UNIX SYSTEMS PROGRAMMING

SPRING 2021

Notes by:
Gerardo Torres

UNIX Systems Programming

Spring 2021

1. C Programming

File I/O

- Opening files

```
FILE *fopen(const char *path, const char *mode);
// r:   reading. Stream positioned at beginning.
// r+: read/write. Stream positioned at beginning.
// w:   write. Create or truncate.
// w+: read/write. Create or truncate
// a:   append. a+: append and read.
// Created files all have default permissions:
0666;
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH;
```

- Closing files

```
int fclose(FILE *fp);
// Return value: 0 for success
```

- Character input

```
int getc (FILE *fp);        // May be a “macro”
int fgetc (FILE *fp);       // Must be function
int getchar(void);          // get from stdin
// Return character or EOF
```

- Character output

```
Character output
int putc (int c, FILE *fp);     // May be a macro
int fputc (int c, FILE *fp);    // Must be function
int putchar(int c);             // Put to stdout
// Return character or EOF
```

- Line input

```
char *gets(char *buf);
// buf will not include the new-line character.
// caller has to make sure that the buffer is large enough.
// Removed in C11 (but gcc hasn't done so yet)
char *fgets(char *buf, int count, FILE *fp);
// buf will include the new-line character if it fits.
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
//n is the size of the buffer, not the length of the line. It includes a count
// for any newline, but not for a null character.
```

- Line output

```
int fputs(const char *buf, FILE *fp);   // Does not append a newline character
int puts(const char *buf);              // Appends a newline character
```

- Reading a file line-by-line

```
int main() {
    char *line = NULL;
    size_t line_len = 0;
    while (getline(&line, &line_len, stdin) > -1) {
        printf("%p: %s", line, line);
    }
    free(line);
}
```

Heap

- There isn't any `new` / `delete`. Heap space is allocated with:

```c
void *malloc(size_t size)
// Does not zero
void *realloc(void *ptr, size_t newsize)
// Does not zero
// Will literally extend the space if possible.
// Will handle copying to a larger memory block if could not extend.
void *calloc(size_t nobj, size_t size)
// Does zero. Note this is the only one that does.
```

Heap space is freed with

```c
void free(void *ptr) // No error value to check for
```

Parameter passing

- All parameter passing is by value. There is no pass by reference. But some people describe passing a pointer as "pass by reference".
- Swapping pointers:

```c
// To swap pointers, we change the location that a and b point to
void swapPtrThree(int **a, int **b) {
    int *tmp = *a;
    *a = *b;
    *b = tmp;
}

int main() {
    // ...
    swapPtrThree(&p, &q);
    printf("After swapPtrThree:\n");
    display(x, y, p, q);
}
```

- What if your function is not expecting any arguments?

> Specify void in the parameter list! Otherwise C will allow arguments to be passed!`

```c
void foo() {
    puts("Hello world");
}

void bar(void) {
    puts("Hello world");
}

int main() {
    foo();
    foo(17);    // Will compile!
    bar(17);    // Will not compile!
    return 0;
}
```

- C passes functions using function pointers.

```c
// Some function's definition:
void doNothing(void) {}

void takesFunction( void (*fp)(void) ) {
    (*fp)();    // "official" way
    fp();       // also accepted
    *fp();      // Oops! dereferences return value
}
int main() {
    takesFunction(doNothing);
}
```

Casting

- In C, to cast a value x to a type T:

```
(T)x
```

Errors

- For a readable error message, you can call

```
// Prints your message, if any, followed by a description of the error
void perror(const char *msg);
```

Environment

- Every process has a set of “environment” variables set up by the process’s creator. They are located above the call stack.
- You can see them with the command: `env`

```
env -i [name=value] ... [utility [arg ...]]
# -i says to ignore inherited environment
# Otherwise only replace specified names
# If no utility is provided, displays the resulting environment
```

- Displaying the environment

```
extern char **environ;

int main() {
    for (int index = 0; environ[index] != NULL; ++index) {
        puts(environ[index]);
    }
    for (char **p = environ; *p != NULL; ++p) {
        puts(*p);
    }
}
```

Static local variables

- Static local variables remember their values between calls.

```
/*
 * increment.c
 * Demonstrates a static local variable.
 */
int increment() {
    // the lifetime is from you first use it, up until the end of the
    // program (as opposed to up until the end of the function)
    static int value = 17;

    // it has a post-increment so it's gonna return 17 the first time
    // it's called
    return value++;
}

int main() {
    for (int i = 0; i < 10; ++i) {
        printf("%d ", increment());
    }
    printf("\n");
}
```

extern

- We've talked about `extern` variables before in the context of environment variables. Let's take another look at them.

```
/*
 * extern.c
 * If built by itself, this would be a linkage error.
 * Needs a file that provides a definition of x.
 */

#include <stdio.h>

extern int x;

int main() {
    printf("x: %d\n", x);
}
```

- Here, we tell the compiler that we're looking at `extern int x` somewhere else.

```
/*
 * define.X.c
 * Provides a definition for the varable x
 */

int x = 17;
```

- So for example, if we compile it with the above program, `defineX.c` there are no linkage errors.

```
/*
 * nostatic.c
 */

#include <stdio.h>

int x = 42;

int main() {
    printf("x: %d\n", x);
}
```

static

- When built alone, this of course works fine. When linked with compilation of `defineX.c` will result in a linkage error as `x` is double defined. Note that there is no conflict when `x` is `static`.

```
/*
 * static.c
 */

#include <stdio.h>

static int x = 42;

int main() {
    printf("x: %d\n", x);
}
```

- We've seen `static` before but in local variables which are treated with global lifetime. Here, we say that `x` is global but when compiled with `defineX.c` there is no linkage error so its definition is not visible or in conflict with any other file's. The version of `defineX.c` is ignored.

