

CS631 - Advanced Programming in the UNIX Environment

—

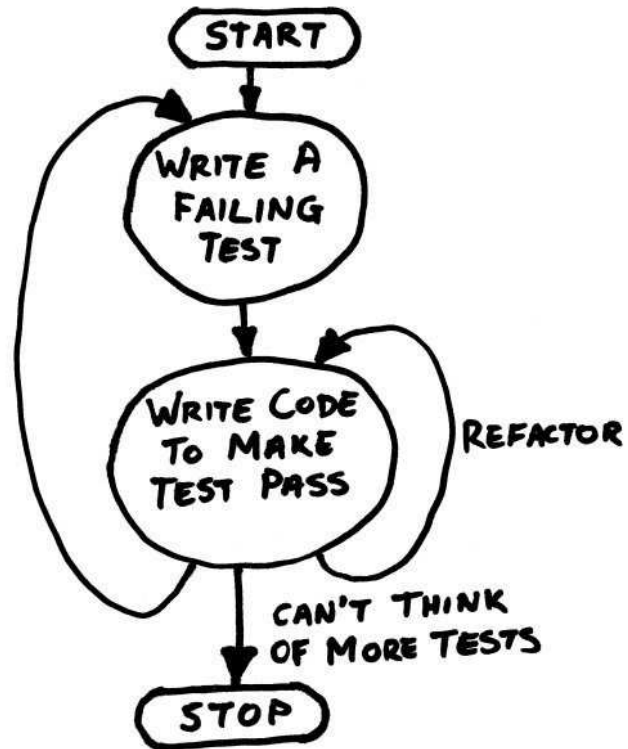
Process Environment, Process Control

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann

`jschauma@stevens.edu`

`http://www.cs.stevens.edu/~jschauma/631/`

Test Driven Development



Avoid useless comments.

```
/* The number of the arguments should be 3. */  
if (argc < 3) {  
  
    int sdf, ddf; /* src, dst file descriptor */  
  
    progame = argv[0]; /*save the program name for the program*/  
  
    char buf[BUFFERSIZE]; //a buffer to transfer data from source to target
```

Commenting code should only be necessary when you have to explain something that is not obvious from reading the code itself.

Dynamic memory allocation...

...you're doing it wrong:

```
char buf[BUFFERSIZE]; //a buffer to transfer data from source to target
```

Use `malloc(3)` - which can fail.

Really, believe me.

Dynamic memory allocation...

```
dstExpanded = (char*)malloc(maxFileLength);  
memset(dstExpanded, maxFileLength, 0);  
  
buff = (char*)malloc(blksize * sizeof(char));  
sfd = open(source, O_RDONLY);
```



Stevens CS631 APUE
@cs631apue



Seriously, malloc(3) can fail. Why won't you believe me?

↩ Reply ↻ Retweet ★ Favorite ⋮ More

Make sure to check all return codes.

Instead of this:

```
newdir= (char*)malloc(len+1);  
strncpy(newdir, buf, len+1);
```

Do this:

```
if ((newdir = (char*)malloc(len+1)) == NULL) {  
    fprintf(stderr, "Unable to allocate memory: %s\n",  
              strerror(errno));  
    exit(EXIT_FAILURE);  
}  
strncpy(newdir, buf, len+1);
```

Use *either* spaces *or* tabs; don't mix whitespace.

Instead of:

```
else{
input = argv[1];
output = argv[2];
    //check if the input file exists
    input_fd = open(input, O_RDONLY);
    if(input_fd<0){
```

...do this:

```
else{
    input = argv[1];
    output = argv[2];
    //check if the input file exists
    input_fd = open(input, O_RDONLY);
    if(input_fd<0){
```

Use `strerror(3)` and print error messages to `stderr`.

