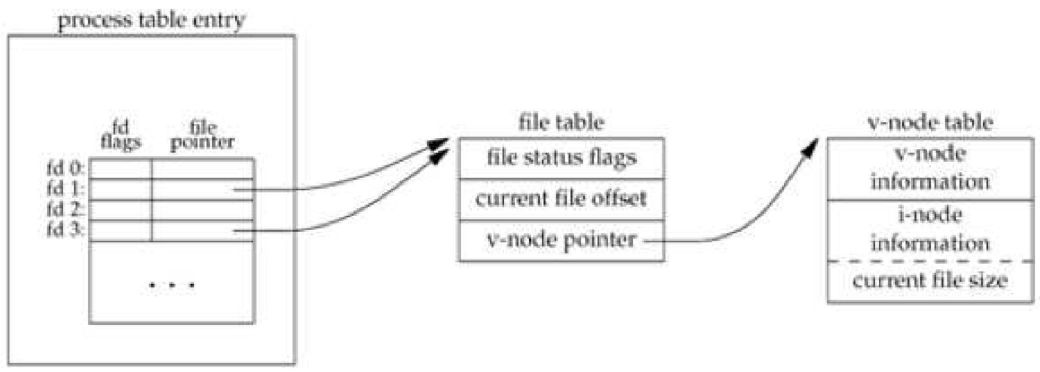
`dup(2)` and `dup2(2)`

```
#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

`stat(2)` family of functions

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

int stat(const char *path, struct stat *sb);
int lstat(const char *path, struct stat *sb);
int 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:

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by `exec` functions

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.

`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)

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

mode_t umask(mode_t numask);
```

Returns: previous file mode creation mask

`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)

```
#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)`

```
#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)

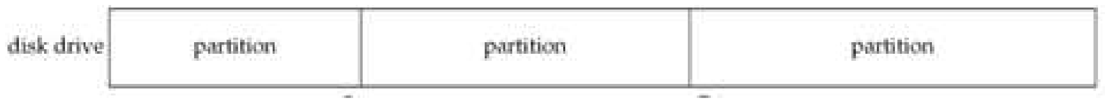
`chown` and friends clear all setuid or setgid bits.

Lecture 04

File Systems, System Data Files, Time & Date

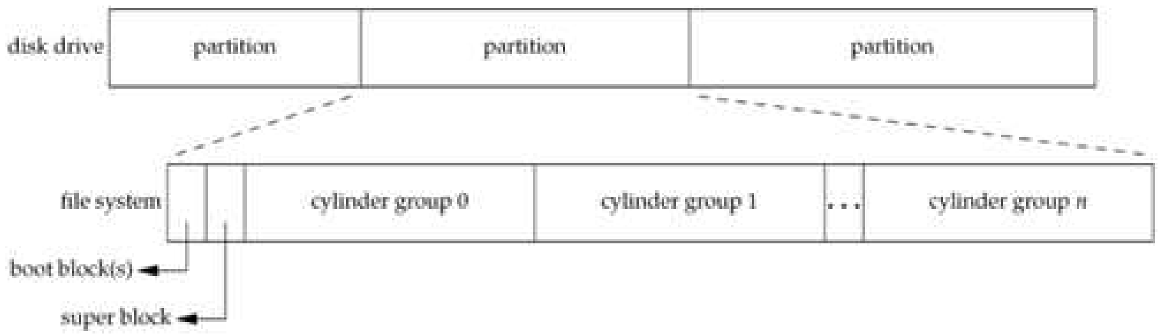
File Systems

- a disk can be divided into logical *partitions*



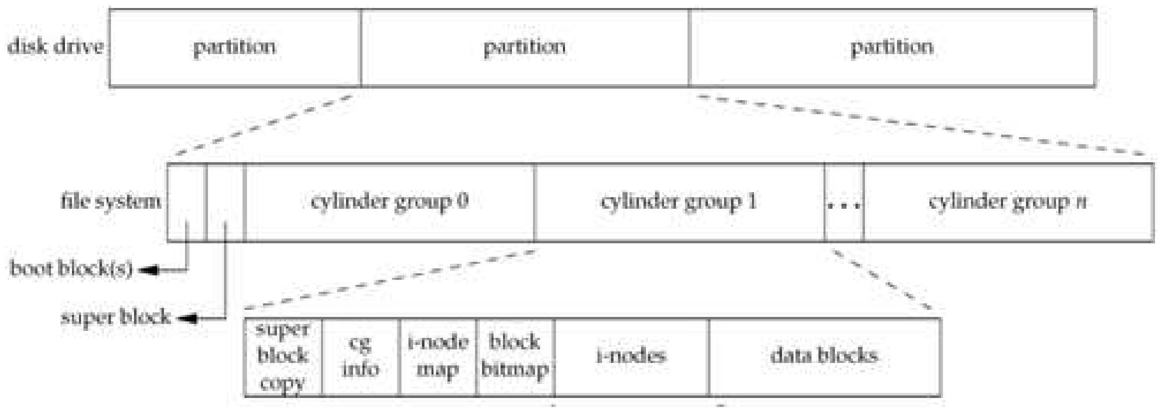
File Systems

- a disk can be divided into logical *partitions*
- each logical *partition* may be further divided into *file systems* containing *cylinder groups*