Struct

- Structs are in a different “namespace” than variables and functions. This allows you to have a variable and a struct with the same name. (Good idea?) It requires that you say that a type is a struct everywhere you use it.

```c
struct MyStruct {
    int x;
    int y;
};

int main() {
    struct MyStruct mine;
    mine.x = 42;
}
```

typedef

- C programmers and C libraries seem to prefer the more verbose approach

```c
typedef struct MyStruct {
    int x;
    int y;
} YourStruct;

int main() {
    struct MyStruct mine;
    mine.x = 42;
    YourStruct yours;
    yours.x = 17;
}
```

Macros

- A macro is a piece of code in a program that is replaced by the value of the macro. Macro is defined by `#define` directive. Whenever a macro name is encountered by the compiler, it replaces the name with the definition of the macro.

```c
// Macro definition
#define LIMIT 5

// Print the value of macro defined
printf("The value of LIMIT is %d", LIMIT);
```

- Some catches:

```c
#define square(x) x * x
square(5);      // 5 * 5
square(1+1);    // 1 + 1 * 1 + 1

#define square(x) ((x) * (x))
square(++a);    // ((++a) * (++a))
```

Long jump

- What if you want to “bail out” of a function and jump somewhere “up the call stack”? For example, `main` called `foo`, `foo` called `bar` and `bar` called `felix`. Now you want to return (perhaps due to an error) all the way back to some line in `main`.

```c
int setjmp(jmp_buf env);
// saves the current environment into the variable environment for later use
// If this macro returns directly from the macro invocation, it returns zero
// but if it returns from a longjmp() function call, then it returns the value
// passed to longjmp as a second argument.

void longjmp(jmp_buf env, int value);
// restores the environment saved by the most recent call to setjmp() macro in
// the same invocation of the program with the corresponding jmp_buf argument.
```

- Example

```c
int main () {
   int val;
   jmp_buf env_buffer;

   /* save calling environment for longjmp */
   val = setjmp( env_buffer );

   if( val != 0 ) {
      printf("Returned from a longjmp() with value = %s\n", val);
      exit(0);
```

```
    }

    printf("Jump function call\n");
    jmpfunction( env_buffer );

    return(0);
}

void jmpfunction(jmp_buf env_buf) {
    longjmp(env_buf, "tutorialspoint.com");
}
```

Output:

```
$ longjump
Jump function call
Returned from a longjmp() with value = tutorialspoint.com
```

2. Processes

- A **process** is an instance of an executing program. A **program** is a file containing a range of information that describes how to construct a process at run time. One program may be used to construct many processes, or, put conversely, many processes may be running the same program.
- We can re-define the definition of a process given at the start of this section as follows: a **process** is an abstract entity, defined by the kernel, to which system resources are allocated in order to execute a program. From the kernel's point of view, a process consists of
    - user-space memory containing program code and variables used by that code, and
    - a range of kernel data structures that maintain information about the state of the process.

Memory Layout

- The memory allocated to each process is composed of a number of parts, usually referred to as segments. These segments are as follows:
    - The **text segment** contains the machine-language instructions of the program run by the process.
    - The **initialized data segment** contains global and static variables that are explicitly initialized.
    - The **uninitialized data segment** contains global and static variables that are not explicitly initialized.
    - The **stack** is a dynamically growing and shrinking segment containing stack frames. One stack frame is allocated for each currently called function. A frame stores the function's local variables (so-called automatic variables), arguments, and return value.
    - The **heap** is an area from which memory (for variables) can be dynamically allocated at run time.
- The following shows various types of C variables along with comments indicating in which segment each variable is located.

```
char globBuf[65536];            /* Uninitialized data segment */
int primes[] = {2, 3, 5, 7};    /* Initialized data segment */

static int square(int x) {      /* Allocated in frame for square() */
    int result;                 /* Allocated in frame for square() */

    result = x * x;
    return result;              /* Return value passed via register */
}

static void doCalc(int val) {   /* Allocated in frame for doCalc() */
    printf("The square of %d is %d\n", val, square(val));

    if (val < 1000) {
        int t;                  /* Allocated in frame for doCalc() */
        t = val * val * val;
        printf("The cube of %d is %d\n", val, t);
    }
}

int main(int argc, char *argv[]) {  /* Allocated in frame for main() */
    static int key = 9973;          /* Initialized data segment */
    static char mbuf[10240000];     /* Uninitialized data segment */
    char *p;                        /* Allocated in frame for main() */

    p = malloc(1024);               /* Points to memory in heap segment */

    doCalc(key);
    exit(EXIT_SUCCESS);
```

```
}
```

- The following diagram shows the arrangement of the various memory segments on the `x86-32` architecture.