Instead of this:

```
if(lstat(argv[1], &s_stat)< 0) {
    if(errno == ENOENT){
        printf("%s, No such file or directory!\n",argv[1]);
        exit(1);
    }else{
        printf("%s lstat error!\n", argv[1]);
        exit(1);
    }
}
```

Do this:

```
if(lstat(argv[1], &s_stat)< 0) {
    fprintf(stderr, "Unable to stat %s: %s\n",
            argv[1], strerror(errno));
    exit(EXIT_FAILURE);
}
```


Control indentation and improve logic flow.

```
if (argc == 3) {
    [...]
    if ( !lstat(srcName,&srcStat)) {
        if(S_ISREG(srcStat.st_mode)) {
            [...]
            if( (tgtFd=open(tgtName, tgtFlags, srcStat.st_mode)) < 0) {
                if ( !lstat(tgtName,&tgtStat) ) {
                    [...]
                } else {
                    quit("error - could not stat...");
                }
            } else {
                quit("error - could not open...");
            }
        } else {
            quit("error - not a regular file");
        }
    } else {
        quit("error - could not stat");
    }
} else {
    quit("wrong number of arguments");
}
```

Control indentation and improve logic flow.

```
if (argc != 3) {
    usage();
}

if (!lstat(srcName,&srcStat)) {
    quit("%s: Unable to lstat %s: %s\n", getprogname(), srcName,
        strerror(errno));
}

if (!S_ISREG(srcStat.st_mode)) {
    quit("%s: %s is not a regular file.", getprogname(), srcName);
}

if ((tgtFd=open(tgtName, tgtFlags, srcStat.st_mode)) < 0) {
    quit("%s: Unable to open %s: %s\n", getprogname(), tgtName,
        strerror(errno));
}

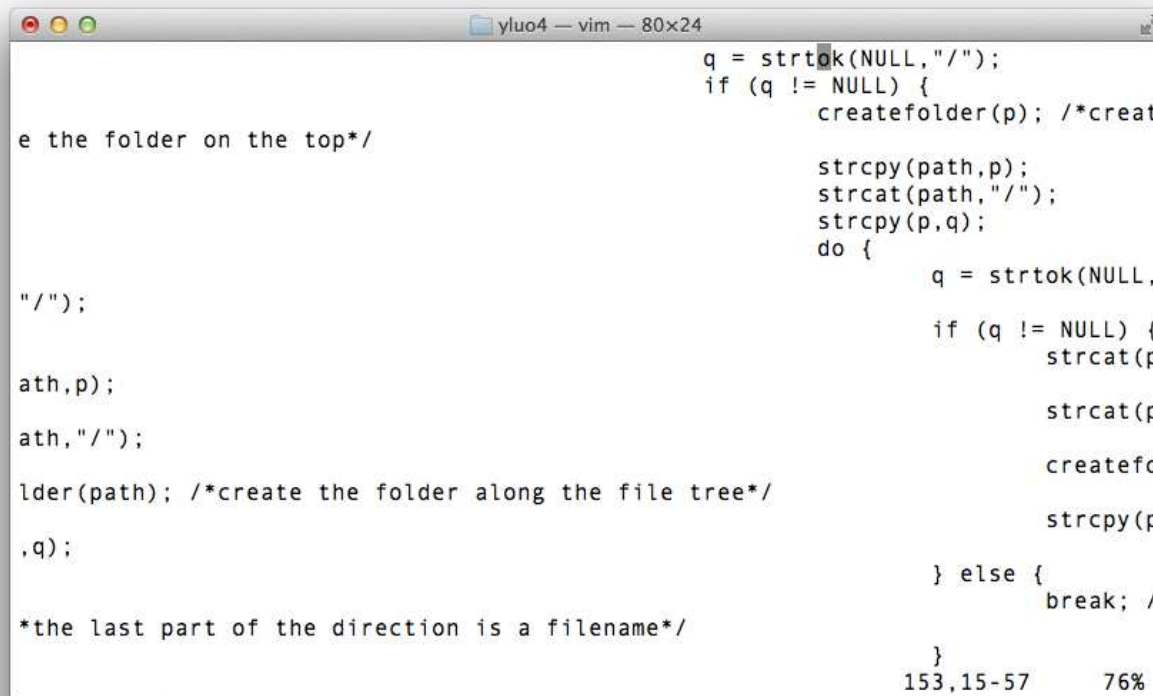
if ( !lstat(tgtName,&tgtStat) ) {
    quit("%s: Unable to lstat %s: %s\n", getprogname(), tgtName,
        strerror(errno));
}
```

Be consistent in your style.

Don't mix whitespace/tab, braces on line or on next, ...

```
if ((infile = open(name1, O_RDONLY)) == -1) return (1);
stat(name2, &to_stat);
if(S_ISDIR(to_stat.st_mode))//dir operation
{
    strcat(temp,name2);
    strcat(temp,slash);
    strcat(temp,name1);
    if((outfile = open(temp, O_WRONLY | O_CREAT, mode)) == -1)
    {
        fprintf(stderr, "Can't create this file\n");
close(infile);
        exit(1);
    }
}
else{
```

Standard terminal size is 80x24.