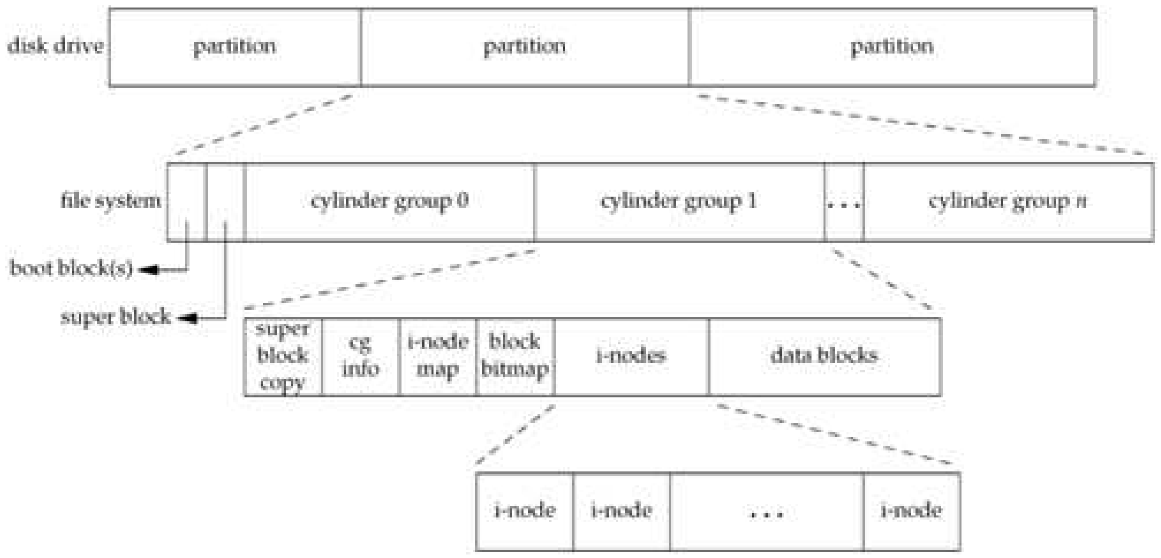
File Systems

- 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*



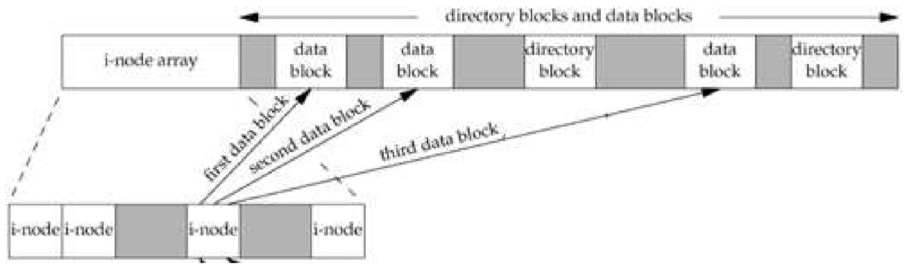
File Systems

- 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*



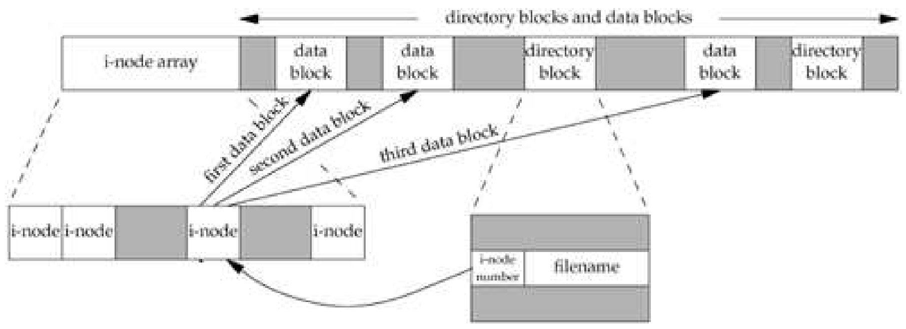
File Systems

- 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*



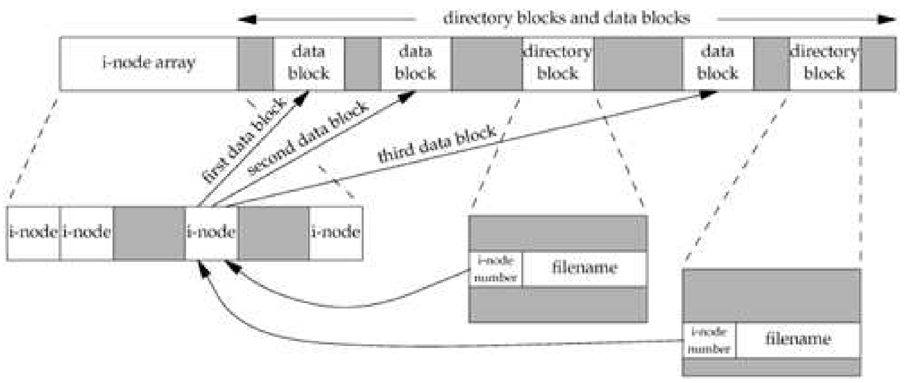
File Systems

- 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



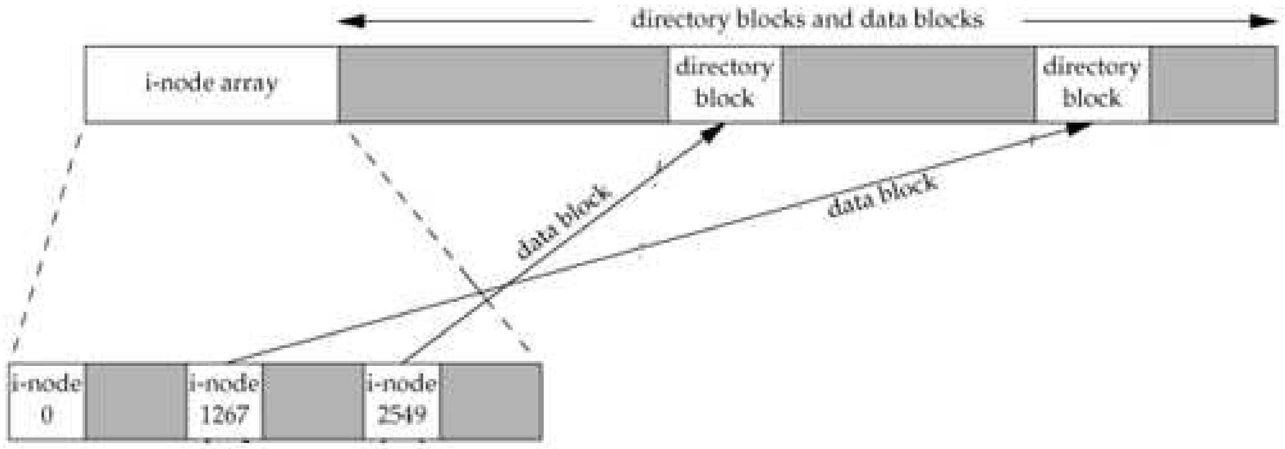
File Systems

- 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



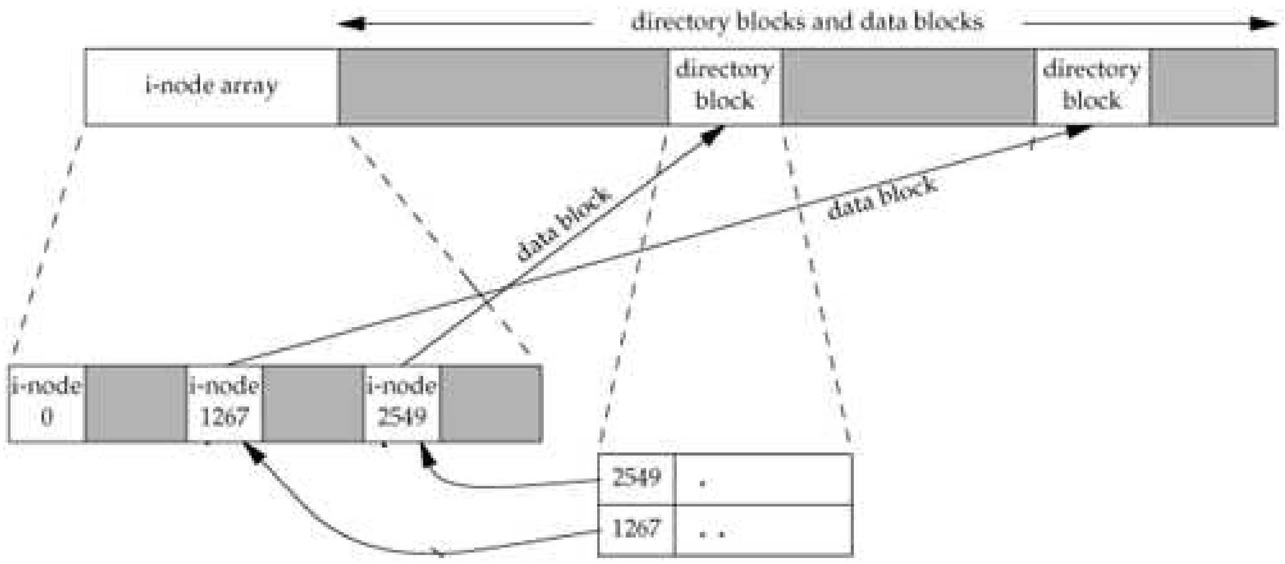
Directories

- directories are special ”files” containing hardlinks



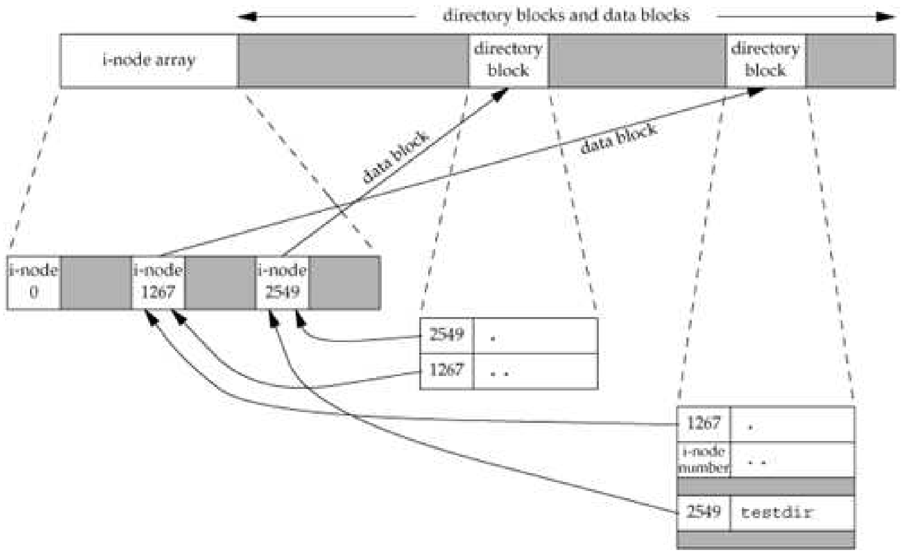
Directories

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



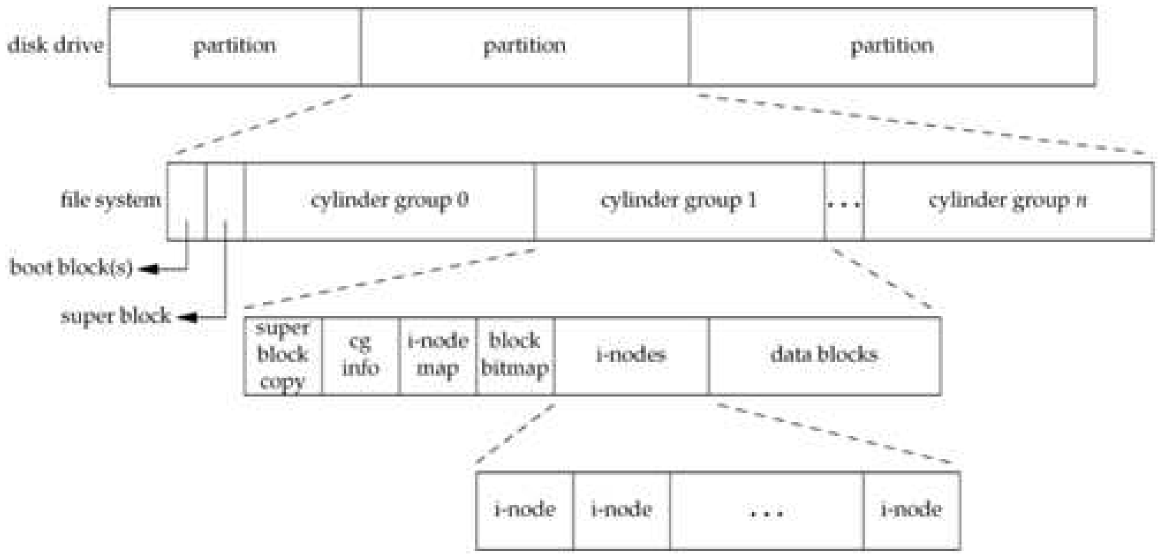
Directories

- 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



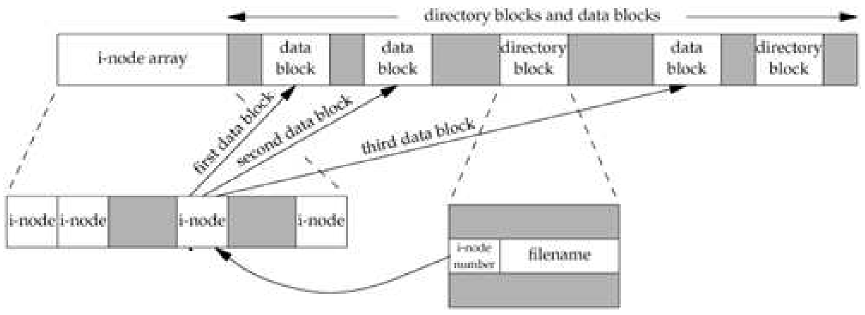
Inodes

- 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)



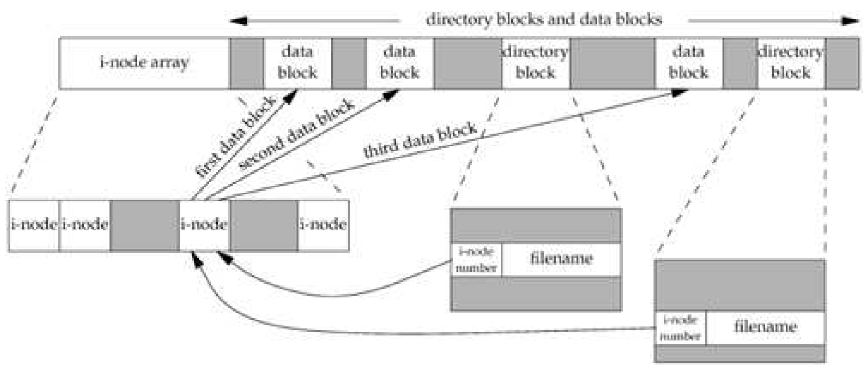
Inodes

- 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).



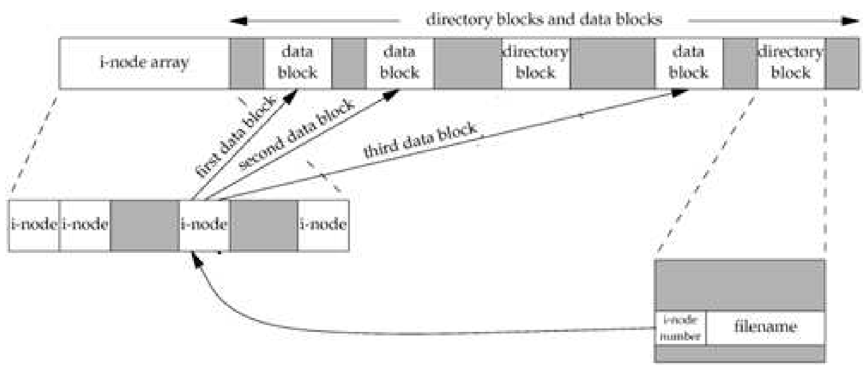
Inodes

- the *inode* contains most of information found in the `stat` structure.
- every *inode* has a *link count* (`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* (`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)`

```
#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)

```
#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)

```
#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

```
#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

```
#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.

File Times

```
#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)`

```