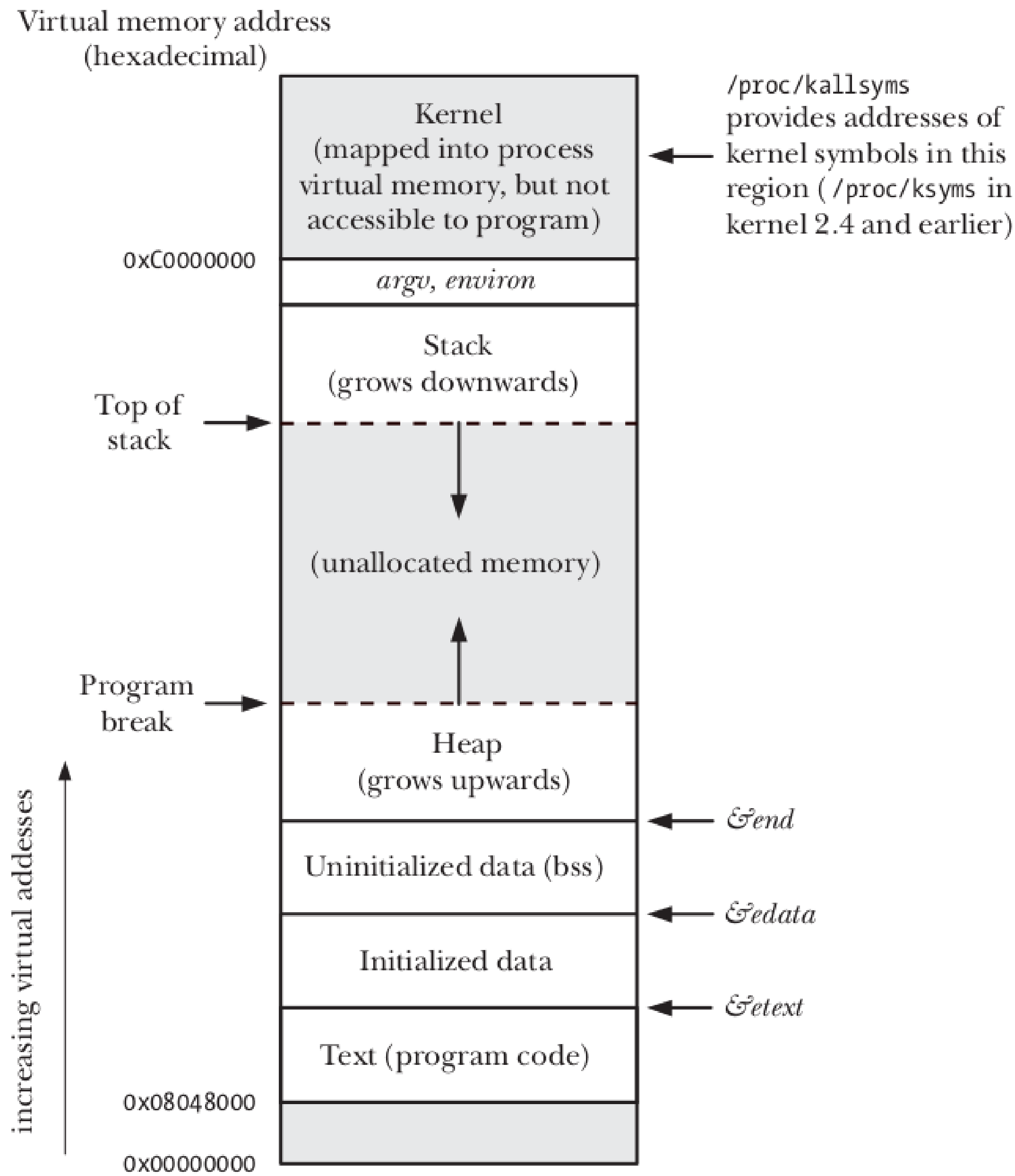


Figure 6-1: Typical memory layout of a process on Linux/x86-32

Environment

- Each process has an associated array of strings called the **environment list**, or simply the **environment**. Each of these strings is a definition of the form `name=value`. Thus, the environment represents a set of name-value pairs that can be used to hold arbitrary information. When a new process is created, it inherits a copy of its parent's environment. This is a primitive but frequently used form of interprocess communication—the environment provides a way to transfer information from a parent process to its child(ren).
- Within a C program, the environment list can be accessed using the global variable `char **environ`. Like `argv`, `environ` points to a NULL-terminated list of pointers to null-terminated strings. Figure 6 5 shows the environment list data structures as they would appear for the environment displayed by the `printenv` command above.

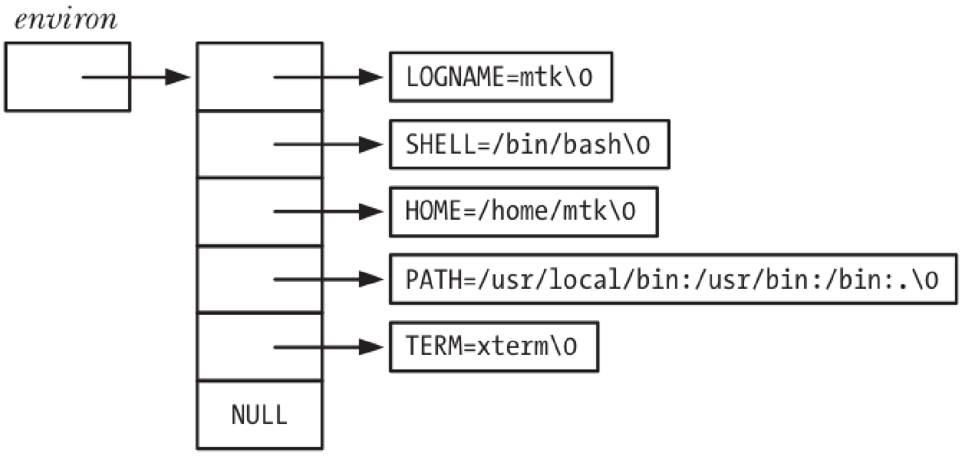


Figure 6-5: Example of process environment list data structures

Command line Arguments

- Every C program must have a function called `main()`, which is the point where execution of the program starts. When the program is executed, the command-line arguments (the separate words parsed by the shell) are made available via two arguments to the function `main()`. The first argument, `int argc`, indicates how many command-line arguments there are. The second argument, `char *argv[],` is an array of pointers to the command-line arguments, each of which is a null-terminated character string.

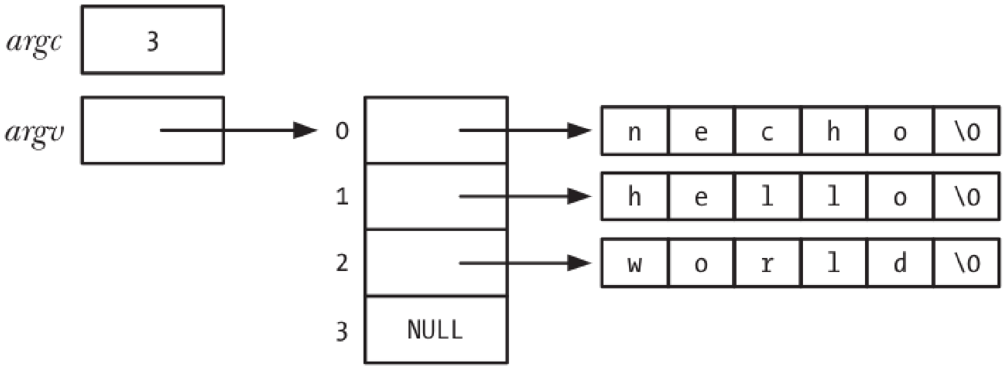


Figure 6-4: Values of *argc* and *argv* for the command *necho hello world*

Fork

- A process can create a new process using the `fork()` system call. The process that calls `fork()` is referred to as the parent process, and the new process is referred to as the child process. The kernel creates the child process by making a duplicate of the parent process. The child inherits copies of the parent's data, stack, and heap segments, which it may then modify independently of the parent's copies.
- Testing the child fork:

```c
int main() {
    pid_t parent_pid = getpid();
    printf("Parent id: %d\n", parent_pid);
    pid_t fork_pid = fork();

    if (fork_pid == -1)  perror("fork failed");

    pid_t new_pid = getpid();

    if (parent_pid == new_pid)
        printf("Yes, I (%d) am the parent & fork said %d\n", new_pid, fork_pid);
    else
        printf("No, I (%d) am not the parent & fork said %d\n", new_pid, fork_pid);
}
```

Exec

Wait

- Two major versions of wait:

```
// You call `wait()` to find out what the exit code of the process.
// The return value is the process ID of the process.
int wait(int * status);

// This has the advantage to wait for a specific process ID. Also,
// you can choose to not wait and you were just checking - `WNOHANG`
// which makes it a non-blocking call
int waitPID(int pid, int * status, const WNOHANG)
```

- Do I have any children processes that have terminated? I will wait until one of my children terminates. If I don't have any children, that's an error and `wait()` will return a `-1`.if I have children and none of them have terminated, `wait()` will block. It's a simple way of synchronization. The return of `wait()` will be the process id of the child that terminated.
- Definition for wait:

```
pid_t wait(int * stat_loc);
```

You pass in an `int` pointer, and if the `int` pointer is `null`, then it means i don't care. Otherwise, the system puts in the reason why it terminates (exit code, signals). The structure of this will be covered another time.