The screenshot shows a vim editor window titled 'yluo4 — vim — 80x24'. The code is written in C and is partially obscured by a vertical scrollbar on the right. The visible code includes comments and function calls for creating a directory tree. The code is as follows:

```
q = strtok(NULL, "/");
if (q != NULL) {
    createfolder(p); /*creat

e the folder on the top*/

    strcpy(path,p);
    strcat(path,"/");
    strcpy(p,q);
    do {
        q = strtok(NULL,
        if (q != NULL) {
            strcat(p
            strcat(p
            createfo
            strcpy(p
        } else {
            break; /
    }
}
153,15-57    76%
```

Remember:

Readability counts.

The `main` function

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

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 Creation

On Linux:

```
$ cc -Wall entry.c
```

```
$ readelf -h a.out | more
```

ELF Header:

```
[...]
```

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

```
$ objdump -d a.out
```

```
[...]
```

```
0000000000400460 <_start>:
```

400460:	31 ed	xor	%ebp,%ebp
400462:	49 89 d1	mov	%rdx,%r9

```
[...]
```

```
$
```


Process Creation

On Linux:

```
$ cc -e foo entry.c
```

```
$ ./a.out
```

```
Foo for the win!
```

```
Memory fault
```

```
$ cc -e bar entry.c
```

```
$ ./a.out
```

```
bar rules!
```

```
$ echo $?
```

```
1
```

```
$ cc entry.c
```

```
$ ./a.out
```

```
Hooray main!
```

```
$ echo $?
```

```
13
```

```
$
```

Process Termination

There are 8 ways for a process to terminate.

Normal termination:

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

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)

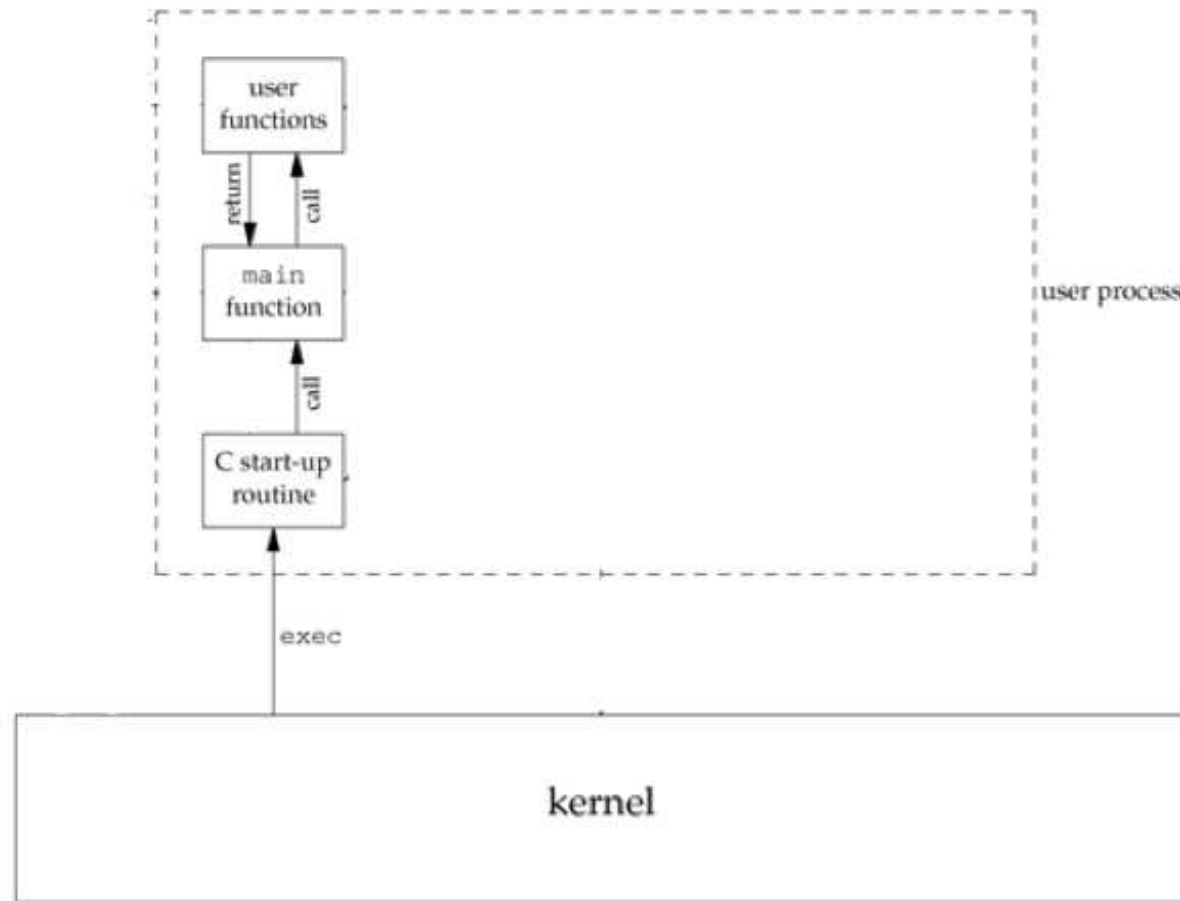
```
#include <stdlib.h>

int atexit(void (*func)(void));
```