#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.

```
#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

```
#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.

```
#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:

Description	`struct passwd` member	POSIX.1
username	`char *pw_name`	x
encrypted passwd	`char *pw_passwd`	
numerical user id	`uid_t pw_uid`	x
numerical group id	`gid_t pw_gid`	x
comment field	`char *pw_gecos`	
initial working directory	`char *pw_dir`	x
initial shell	`char *pw_shell`	x

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

```
#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
```

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

struct passwd *getpwent(void);
                                    Returns: pointer if OK, NULL on error
void setpwent(void);
void endpwent(void);
```

- `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:

Description	`struct group` **member**	**POSIX.1**
groupname	`char *gr_name`	x
encrypted passwd	`char *gr_passwd`	
numerical group id	`uid_t gr_uid`	x
array of pointers to user names	`char **gr_mem`	x

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

Group File

```
#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:

```
#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 opsys 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
```

Time and Date

```
#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
```

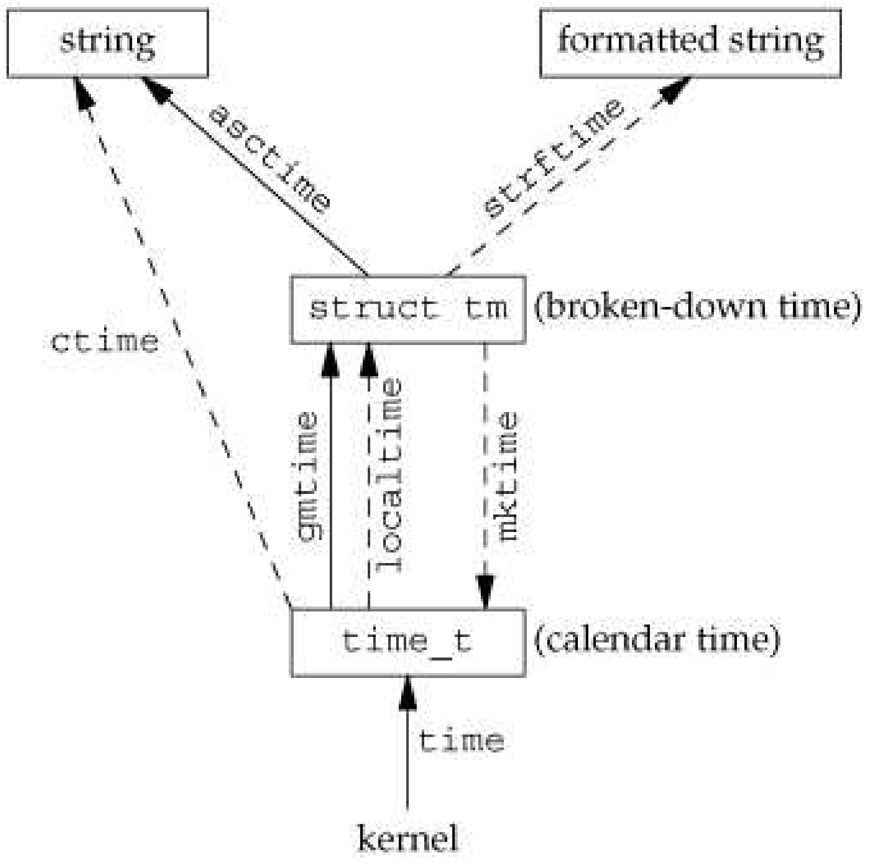
`localtime(3)` takes into account daylight savings time and the *TZ* environment variable. The `mktime(3)` function operates in the reverse direction. To output human readable results, use:

```
#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

Process Environment, Process Control

The `main` function