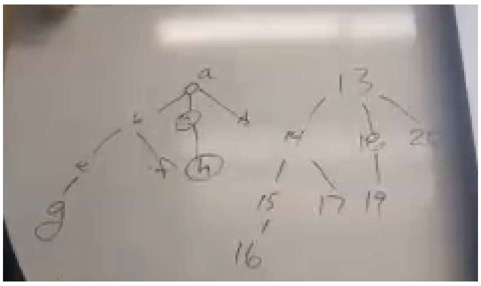
- Let' s see an example of wait

```
#include <sys/wait.h>   // wait()

const int N = 3;        // Default

int main(int  argc, char* argv[]) {
    if (argc > 2) {
        fprintf(stderr, "Use: loopFork [count] \n");
        exit(1);
    }

    // Allowing the default n to be overwritten.
    int n = (argc == 1) ? N: atoi(argv[1]);

    // Print the process id
    printf("pid: %d\n", getpid());

    for (int i = 0; i < n; ++i) {
        // If i'm the child, report who I am and who my parent is
        if (fork() == 0)
            printf("pid: %d; ppid: %d\n", getpid(), getppid());
    }
}
```

This is the output we see:

```
pid: 93913
pid: 93914; ppid: 93913
pid: 93915; ppid: 93914
pid: 93916; ppid: 93915
pid: 93917; ppid: 93914
pid: 93918; ppid: 93913
pid: 93919; ppid: 93918
pid: 93920; ppid: 93913
```

This is what the tree looks like (with only the last two digits showing in the tree):



There are 8 or 2^n total number of processes. If your goal was to create `n` children, clearly you didn't do that.

- What if we wanted to achieve having a fan out of `n` children? Here's a modification:

```c
#include <sys/wait.h>   // wait()

const int N = 3;        // Default

int main(int  argc, char* argv[]) {
    if (argc > 2) {
        fprintf(stderr, "Use: loopFork [count] \n");
        exit(1);
    }

    // Allowing the default n to be overwritten.
    int n = (argc == 1) ? N: atoi(argv[1]);

    // Print the process id
    printf("pid: %d\n", getpid());

    for (int i = 0; i < n; ++i) {
        if (fork() == 0) {
            break;
        } else {
            wait(NULL);
        }
    }
    // Report who I am and who my parent is after the loop
    printf("pid: %d; ppid: %d\n", getpid(), getppid());
}
```

Here's the output:

```
pid: 93943
pid: 93944; ppid: 93943
pid: 93945; ppid: 93943
pid: 93946; ppid: 93943
pid: 93943; ppid: 93942 # original process and parent (the shell itself)
```

- What if we want a chain going down with `n` descendants?

```c
#include <sys/wait.h>   // wait()

const int N = 3;        // Default

int main(int  argc, char* argv[]) {
    if (argc > 2) {
        fprintf(stderr, "Use: loopFork [count] \n");
        exit(1);
    }

    // Allowing the default n to be overwritten.
    int n = (argc == 1) ? N: atoi(argv[1]);

    // Print the process id
    printf("pid: %d\n", getpid());

    for (int i = 0; i < n; ++i) {
        if (fork() == 0) {
            // If you're the child, do your work, if you're the
            // parent, terminate (returning to main) or just exit(0)
            printf("pid: %d; ppid: %d\n", getpid(), getppid());
        } else {
            wait(0);
            return 0;
        }
    }
}
```

The output:

```
pid: 94051
pid: 94052; ppid: 94051
pid: 94053; ppid: 94052
pid: 94054; ppid: 94053
```

Exit

- There are two exits. There is `exit()` which is a C call and there is `_exit()` which is a UNIX call. `exit()` calls the lower level `_exit()` call but before that, it flushes its C stream buffers. After that, `_exit()` closes its file descriptors like standard out file and makes sure that all those get written out.
- `exit()` flushes the standard I/O buffers, file streams. The C library `exit()` calls the UNIX call `_exit()` which makes sure that all the UNIX file descriptors are all properly written out.

Orphans

- An example of `fork()` involving orphans:

```
pid_t parent_pid = getpid();
printf("Parent id: %d\n", parent_pid);

pid_t fork_pid = fork();
if (fork_pid < 0) {
    // Error checking
    perror("fork failed!");
    exit(1);
} else if (fork_pid > 0) {
    // If you're the parent, wait a little bit and terminate
    sleep(3);
    exit(0);
} else {
    // You are the child process
    printf("I am the child (%d). My parent is %d\n", getpid(), getppid());

    // Waiting for the parent to terminate - the only way you get
    // a different parent is if your parent terminates and you get
    // adopted by init
    while (getppid() == parent_pid) sleep(1);
    printf("Now I (%d) am an orphan! (Woe is me!)\n");
    printf("But I was adopted and my parent is %d\n", getpid(), getppid());
}
```

The output:

```
Parent id: 93846
I am the child (93848). My aparent is 93846
Now I (93848) am an orphan! (Woe is me!)
But I was adopted and my parent is 1
```

- There is a special process, `init()` that is the first process. All orphan processes get adopted by the `init()` process.
- When a process terminates, there's some information left behind: information about why it terminated. Is it because it turned exit? Or because somebody cit `ctrl-c`? Or another signal? You might wanna know these signals.

Zombies

- Example of a zombie

```
/*
 6_zombie.c
*/
int main() {
    pid_t parent_pid = getpid();
    printf("Starting id: %d\n", parent_pid);
    pid_t fork_pid = fork();

    if (fork_pid > 0) { // Parent sleeps
        sleep(20);
    } else {            // Child
        printf("I am the child (%d). My parent is %d\n", getpid(), getppid());
        sleep(10);
        printf("death...\n");
    }
}
```

The child is terminated but the parent is still alive because it sleeps for 20 seconds instead of the child's 10 seconds.