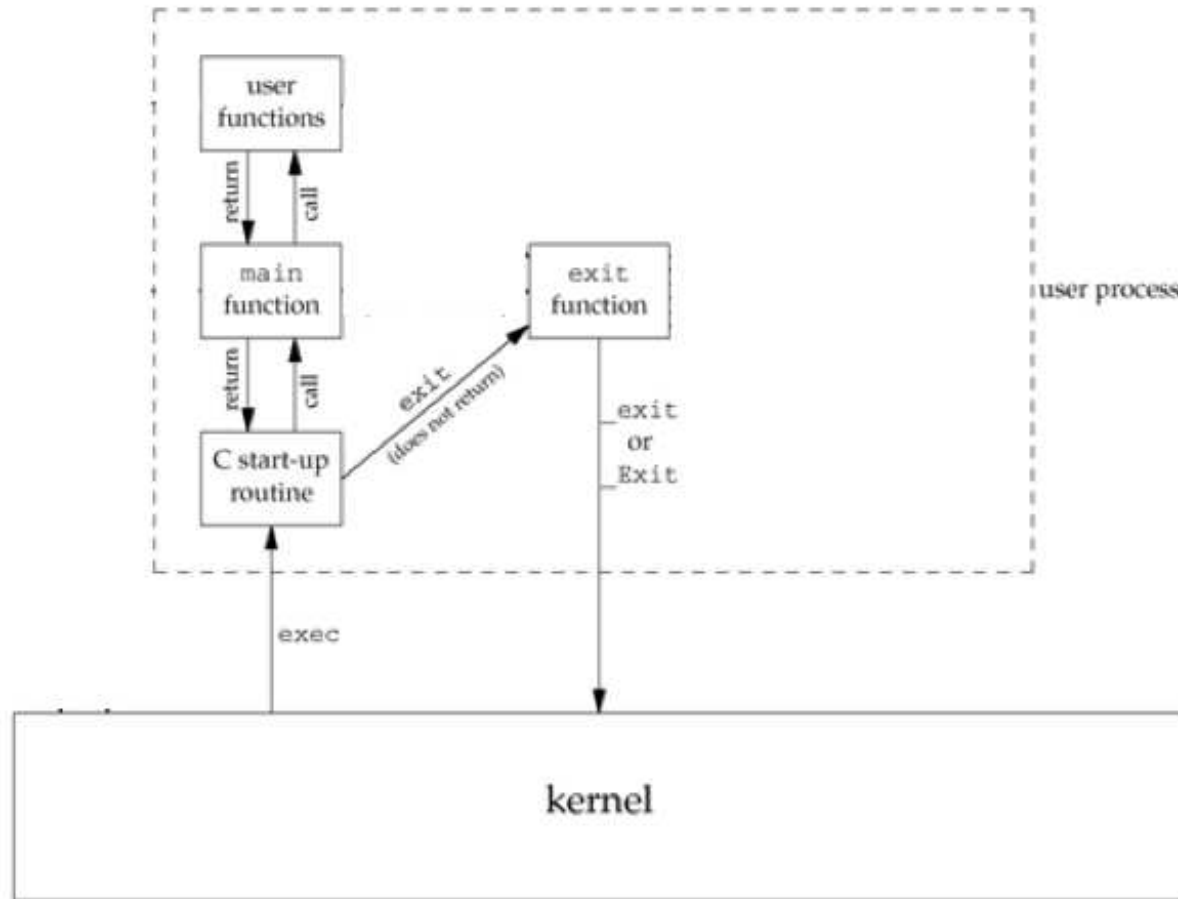
- Registers a function with a signature of `void funcname(void)` to be called at exit
- Functions invoked in reverse order of registration
- Same function can be registered more than once
- Extremely useful for cleaning up open files, freeing certain resources, etc.