```
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 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)

```
#include <stdlib.h>

void exit(int status);
void _Exit(int status);


#include <unistd.h>
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*

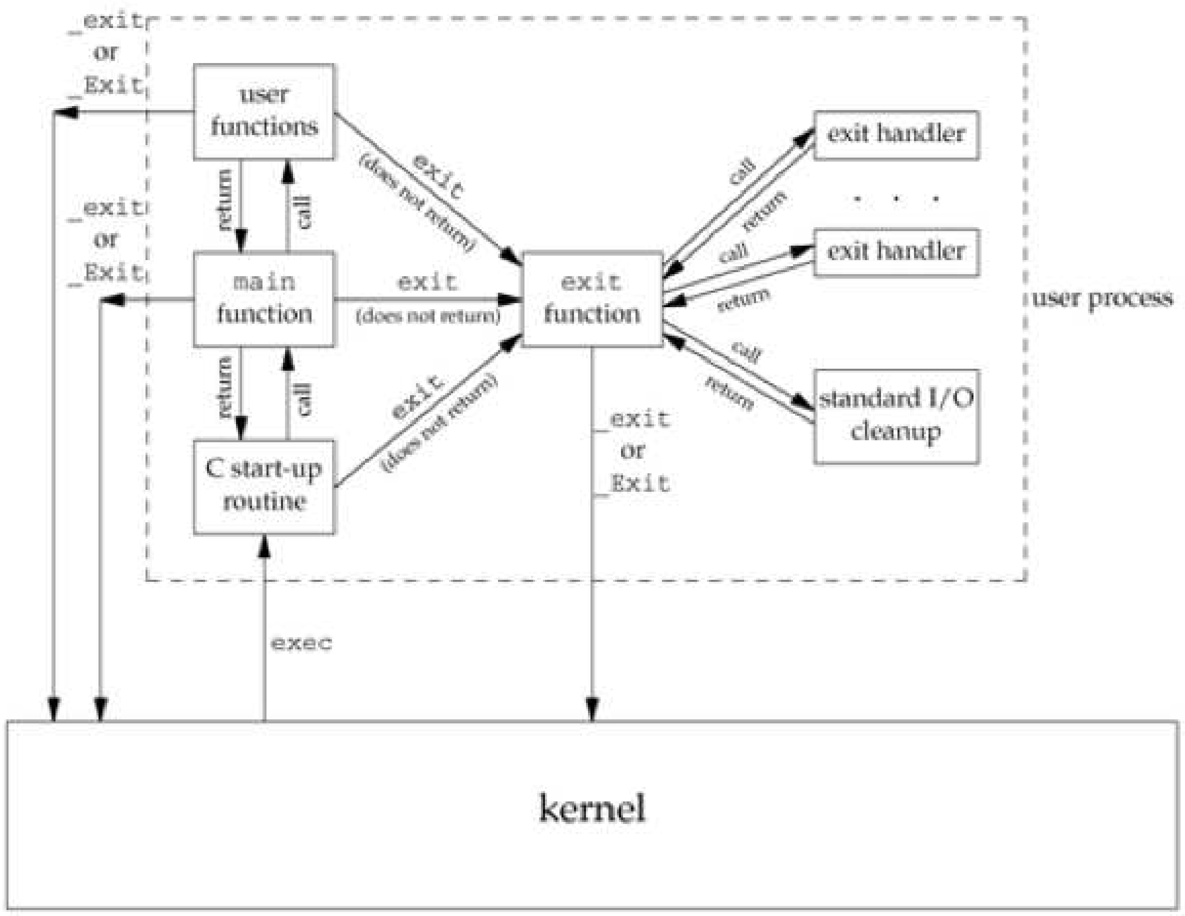
atexit(3)

```
#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:

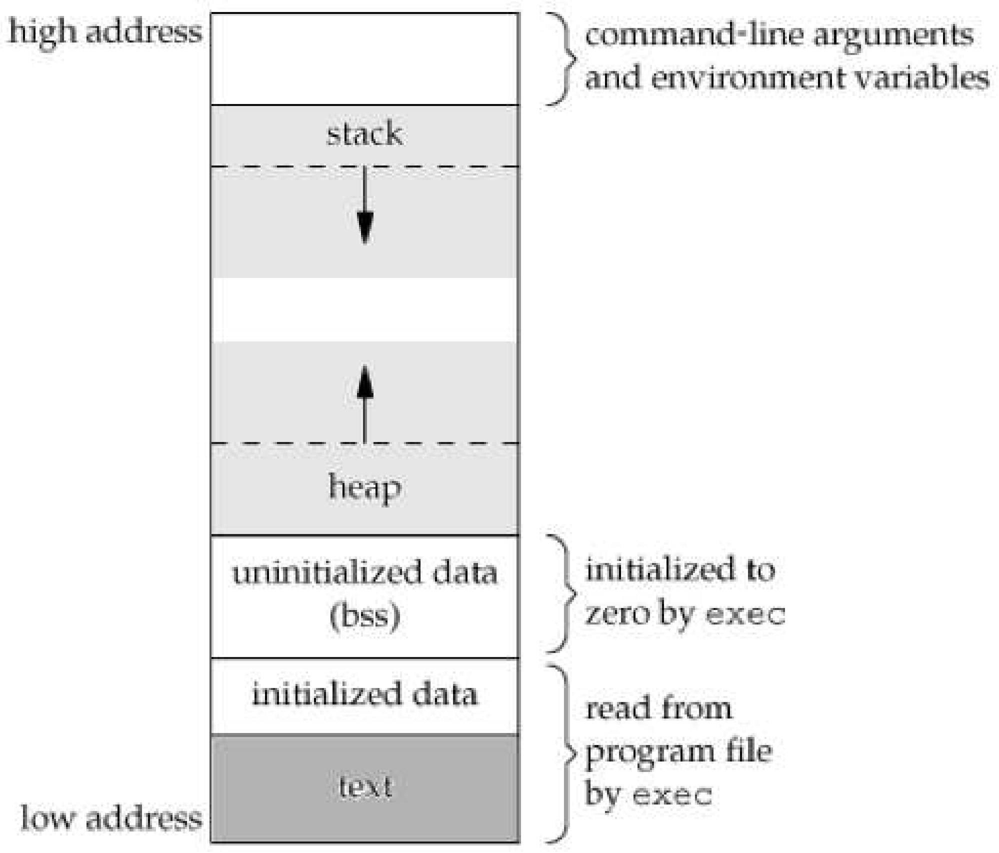
```
extern char **environ;
```

The list is `null` terminated.

These can also be accessed by:

```
#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(cont char *name);
```

Memory Layout of a C Program


high address
low address
stack
heap
uninitialized data (bss)
initialized data
text
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec

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
$
```

`getrlimit(2)` and `setrlimit(2)`

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

int getrlimit(int resouce, struct rlimit *rlp);
int setrlimit(int resouce, const struct rlimit *rlp);
```

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

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

Process ID's 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)