One shell:

```
$ 6_zombie
Starting id: 3451
I am the child (3452). My parent 3451
death...
```

Another Shell:

```
$ ps -j
PID     PPID        STAT    COMMAND
3451    3447        S+      6_zombie
3452    3451        S+      6_zombie
# you had a command 6_zombie and it forked off another
# child so it has the same command name, fine.
$ ps -j
PID     PPID        STAT    COMMAND
3451    3447        S+      6_zombie
3452    3451        Z+      (6_zombie)
# we waited until the child said "death...", and got a
# different formatting where the STAT of the child
# process is Z which means unscheduleable - its terminated
# and further with (6_zombie).
```

- It's no longer alive but it's still around (it has not had its parent wait for it) so it's called a zombie. If the parent never terminates, the child stays a zombie for eternity. But eventually, either the parent waits for the zombie, or the parent terminates and the zombie gets adopted by `init()`.
- Zombies can be a problem if there is a horde of them because a horde of them would mean that there was a whole horde of system information being kept about a lot of terminated processes and that's just taking up system resources. In principle, there are a fixed number of process IDs.

3. File System

Links

- Underneath the file system, files are represented by **inodes**. A file in the file system is basically a link to an inode. A **hard link**, then, just creates another file with a link to the same underlying inode. When you delete a file, it removes one link to the underlying inode. The inode is only deleted (or deletable/over-writable) when all links to the inode have been deleted.
- A **symbolic link** is a link to another name in the file system.
- Once a hard link has been made the link is to the inode. Deleting, renaming, or moving the original file will not affect the hard link as it links to the underlying inode. Any changes to the data on the inode is reflected in all files that refer to that inode.
- Hard links are only valid within the same File System. Symbolic links can span file systems as they are simply the name of another file.
- Create two files:

  ```
  $ touch foo; touch bar
  ```

  Enter some data into them:

  ```
  $ echo "Cat" > foo
  $ echo "Dog" > bar
  ```

  Let's create hard and soft links:

  ```
  $ ln foo foo-hard
  $ ln -s bar bar-soft
  ```

  Let's see what just happened:

  ```
  $ ls -l

  foo
  foo-hard
  bar
  bar-soft -> bar
  ```

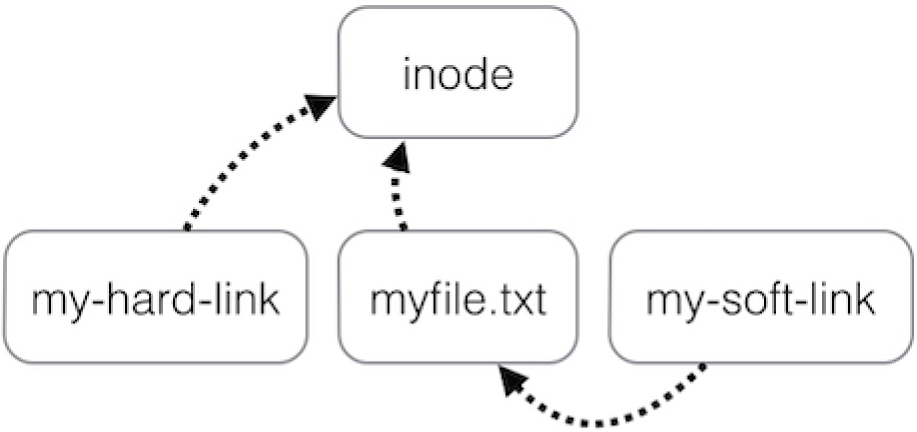
  Changing the name of `foo` does not matter:

  ```
  $ mv foo foo-new
  $ cat foo-hard
  Cat
  ```

`foo-hard` points to the inode, the contents, of the file - that wasn't changed.

```
$ mv bar bar-new
$ ls bar-soft
bar-soft
$ cat bar-soft
cat: bar-soft: No such file or directory
```

The contents of the file could not be found because the soft link points to the name, that was changed, and not to the contents. Likewise, If `foo` is deleted, `foo-hard` still holds the contents; if `bar` is deleted, `bar-soft` is just a link to a non-existing file.



stat

- `stat()` is a Unix system call that returns file attributes about an inode. As an example, Unix command ls uses this system call to retrieve information on files that includes:
    - `atime` : time of last access (`ls -lu`)
    - `mtime` : time of last modification (`ls -l`)
- `ctime` : time of last status change (`ls -lc`)
- Example program

```
/*
 fileinfo.c
 Display some information from the inode
 Uses stat function and struct stat
 */

int main(int argc, char *chargv[]) {
    struct stat info;

    if (argc != 2) {
        fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(FORMAT_ERROR);
    }

    stat(argv[1], &info);
    printf("    mode: %o\n", info.st_mode);         // type + mode
    printf("   links: %u\n", info.st_nlink);        // no of links
    printf(" user_id: %d\n", info.st_uid);          // user id
    printf("group_id: %d\n", info.st_gid);          // group id
    printf("    size: %d\n", (int)info.st_size);    // file size
    printf(" modtime: %d\n", (int)info.st_mtime);   // modified
    printf("    name: %s\n", argv[1]);              // filename

    return 0;
}
```

- What's one piece of information that is missing from inode? The filename! Why?

> it can have lots of different names, it's represented by the inode but the names are just whatever anyone wants to call them.

- How does `ls` work? What do you do the things it does?

> read info from inode (`stat`). read directories, they are special file types `opendir()` , `readdir()`

Inodes

- A file system's i-node table contains one i-node (short for **index node**) for each file residing in the file system. I-nodes are identified numerically by their sequential location in the i-node table. The information maintained in an i-node includes the following:
    - File type (e.g., regular file, directory, symbolic link, character device).
    - Owner (also referred to as the user ID or UID) for the file.
    - Group (also referred to as the group ID or GID) for the file.
    - Three timestamps:
        - time of last access to the file (shown by `ls –lu`),
        - time of last modification of the file (the default time shown by `ls –l`),
        - and time of last status change (last change to i-node information, shown by `ls –lc`). As on other UNIX implementations, it is notable that most Linux file systems don't record the creation time of a file.
    - Number of hard links to the file.
    - Size of the file in bytes.
    - Number of blocks actually allocated to the file, measured in units of 512-byte blocks. There may not be a simple correspondence between this number and the size of the file in bytes, since a file can contain holes (Section 4.7), and thus require fewer allocated blocks than would be expected according to its nominal size in bytes.
    - Pointers to the data blocks of the file.