`exit-handlers.c`

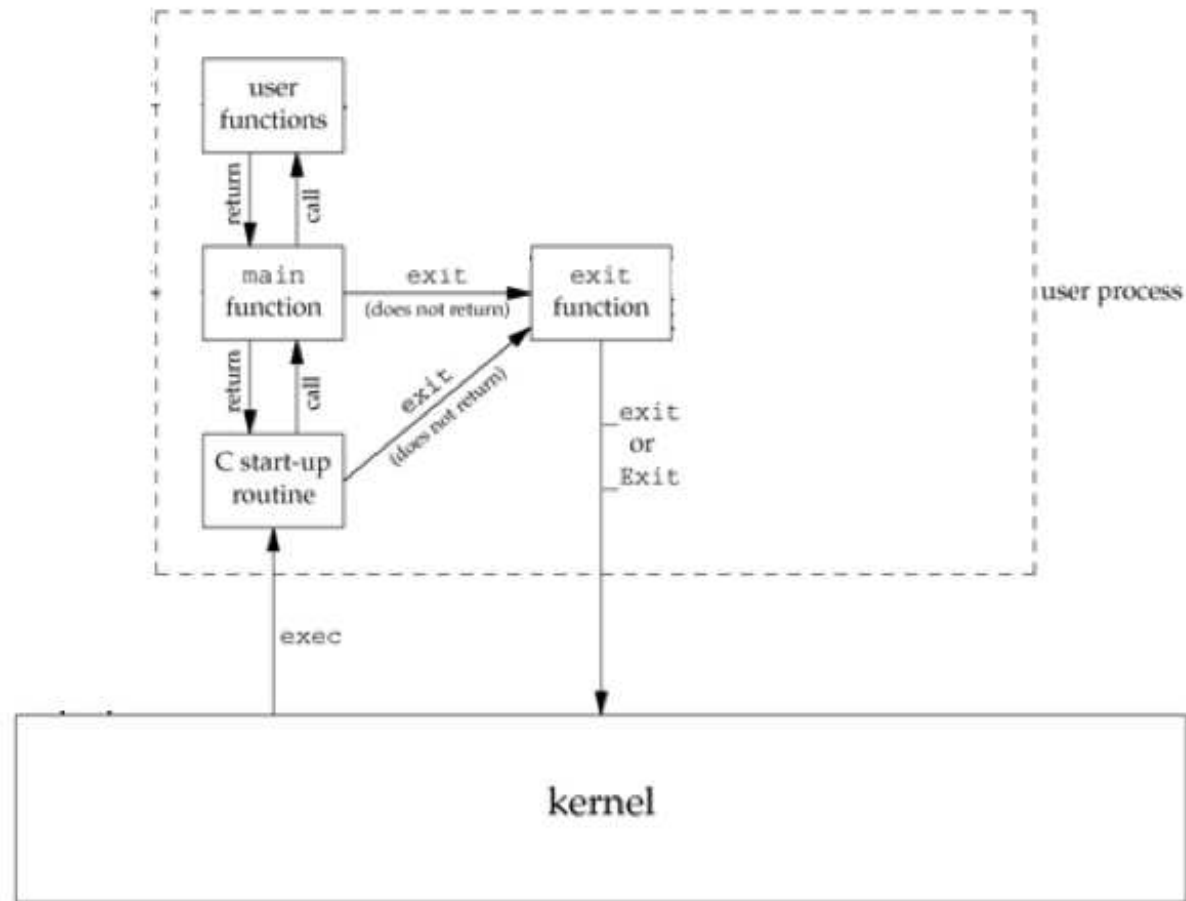
Lifetime of a UNIX Process



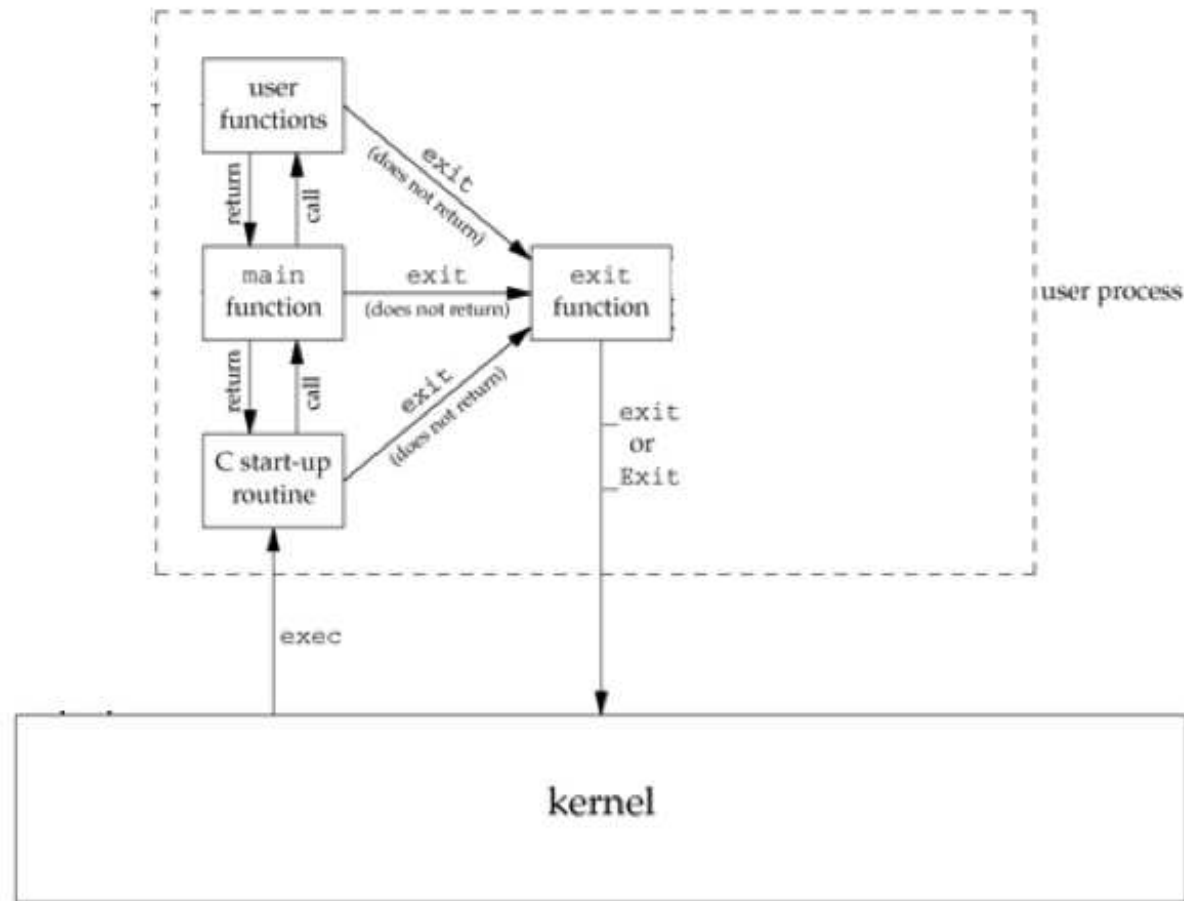
Lifetime of a UNIX Process



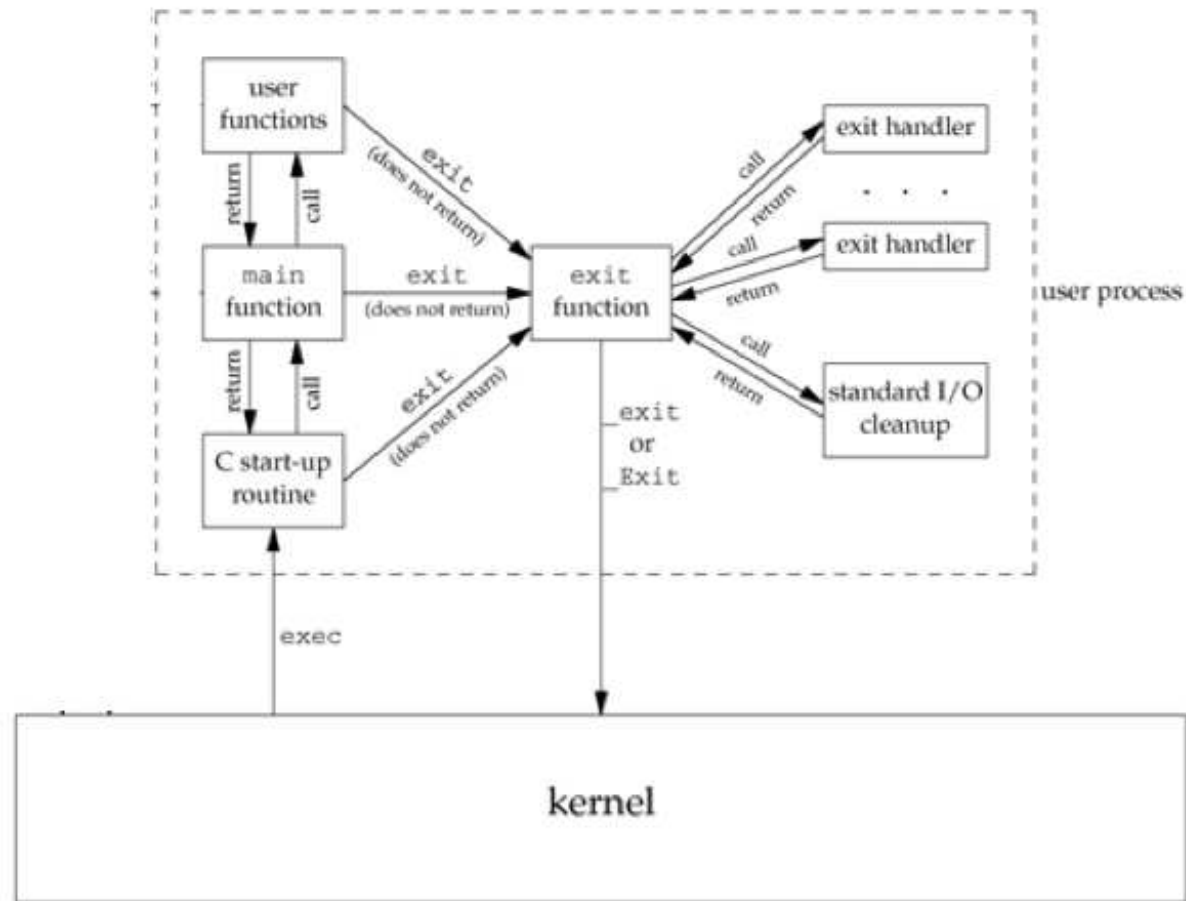
Lifetime of a UNIX Process