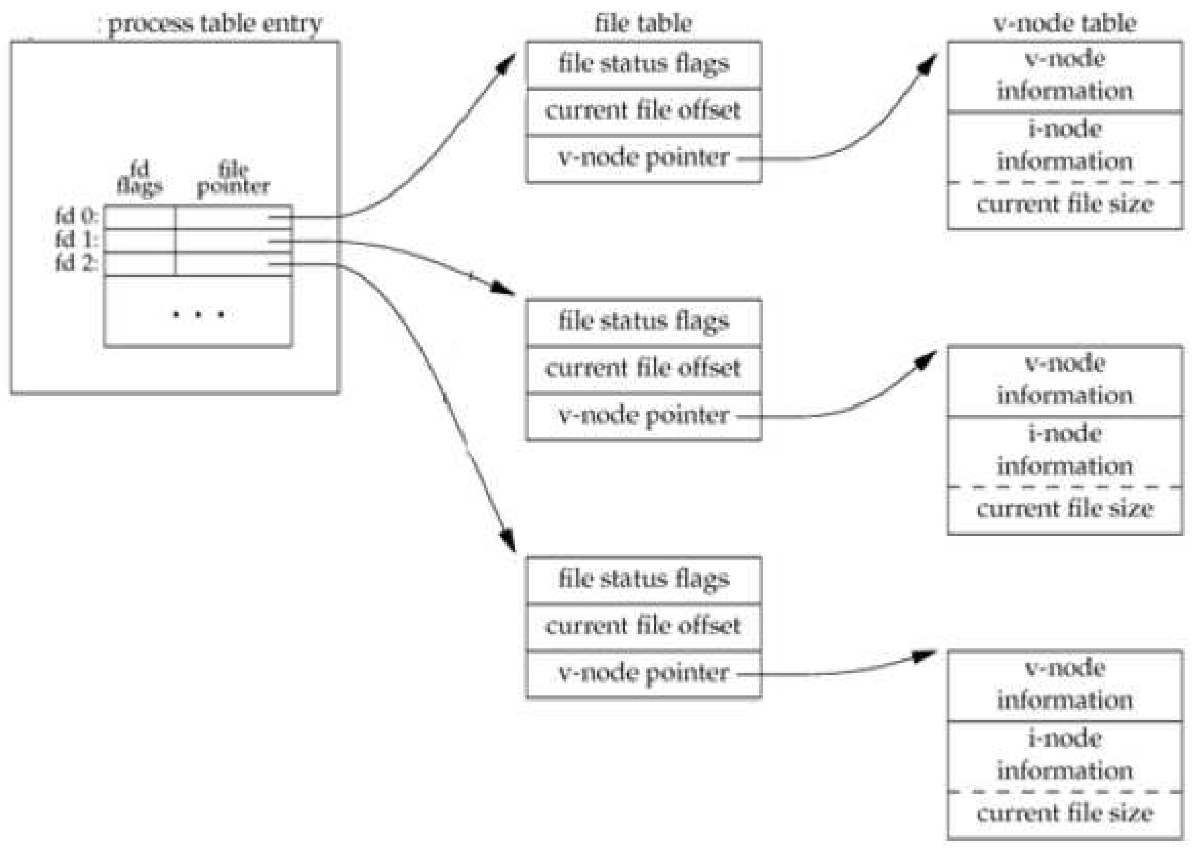
```
#include <unistd.h>

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:

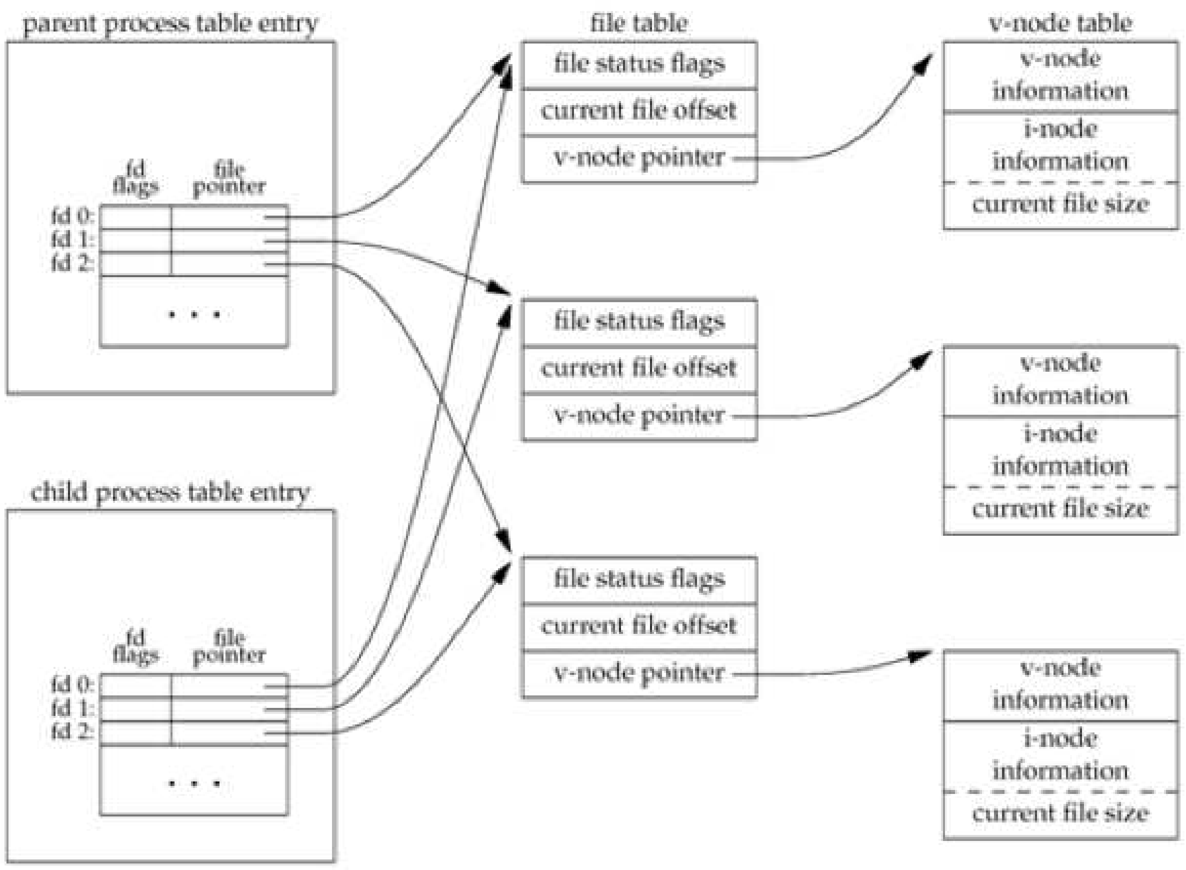
- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors.
- The child process' resource utilizations are set to 0.

Note: no order of execution between child and parent is guaranteed!

fork(2)


process table entry
fd flags
file pointer
fd 0:
fd 1:
fd 2:
. . .
file table
file status flags
current file offset
v-node pointer
v-node table
v-node information
i-node information
current file size

fork(2)


parent process table entry
fd flags
file pointer
fd 0:
fd 1:
fd 2:
. . .
child process table entry
fd flags
file pointer
fd 0:
fd 1:
fd 2:
. . .
file table
file status flags
current file offset
v-node pointer
file status flags
current file offset
v-node pointer
file status flags
current file offset
v-node pointer
v-node table
v-node information
i-node information
current file size
v-node information
i-node information
current file size
v-node information
i-node information
current file size

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(3)` functions

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ...  /* (char *) 0 */);

int execv(const char *pathname, char * const argvp[]);

int execle(const char *pathname, const char *arg0, ...  /* (char *) 0, char *const envp[] */ );

int execve(const char *pathname, char * const argvp[], char * const envp[]);

int execlp(const char *filename, const char *arg0, ...  /* (char *) 0 */);

int execvp(const char *filename, char *const argv[]);
```

The `exec()` family of functions are used to completely replace a running process with a 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)`

```
#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

wait(2) and waitpid(2)

Differences between wait(2), wait3(2), wait4(2) and waitpid(2):

- wait(2) will block until the process terminates, waitpid(2) has an option to prevent it from blocking
- waitpid(2) can wait for a specific process to finish
- wait3(2) and wait4(2) allow you to get detailed resource utilization statistics
- wait3(2) is the same as wait4(2) with a *wpid* value of -1

`wait(2)` and `waitpid(2)`

Once we get a termination status back in `status`, we'd like to be able to determined how a child died. We do this with the following macros:

- `WIFEXITED(status)` – true if the child terminated normally. Then execute `WEXITSTATUS(status)` to get the exit status.
- `WIFSIGNALED(status)` – true if child terminated abnormally (by receiving a signal it didn't catch). The we call:
  - `WTERMSIG(status)` to retrieve the signal number
  - `WCOREDUMP(status)` to see if the child left a core image
- `WIFSTOPPED(status)` – true if the child is currently stopped. Call `WSTOPSIG(status)` to determine the signal that caused this.

Additionally, `waitpid`'s behavior can be modified by supplying `WNOHANG` as an option, which says that if the requested pid has not terminated, return immediately instead of blocking.

Lecture 06

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
```

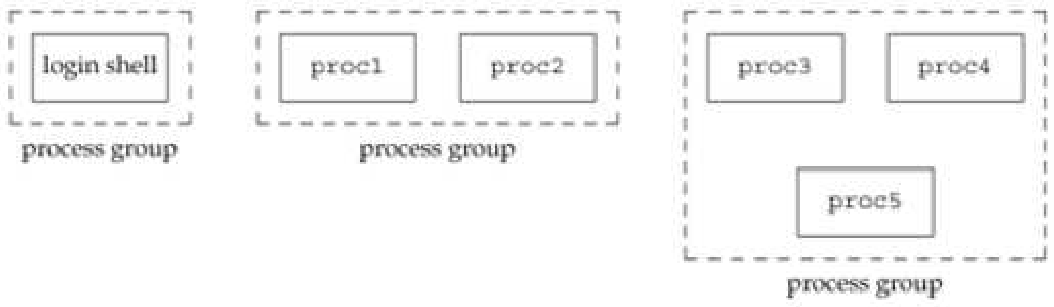
Process Groups

```
#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 assocaited 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 $\Rightarrow$ *login shell*

```
$ proc1 | proc2 &
[1] 10306
$ proc3 | proc4 | proc5
```

Process Groups and Sessions