Directories

- The `opendir()` function opens a directory and returns a handle that can be used to refer to the directory in later calls.

```
#include <dirent.h>

/* Returns directory stream handle, or NULL on error */
DIR *opendir(const char * dirpath );
```

  The `opendir()` function opens the directory specified by `dirpath` and returns a pointer to a structure of type DIR. This structure is a so-called directory stream, which is a handle that the caller passes to the other functions described below.

- The `readdir()` function reads successive entries from a directory stream.

```
#include <dirent.h>

/*
    Returns pointer to a statically allocated structure describing
    next directory entry, or NULL on end-of-directory or error
*/
struct dirent *readdir(DIR * dirp );
```

- Each call to `readdir()` reads the next directory from the directory stream referred to by `dirp` and returns a pointer to a statically allocated structure of type `dirent`, containing the following information about the entry:

```
struct dirent {
    ino_t d_ino;        /* File i-node number */
    char d_name[];      /* Null-terminated name of file */
};
```

  This structure is overwritten on each call to `readdir()`.

- Further information about the file referred to by `d_name` can be obtained by calling `stat()` on the pathname constructed using the `dirpath` argument that was specified to `opendir()` concatenated with (a slash and) the value returned in the `d_name` field.
- On end-of-directory or error, `readdir()` returns NULL , in the latter case setting `errno` to indicate the error. To distinguish these two cases, we can write the following:

```
errno = 0;
direntp = readdir(dirp);
if (direntp == NULL) {
    if (errno != 0) {
        /* Handle error */
    } else {
        /* We reached end-of-directory */
    }
}
```

Special Files

- If we want to filter out the error messages, we redirect standard error to `/dev/null`:

```
$ find / -name foo 2> /dev/null
```

- Read operations from `/dev/zero` return as many null characters (`0x00`) as requested in the read operation. Unlike `/dev/null`, `/dev/zero` may be used as a source, not only as a sink for data. All write operations to `/dev/zero` succeed with no other effects. However, `/dev/null` is more commonly used for this purpose.

4. UNIX I/O

File Descriptors

- When a file is opened or created by a process the kernel assigns a position in the array called the file descriptor.
- By convention Unix shells (although *not* the kernel) employ the following values:

File	File Descriptor	POSIX Constant
Standard Input	0	`STDIN_FILENO`
Standard Output	1	`STDOUT_FILENO`
Standard Error	2	`STDERR`

Basic Calls

- `open()`: open or create a file for reading or writing

```
int open(const char *path, int flags[, mode_t mode]);

// One of these three must be included:
O_RDONLY;   // open for reading only
O_WRONLY;   // open for writing only
O_RDWR;     // open for reading and writing

// The following are optional arguments
O_APPEND;   // append on each write
O_CREAT;    // create file if it does not exist: REQUIRES mode
O_TRUNC;    // truncate size to 0

open("pathtofile",O_WRONLY | O_CREAT, 0666);

// Returns file descriptor on success
```

- `close()`: detach the use of the file descriptor for a process

```
int close(int d);
// arguments    d: a file descriptor
// returns:     0 on success (the file descriptor deleted)
```

- `read()`: starts at the file's current offset, which is then offset by the number of bytes read

```
ssize_t read(int d, void *buf, size_t nbytes);
// arguments:   d: a file descriptor
//              buf: buffer for storing bytes read
//              nbytes: maximum number of bytes to read
// returns:     number of bytes read and placed in buf or 0 if end of file
```

- `write()`: starts at the file's current offset, which is then offset by the number of bytes written to the file

```
size_t write(int d, void *buf, size_t nbytes);
// arguments    d: a file descriptor
//              buf: buffer for storing bytes to be written
//              nbytes: maximum number of bytes to read
// returns:     number of bytes written
```

- `lseek()`: Every file descriptor has an associated *current file offset*, a number of bytes from the beginning of the file. Read and write operations normally start at the current offset and cause the offset to be incremented the number of bytes read or written. `lseek` explicitly repositions this offset value.

```
off_t lseek(int d, off_t offset, int base);
// arguments:       d: a file descriptor
//                  offset:  the number of bytes to be offset
//                  base:    the position from which the bytes will be offset:
//                  SEEK_SET:    offset bytes from beginning of the file.
//                  SEEK_CUR:    offset bytes from current value of offset.
//                  SEEK_END:    offset bytes from end of the file.
// returns:         The resulting offset location as measured in bytes from the beginning of the file.
```

umask

- `umask` - set file mode creation mask
- Setting `umask(077)` ensures that any files created by the program will only be accessible to their owner (0 in first position = all permissions potentially available) and nobody else (7 in second/third position = all permissions disallowed to group/other).

Pipes

- Create a pipe:

```
/*
createPipe.c
Create a pipe, write and read.

No error checking, for clarity.
*/

#include <string.h>
#include <unistd.h>

const int MAXLINE = 80;

int main() {
    char mesg[] = "Listen to me!\n";    // string to write
    char line[MAXLINE];                 // buffer to read into
    int fd[2];                          // file descriptors for pipe

    pipe(fd);                           // create the pipe

    write(fd[1], mesg, strlen(mesg));   // Send the mesage to pipe
    int count = read(fd[0], line, MAXLINE); // Read it back

    // Display the read msg to standard output
    write(STDOUT_FILENO, line, count);
}
```

dup and dup2

- `dup` and `dup2` duplicate the contents of an existing file descriptor. Remember, a file descriptor is the index of an array which contains a pointer to the file table. These functions allow a second file descriptor to index a pointer to the same file table. The difference is that `dup` takes a single argument, the file descriptor you want to duplicate, and returns a new file descriptor which is guaranteed to be the lowest available. `dup2` gives you more control over the new file descriptor: it takes two arguments, an already opened file descriptor and a new file descriptor, and directs the new file descriptor to point to the same file table. This is especially valuable when we want to create pipes between programs. If the new file descriptor is actually being used, `dup2` closes the file descriptor first, then reassigns it; if the two file descriptors are the same, nothing occurs.