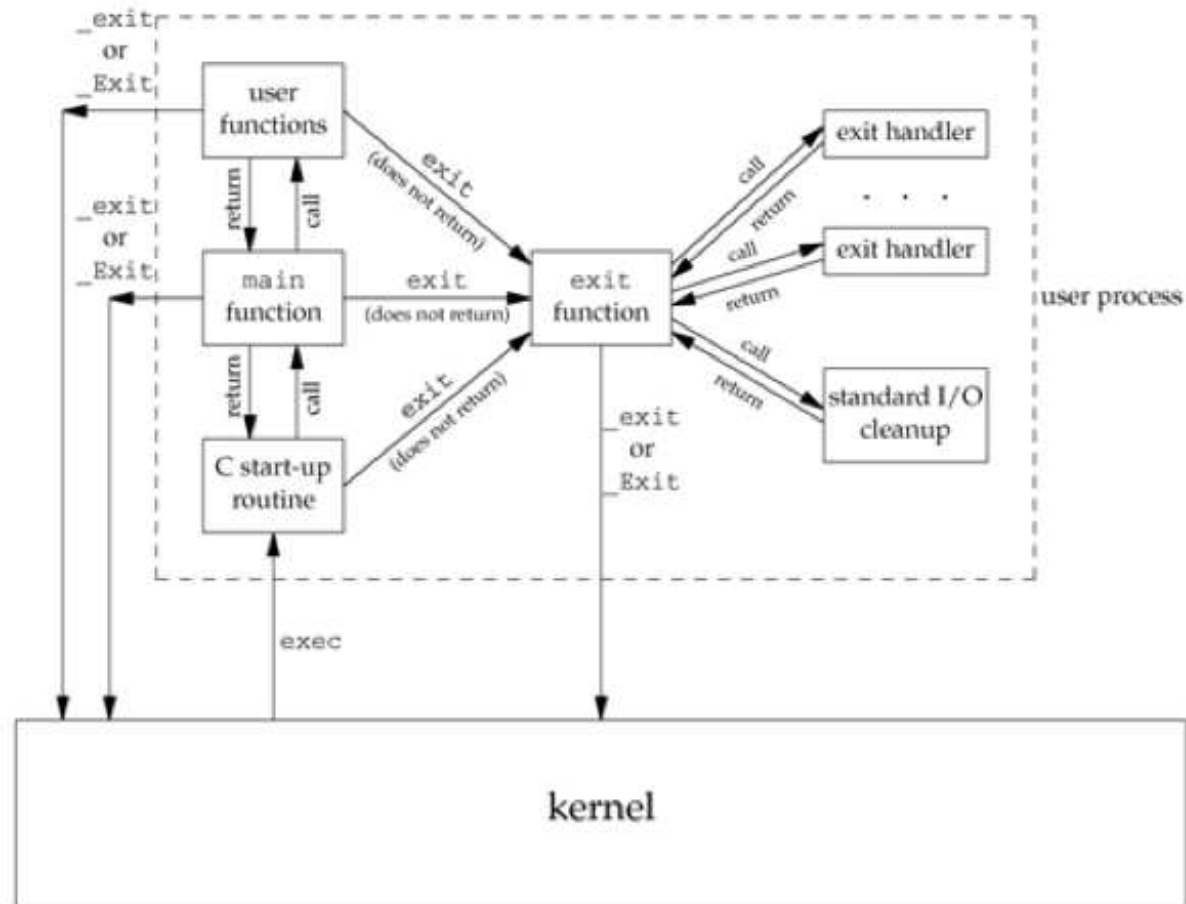
Lifetime of a UNIX Process



Lifetime of a UNIX Process



Lifetime of a UNIX Process



Exit codes

```
$ cc -Wall hw.c
hw.c: In function 'main':
hw.c:7: warning: control reaches end of non-void function