```
#include <unistd.h>

pid_t setsid(void);
```

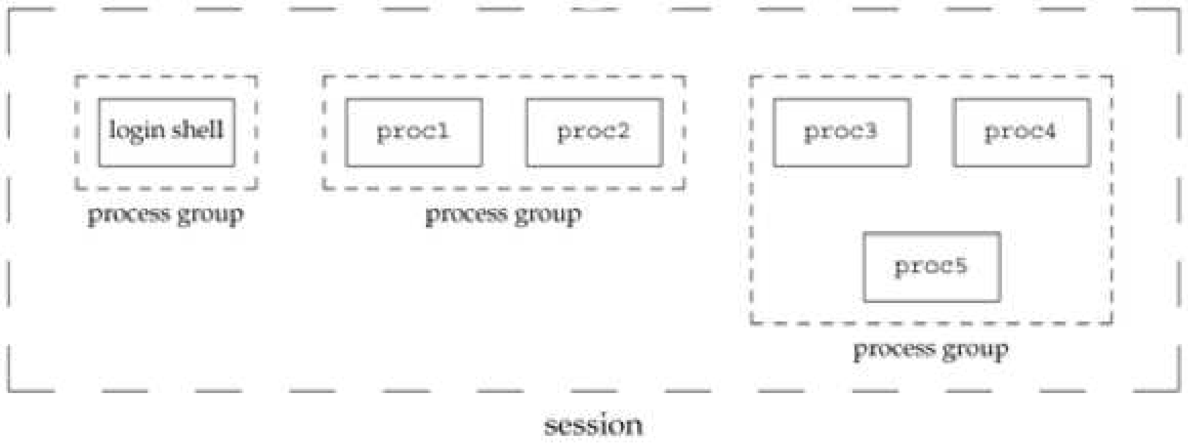
Returns: process group ID if OK, -1 otherwise

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

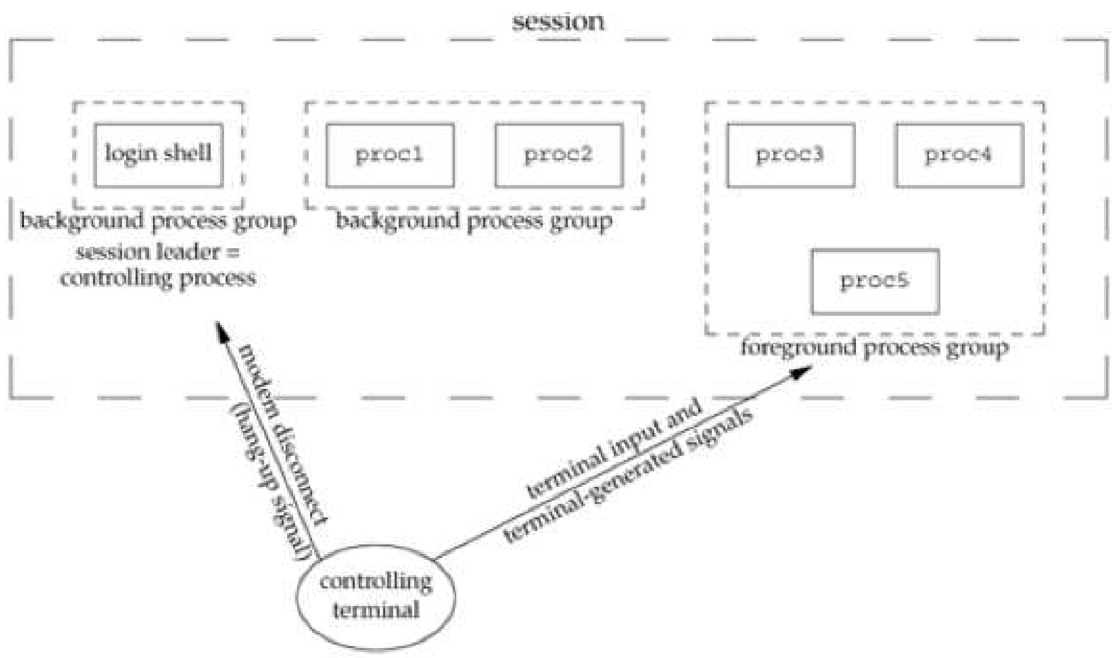
Process Groups



init ⇒ *login shell*

```
$ proc1 | proc2 &
[1] 10306
$ proc3 | proc4 | proc5
```

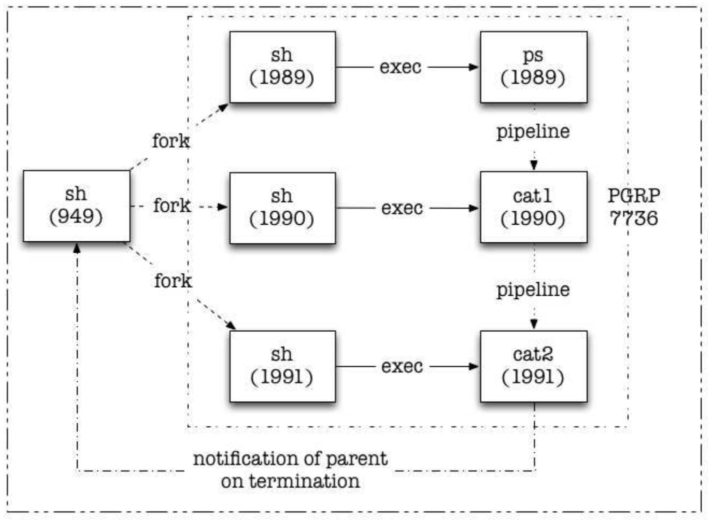
Process Groups and Sessions



init ⇒ *login shell*