Implementing I/O Redirection

- Do this:

```
/*
  duping.c
  demonstrates redirection of standard input by
  1) opening a file.
  2) closing file descriptor 0
  3) duping the original fd

```

```
  Note that error checking was omitted for readability.
  You should provide it!
 */

#include <unistd.h> // close
#include <fcntl.h>  // open
#include <stdlib.h> // exit

const int BUFFSIZE = 100;

int main() {
    char line[BUFFSIZE];

    // read from the console and print to the console
    // (same as in redirect.c)
    int n = read(0, line, BUFFSIZE-1);
    line[n] = '\n';
    write(1, line, n+1);

    // Open the file before redirecting
    int fd = open("/etc/passwd", O_RDONLY);

    // Redirect standard input to come from the file
    close(STDIN_FILENO);   // First closing standard input
    dup(fd);     // and then dup'ing the file descriptor into the
                 // lowest free fd. Returns new fd or -1.

    // and close the file's original descriptor as we don't want
    // unused open file descriptors.
    close(fd);

    // again read from standard input, now the file,
    // and print to the console. Same code as in the beginning.
    // (and same as in redirect.c)
    n = read(0, line, BUFFSIZE-1);
    line[n] = '\n';
    write(1, line, n+1);
}
```

- There is one issue: we're depending on the fact that when we `dup()` a file descriptor it's going to use the lowest available number. In this case, we can be confident that it'll be `0` because that's the one we're closing.
- But what if it were something other than `0` that we were talking about? Or what if some other portion of the program might be trying to make use of some other target file descriptor? How can we guarantee that we're going to duplicate our file descriptor into a particular one?

> There's one other call: `dup2()` which says "i have file descriptor that's open, i want to duplicate that file descriptor into this other one."

- Example using `dup2()`:

```
...
// Open the file before redirecting
// (same as duping.c)
int fd = open("/etc/passwd", O_RDONLY);

// redirect standard input to come from the file
// Now there's no ambiguity as to which fd to use as the target
dup2(fd, STDIN_FILENO);

// and close the file's original descriptor as we don't want
// unused open file descriptors.
// (same as duping.c)
close(fd);
...
```

- It says "okay you can have file descriptor `0`, if it's not already closed, i will close it for you". So `dup2()` closes the 2nd argument and duplicates the first argument into it.
- And that's all there is to redirecting I/O : fooling your system by saying that the file descriptor that you want to be using -- whether it's for input or output or stderr -- is actually a file.

Holes

- If the place we end up is before the beginning of the file or after the end of the file, what would you expect to have happened. For example, say seek 10 bytes before the beginning or 10 byes after the end.
    - seeking after the end of a file is perfectly legal. But then you end up with a **hole** which is a range of bytes that the file system doesn't allocate space for.
    - seeking before the start of a file is an error
- Why is seeking before the start of a file an error but not seeking past the end?

> A: It doen't make sense to rewrite the 10 direct block pointers because you'll have to move everything else over by 10 bytes.

5. Signals

What is a Signal?

- A signal is a notification to a process that an event has occurred. Signals are sometimes described as software interrupts. One process can (if it has suitable permissions) send a signal to another process. In this use, signals can be employed as a synchronization technique, or even as a primitive form of interprocess communication (IPC).
- Upon delivery of a signal, a process carries out one of the following default actions, depending on the signal:
    - The signal is **ignored**; that is, it is discarded by the kernel.
    - The process is **terminated** (killed).
    - A **core dump file** is generated, and the process is terminated. A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger.
    - The process is **stopped**—execution of the process is suspended.
    - Execution of the **process** is resumed after previously being stopped.

Signal Disposition

- Instead of accepting the default for a particular signal, a program can change the action that occurs when the signal is delivered. This is known as setting the disposition of the signal. A program can set one of the following dispositions for a signal:
    - The default action should occur, `SIG_DEF`.
    - If the disposition is set to `SIG_IGN`, then the signal is ignored
- UNIX systems provide two ways of changing the disposition of a signal: `signal()` and `sigaction()`.

```
void ( *signal(int sig , void (* handler )(int)) ) (int);
// Returns previous signal disposition on success, or SIG_ERR on error
```

Signal Handlers

- Signal demo 0

```
/*
 sigdemo0.c
*/
int main() {
    while(1) {
        puts("Hanging out");
        sleep(1);
    }
}
```

Shell:

```
$ sigdemo0
hanging out
hanging out
hanging out
^C             # Stops when we hit CTRL-C
```

- Signal Demo 1:

```
/*
 sigdemo1.c
 catches SIGINT
*/
```

```c
typedef void (*sig_hanlder_t) (int num);

/* Signal handler) */
void myHanlder(int signalNum) {
    puts("ouch!! Stop that !!!");
}

int main() {
    // Set handler for SIGINT, and remember the prior disposition
    sig_handler_t old_handler = signal(SIGINT, myHandler);

    // Hang out
    while(1) {
        puts("Hanging out");
        sleep(1);
    }

    // Restoring the original disposition for SIGINT
    signal(SIGINT, old_hadler);
}
```

Shell:

```
$ sigdemo1
Hanging out
Hanging out
Hanging out
^C ouch!! Stop that !!! # CTRL-C doesn't stop it
Hanging out
Hanging out
^\ Quit: 3              # but CTRL-\ does
```

It doesn't handle signal `3`.

- Signal Demo 2:

```c
void sigquit_handler(int signalNum) {
    puts("I won't quit!!!");
}

int main() {
    signal(SIGINT, SIGIGN);             // Ignore SIGINT
    signal(SIGQUIT, sigquit_hanlder);   // Handle ^\

    puts("you can't stop me!");

    // Hang out
    while(1) {
        puts("Haha!!!");
        sleep(1);
    }

}
```

Shell:

```
$ sigdemo1
you cant stop me!
Haha!!!
Haha!!!
Haha!!!
^C Haha!!!
Haha!!!
^\ i wont quit!
Haha!!!
Haha!!!
```

You have to do:

```
$ kill (process_id)
```

- Signal Demo 3:

```c
void sigquit_handler(int signalNum) {
    puts("I won't quit!!!");
}

void siqterm_Handler(int signalNum) {
    puts("cant kill be that easily!!!");
}

int main() {
```

```
    signal(SIGINT, SIGIGN);             // Ignore SIGINT
    signal(SIGQUIT, siqquit_hanlder);   // Handle ^\
    signal(SIGTERM, siqterm_Handler):   // Handle $ kill (process)

    puts("you can't stop me!");

    // Hang out
    while(1) {
        puts("Haha!!!");
        sleep(1);
    }

}
```