```
$ proc1 | proc2 &
[1] 10306
$ proc3 | proc4 | proc5
```

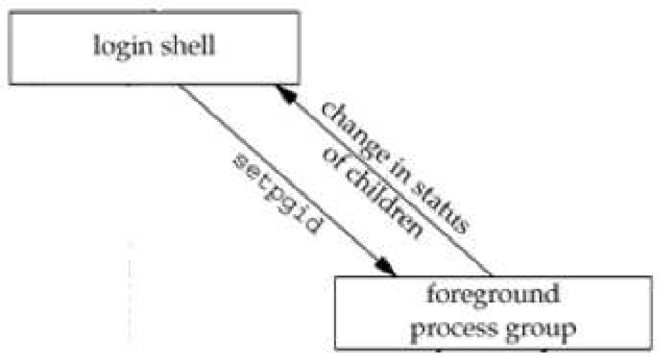
Process Groups and Sessions



Session 949

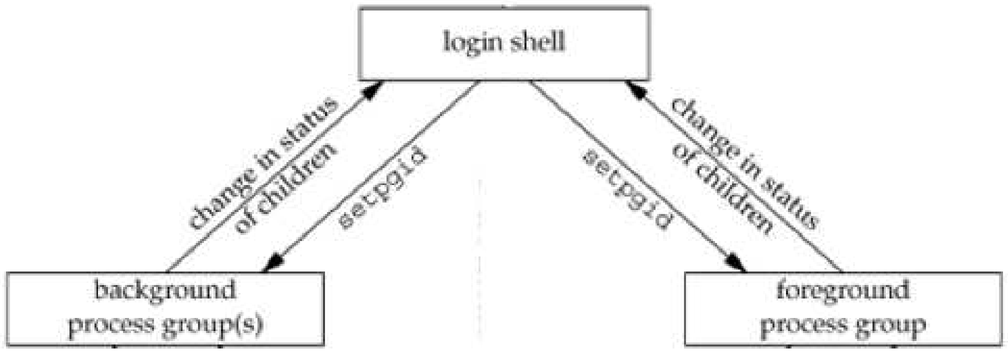
```
$ ps -o pid,ppid,pgid,sess,comm | ./cat1 | ./cat2
  PID  PPID  PGRP  SESS COMMAND
 1989   949  7736   949 ps
 1990   949  7736   949 cat1
 1988   949  7736   949 cat2
  949 21401   949   949 sh
```

Job Control



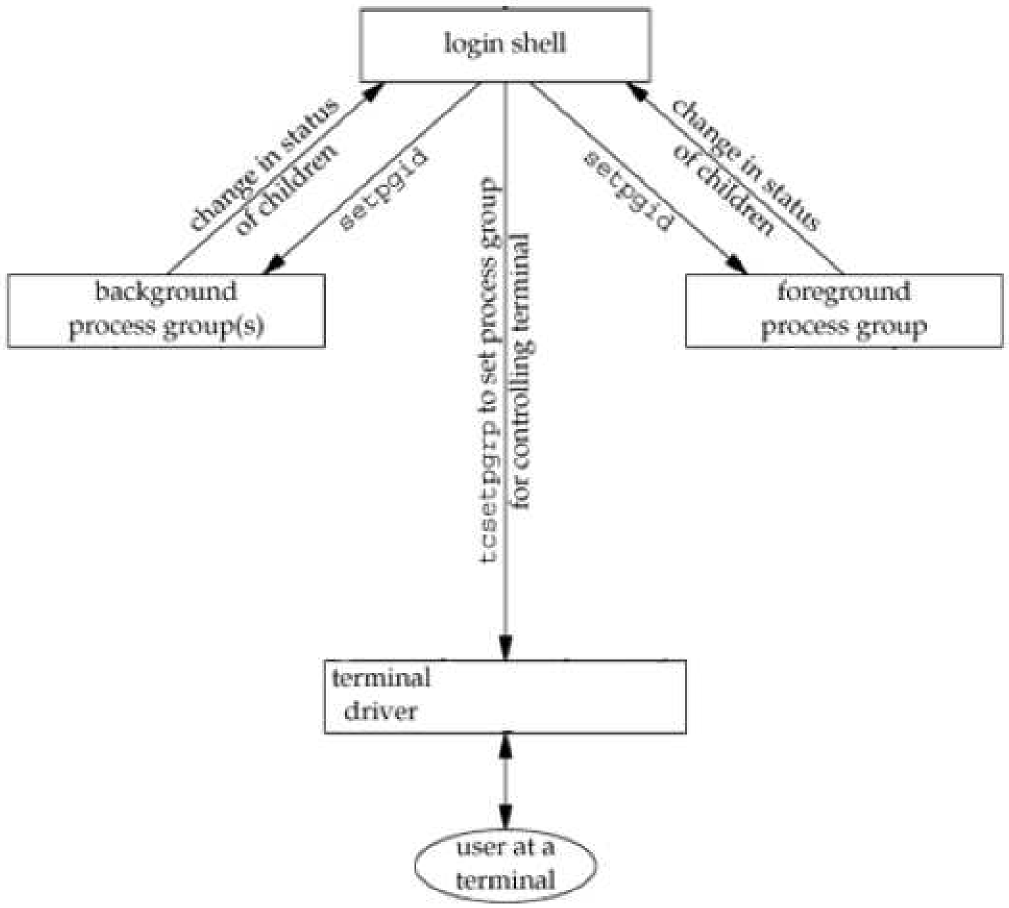
```
$ ps -o pid,ppid,pgid,sess,comm
  PID  PPID  PGRP  SESS COMMAND
24251 24250 24251 24251 ksh
24620 24251 24620 24251 ps
$ echo $?
0
$
```

Job Control


login shell
change in status of children
setpgid
setpgid
change in status of children
background process group(s)
foreground process group

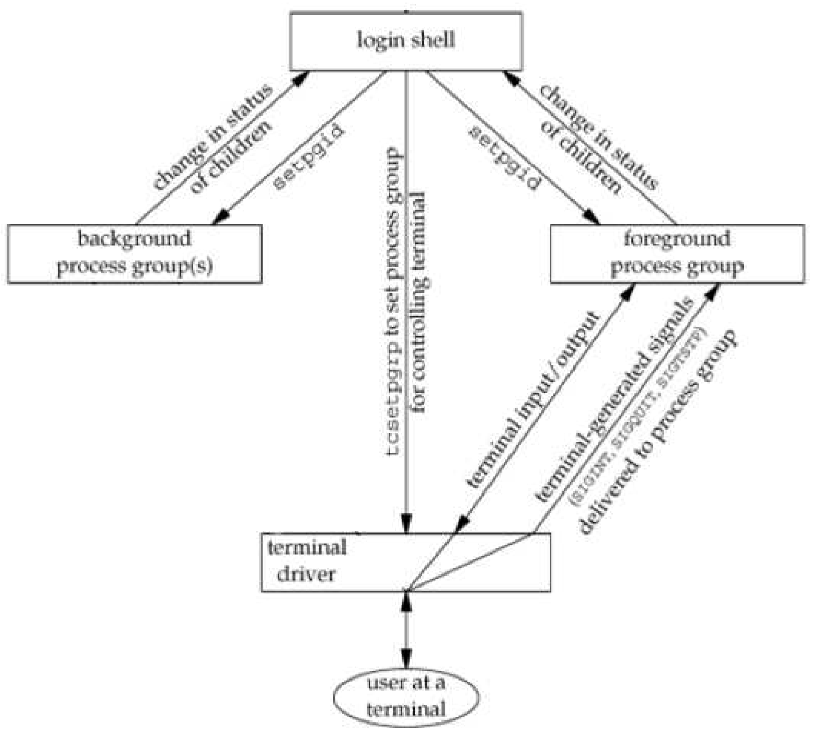
```
$ dd if=/dev/zero of=/dev/null bs=512 count=2048000 >/dev/null 2>&1 &
[1] 24748
$ ps -o pid,ppid,pgid,sess,comm
  PID  PPID  PGRP  SESS COMMAND
24251 24250 24251 24251 ksh
24748 24251 24748 24251 dd
24750 24251 24750 24251 ps
$
[1] +  Done     dd if=/dev/zero of=/dev/null bs=512 count=2048000 >/dev/null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

Job Control


login shell
change in status of children
setpgid
tcsetpgrp to set process group for controlling terminal
setpgid
change in status of children
background process group(s)
foreground process group
terminal driver
user at a terminal

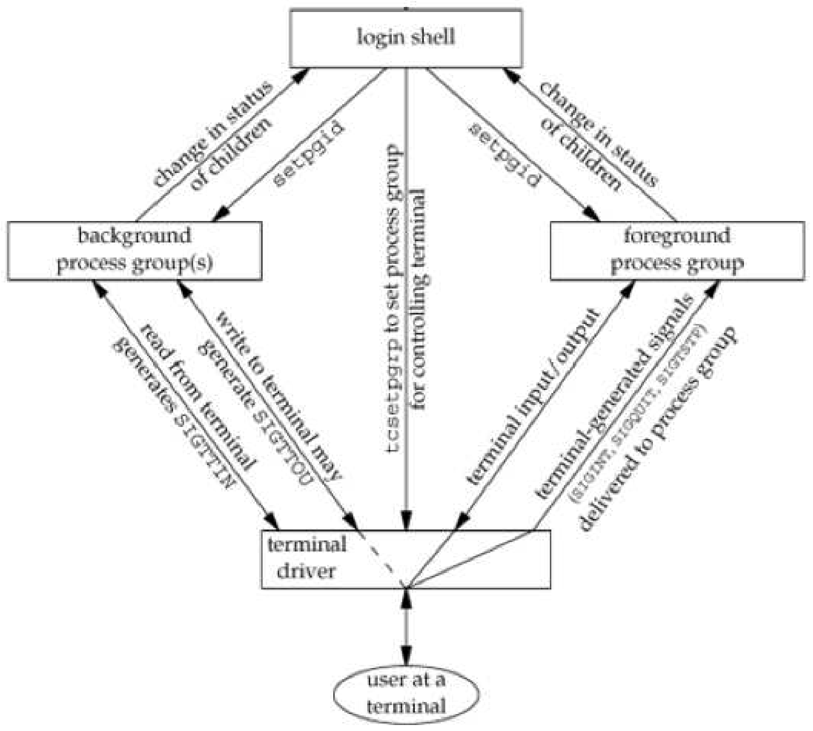
Job Control

```
$ 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
$
```


login shell
change in status of children
setpgid
tcsetpgrp to set process group for controlling terminal
setpgid
change in status of children
background process group(s)
foreground process group
terminal input/output
terminal-generated signals (SIGINT, SIGQUIT, SIGTSTP) delivered to process group
terminal driver
user at a terminal

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.
$
```


login shell
change in status of children
setpgid
tcsetpgrp to set process group for controlling terminal
setpgid
change in status of children
background process group(s)
foreground process group
read from terminal generates SIGTTIN
write to terminal may generate SIGTTOU
terminal input/output
terminal-generated signals (SIGINT, SIGQUIT, SIGTSTP) delivered to process group
terminal driver
user at a 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)`

```
#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)

```
#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

Resetting Signal Handlers

Note: on some systems, invocation of the handler *resets* the disposition to `SIG_DFL`!

```
$ cc -DSLEEP=3 -Wall pending.c
$ ./a.out
=> Establishing initial signal hander via signal(3).
^\sig_quit: caught SIGQUIT (1), now sleeping
sig_quit: exiting (1)
=> Time for a second interruption.
^\sig_quit: caught SIGQUIT (2), now sleeping
sig_quit: exiting (2)
=> Establishing a resetting signal hander via signal(3).
^\sig_quit_reset: caught SIGQUIT (3), sleeping and resetting.
sig_quit_reset: restored SIGQUIT handler to default.
=> Time for a second interruption.
^\Quit: 3
$
```

Signal Queuing

Signals arriving while a handler runs are queued.

```
$ ./a.out >/dev/null
^\sig_quit: caught SIGQUIT (1), now sleeping
^\^\^\^\^\^\sig_quit: exiting (1)
sig_quit: caught SIGQUIT (2), now sleeping
^\^\^\^\sig_quit: exiting (2)
sig_quit: caught SIGQUIT (3), now sleeping
^\sig_quit: exiting (3)
sig_quit: caught SIGQUIT (4), now sleeping
sig_quit: exiting (4)
[...]
```

(Note that ”simultaneously” delivered signals may be ”merged” into one.)

Signal Queuing

Signals arriving while a handler runs are queued.
Unless they are blocked.

```
$ ./a.out
[...]
=> Establishing a resetting signal hander via signal(3).
^\sig_quit_reset: caught SIGQUIT (1), sleeping and resetting.
sig_quit_reset: restored SIGQUIT handler to default.
=> Time for a second interruption.
=> Blocking delivery of SIGQUIT...
=> Now going to sleep for 3 seconds...
^\
=> Checking if any signals are pending...
=> Checking if pending signals might be SIGQUIT...
Pending SIGQUIT found.
=> Unblocking SIGQUIT...
Quit: 3
```

Signal Queuing

Multiple identical signals are queued, but you can receive a different signal *while in a signal handler*.

```
$ ./a.out >/dev/null
^\sig_quit: caught SIGQUIT (1), now sleeping
^\^\^\^\^Csig_int: caught SIGINT (2), returning immediately
sig_quit: exiting (2)
sig_quit: caught SIGQUIT (3), now sleeping
^\^\sig_quit: exiting (3)
sig_quit: caught SIGQUIT (4), now sleeping
sig_quit: exiting (4)
[...]
```

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`.

Interrupted System Calls

Previously necessary code to handle `EINTR`:

```
again:
        if ((n = read(fd, buf, BUFFSIZE)) < 0) {
                if (errno == EINTR)
                        goto again;  /* just an interrupted system call */
                /* handle other errors */
}
```

Nowadays, many Unix implementations automatically restart certain system calls. See also: `sigaction(2)/SA_RESTART`

Reentrant Functions

If your process is currently handling a signal, what functions are you allowed to use?

POSIX lists around 118 reentrant functions.
See p. 306 in Stevens for a list.

See also:
`http://www.ibm.com/developerworks/linux/library/l-reent/index.html`

Lecture 07

Interprocess Communication

Pipes: `pipe(2)`

```
#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)`

```
#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

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.

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)`

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Some of the currently supported domains are:

Domain	Description
PF_LOCAL	local (previously UNIX) domain protocols
PF_INET	ARPA Internet protocols
PF_INET6	ARPA IPv6 (Internet Protocol version 6) protocols
PF_ARP	RFC 826 Ethernet Address Resolution Protocol
...	...

Some of the currently defined types are:

Type	Description
SOCK_STREAM	sequenced, reliable, two-way connection based byte streams
SOCK_DGRAM	connectionless, unreliable messages of a fixed (typically small) maximum length
SOCK_RAW	access to internal network protocols and interfaces
...	...

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)

Sockets: Connections using stream sockets

- 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 09

Shared Libraries

Shared Libraries

What is a shared library, anyway?

- contains a set of callable C functions (ie, implementation of function prototypes defined in `.h` header files)
- code is position-independent (ie, 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?

- contents of *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

Statically Linked Shared Libraries

Static libraries:

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

Statically Linked Shared Libraries

```
$ cc -Wall -c ldtest1.c ldtest2.c
$ ar -vq libldtest.a ldtest1.o ldtest2.o
$ ar -t libldtest.a
$ cc -Wall main.c libldtest.a

$ cc -Wall -c main.c
$ cc main.o -L. -lldtest -o a.out.dyn
$ cc -static main.o -L. -lldtest -o a.out.static
$ ls -l a.out.*
$ ldd a.out.*
$ nm a.out.dyn | wc -l
$ nm a.out.static | wc -l
```

Dynamically Linked Shared Libraries

Dynamic libraries:

- created by the compiler/linker (ie multiple steps)
- usually end in `.so`
- frequently have multiple levels of symlinks providing backwards compatibility / ABI definitions

Dynamically Linked Shared Libraries

```
$ rm *.o libldtest*
$ cc -Wall -c -fPIC ldtest1.c
$ cc -Wall -c -fPIC ldtest2.c
$ mkdir lib
$ cc -shared -Wl,-soname,libldtest.so.1 -o lib/libldtest.so.1.0 ldtest1.o ldtest2.o
$ ln -s libldtest.so.1.0 lib/libldtest.so.1
$ ln -s libldtest.so.1.0 lib/libldtest.so
$ cc -static -Wall main.o -L./lib -lldtest
[...]
$ cc -Wall main.o -L./lib -lldtest
[...]
$ ./a.out
[...]
$ ldd a.out
[...]
```

Dynamically Linked Shared Libraries

The link loader needs to know where to find all required shared libraries.

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./lib
$ ldd a.out
[...]
$ ./a.out
[...]
$ mkdir lib2
$ cc -Wall -c -fPIC ldtest1.2.c
$ cc -shared -Wl,-soname,libldtest.so.1 -o lib2/libldtest.so.1.0 ldtest1.2.o ldtest2
$ ln -s libldtest.so.1.0 lib2/libldtest.so.1
$ ln -s libldtest.so.1.0 lib2/libldtest.so
$ export LD_LIBRARY_PATH=./lib2:$LD_LIBRARY_PATH
$ ldd a.out  # note: no recompiling!
[...]
$ ./a.out
[...]
```

Dynamically Linked Shared Libraries

Avoiding `LD_LIBRARY_PATH`:

```
$ cc -Wall main.o -L./lib -lldtest -Wl,-rpath,./lib
$ echo $LD_LIBRARY_PATH
[...]
$ ldd a.out
[...]
$ ./a.out
[...]
$ unset LD_LIBRARY_PATH
$ ldd a.out
[...]
$ ./a.out
[...]
$
```

Lecture 10

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

Preprocessing

The compiler usually performs preprocessing (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 usually performs preprocessing (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 usually performs preprocessing (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 usually performs preprocessing (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 /lib64/ld-linux-x86-64.so.2 \
        /usr/lib/x86_64-linux-gnu/crt1.o         \
        /usr/lib/x86_64-linux-gnu/crti.o hello.o \
        -lc /usr/lib/x86_64-linux-gnu/crtn.o
$ file a.out
$ ./a.out
```

`cc(1)` and `ld(1)`

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

Different flags can be passed to `cc(1)` to be passed through to each tool as well as to affect all tools.

The order of the command line flags *may* play a role! Directories searched for libraries via `-L` and the resolving of undefined symbols via `-l` are examples of position sensitive flags.

The behavior of the compiler toolchain may be influenced by environment variables (eg `TMPDIR`, `SGI_ABI`) and/or the compilers default configuration file (MIPSPro's `/etc/compiler.defaults` or gcc's `specs`).

`gdb(1)`

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*

gdb(1)

```
$ cd student
$ make
$ ./a.out -lR ~djd >/dev/null
total 2701553
[...]
Memory fault
$ gdb ./a.out
 run -lR ~djd
Starting program: /home/jschauma/t/student/a.out -lR ~djd
total 2701553
[...]

Program received signal SIGSEGV, Segmentation fault.
0x000000000040214a in print_entries (entryvect=0x6050a0, options=0x605010) at print.c
221                     printf("%10s ", grpentry->gr_name);
```

gdb(1)

```
(gdb) bt
#0  0x000000000040214a in print_entries (entryvect=0x6050a0, options=0x605010) at pri
#1  0x0000000000401ad5 in list_dir_contents (path=0x7fffffffec59 "/home/djd", ftsopts
#2  0x000000000040292e in main (argc=1, argv=0x7fffffffe9e8) at ls.c:57
(gdb) li
216     } else {
217         pwentry = getpwuid(myentryp->fts_statp->st_uid);
218         printf("%8s ", pwentry->pw_name);
219
220         grpentry = getgrgid(myentryp->fts_statp->st_gid);
221         printf("%10s ", grpentry->gr_name);
222     }
223
224     // displaying the size
225     size = (long long)(myentryp->fts_statp->st_size);
(gdb) p grpentry
$1 = (struct group *) 0x0
```

gdb(1)

```
$ cc -g gdb1.c
$ ./a.out
[...]
$ gdb ./a.out
[...]
(gdb) break main
Breakpoint 1 at 0x400603: file gdb1.c, line 10.
(gdb) run
Starting program: /home/jschauma/t/gdb-examples/a.out

Breakpoint 1, main (argc=1, argv=0x7fffffffe9e8) at gdb1.c:10
10 c = fgetc(stdin);
(gdb) n
```

That’s all, folks!