Sending a Signal

- Universal solution for killing any process:

```
$ kill -KILL (process_id)
or
$ kill -9 (process_id)
```

- Kill commands:

```
$ kill -l
 1) SIGHUP      2) SIGINT    3) SIGQUIT     4) SIGILL     5) SIGTRAP
 6) SIGABRT     7) SIGBUS    8) SIGFPE      9) SIGKILL   10) SIGUSR1
11) SIGSEGV    12) SIGUSR2 13) SIGPIPE     14) SIGALRM  15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD 18) SIGCONT     19) SIGSTOP  20) SIGTSTP
21) SIGTTIN    22) SIGTTOU 23) SIGURG      24) SIGXCPU  25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF 28) SIGWINCH    29) SIGIO    30) SIGPWR
31) SIGSYS
```

- Sending Signals in C

```
int kill(pid_t pid, int sig);
// The pid argument identifies one or more processes to which the signal
// specified by sig is to be sent.
//      If pid is greater than 0, the signal is sent to the process with the
//      process ID specified by pid.
//      If pid equals 0, the signal is sent to every process in the same process
//      group as the calling process, including the calling process itself.
```

- **Slow I/O**: Any operation that can block indefinitely.
    - Reading from a pipe. reading from `stdin`, reading from network.
    - Writing to something that's full (pipes)
    - Opening a FIFO

Signal Blocking

- We can do something else with signals: if we don't want to handle it *right* away but also don't want to ignore it forever. We can block the signal until we can handle it. We can block a signal by setting up a signal mask.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

What is `int how`?

- `SIG_BLOCK`: the set of blocked signals is the union of the current set and the *set* argument.
- `SIG_UNBLOCK`: the signals in *set* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- `SIG_SETMASK`: the set of blocked signals is set to the argument *set*.

What is `sigset_t *oldset`?

- a pointer to the current set before modifying so we can put it back when you're done.

`sigset_t` is the set of the 31 signals.

```
int sigemptyset(sigset_t*);     // clear all
int sigfillset(sigset_t*);      // set all
```

Long Jumps

- Q: what's the signal mask after a long jump?

  > A: whatever it was just before the long jump

  Q: is that what you want?

  > A: to give you a choice when setting up the jump buffer, we introduce `sigsetjmp` and `siglongjmp`.

Implementing sleep

- Initial code:

  ```
  static void sig_alrm(int signo) {}

  unsigned int toDream(unsigned int nsecs) {
      signal(SIGALRM, sig_alrm);
      alarm(nsecs);       // start the timer
      pause();            // next caught signal wakes us up
      return alarm(0);    // return timer, return unslept time
  }
  ```

  Q: What happens if there was already an alarm set?

  > A: If there was an alarm set and its further along, you might want to remember how much further it is, set up your own alarm, deal with it, and turn the other alarm back on with the correct amount of time.
  >
  > Another would be to save the disposition and put it back to what it was before we return.

  Q: What about the race condition?

  > A: Say I want an alarm in 2 seconds. Before you get to call `pause()` you get a context switch, and for some weird reason, you don't get back scheduled to run again for more than 2 seconds. While you're not running, the `SIGALRM` got send to you. So when you're finally scheduled, your handler fires off to do what it's supposed to do before you even had a chance to run `pause()`. You then call `pause()` and you're gonna block forever. It's an odd case but in principle, it can happen.

- Fixing the race condition

  We have to figure out what this process is currently blocking on; we don't want to change that. We then want to add into that set `SIGALRM`.

  ```
  // Before we set our alarm, block SIGALRM
  sigemptyset(&newmask);
  sigaddset(&newmask, SIGALRM);

  // Add SIGALRM, remembering prior mask state
  sigprocmask(SIG_BLOCK, &newmask, &oldmask);

  // Set the alarm, while blocking SIGALRM
  alarm(nsecs);

  // Make sure SIGALRM won't be blocked while suspended.
  suspmask = oldmask;
  sigdelset(&suspmask, SIGALRM);

  // Wait for any signal to be caught that wasn't being blocked before.
  // This function is the same as pause but it gies us a chance to pass
  // in an argument the address of a signal mask.
  sigsuspend(&suspmask);
  ```

6. UNIX Command Line

Common Utilities

- `ls` – list directories
- `mkdir`, `rmdir` – make or remove a directory
- `cd`, `pwd` – change / print the current working directory
- `cp` – make a copy of a file
- `mv` – move/rename a file
- `rm` – remove a file
- `cat` – concatenate files, also used to display a short file
- `more` (or less) – display a [large] file, one screen at a time.
- `man` – look up manual pages. Use it often! (Also, try info.)

Less Common but still important

- `gcc` – compile a C program
- `make` – build a project
- `grep` – find lines that match a regular expression
- `gdb` – debugger
- `find` – where did I put it?
- `chmod` – change file permissions
- `echo` – print to stdout
- `diff` – differences between files

Globbing - used to match file names

- An asterisk matches any string
  - `ls *.c`
- A question mark matches a single character
  - `ls ?.c`
- Square brackets can specify a "character class":
  - `ls [abcd]*.c`

I/O Redirection: input

- Simple:

  ```
  $ mail jsterling@poly.edu < Jabbercoky.txt
  ```

- "Here docs"

  ```
  $ mail jsterling@poly.edu << blah
  this is the stuff before blah
  and more stuff
  blah
  ```

I/O Redirection: output

- Standard Output

  ```
  $ ls > outputfile  # replaces outputfile
  $ ls >> outputfile # appends to outputfile
  ```

- Standard error

  ```
  $ myProgram 2> errorFile
  ```

- Standard output *and* standard error to the same file. Might *try* one of:

  ```
  $ myProgram 2> aFile > aFile
  $ myProgram > aFile 2> aFile
  ```

  But these don't work. Correct way is either:

```
$ myProgram > aFile 2>&1
$ myProgram &> aFile
```

Piping

- Piping allows the output of one process to be fed into another, in the shell `|` is used. If a command has a lot of output, feed it to `more` or `less`:

  ```
  $ dmesg | less
  $ dmesg | more
  ```

- Getting a count of lines:

  ```
  ps ax | wc -l
  ```

- Spell checker:

  ```
  $ tr -d '.,;:"\!\[]()?' | tr '\t' ' ' | tr ' ' '\n' \
      | tr '[A-Z]' '[a-z]' \
      | sort | uniq | comm -23 - /user/dict/words
  ```