$ ./a.out
Hello World!
$ echo $?
10
$
```

Exit codes

```
$ cc -Wall hw.c
hw.c: In function 'main':
hw.c:7: warning: control reaches end of non-void function
$ ./a.out
Hello World!
$ echo $?
10
$ cc -Wall --std=c99 hw.c
$ ./a.out
0
$
```

Environment List

Environment variables are stored in a global array of pointers:

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

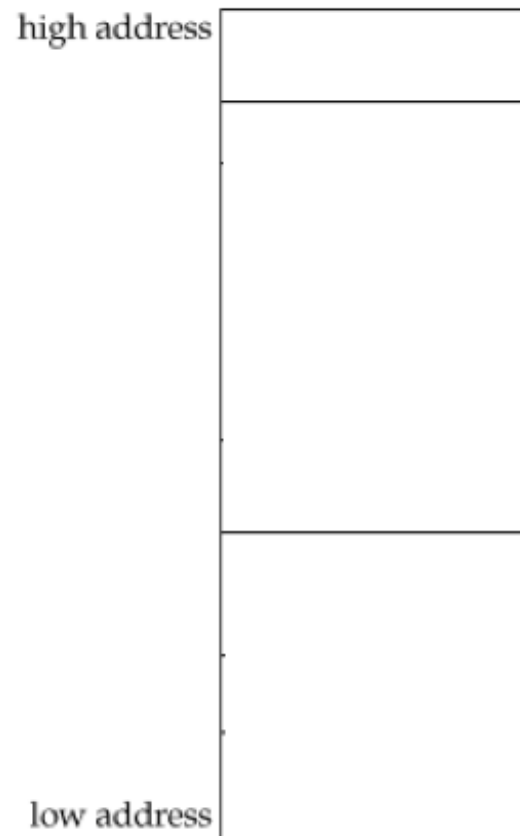
The list is `null` terminated.

These can also be accessed by:

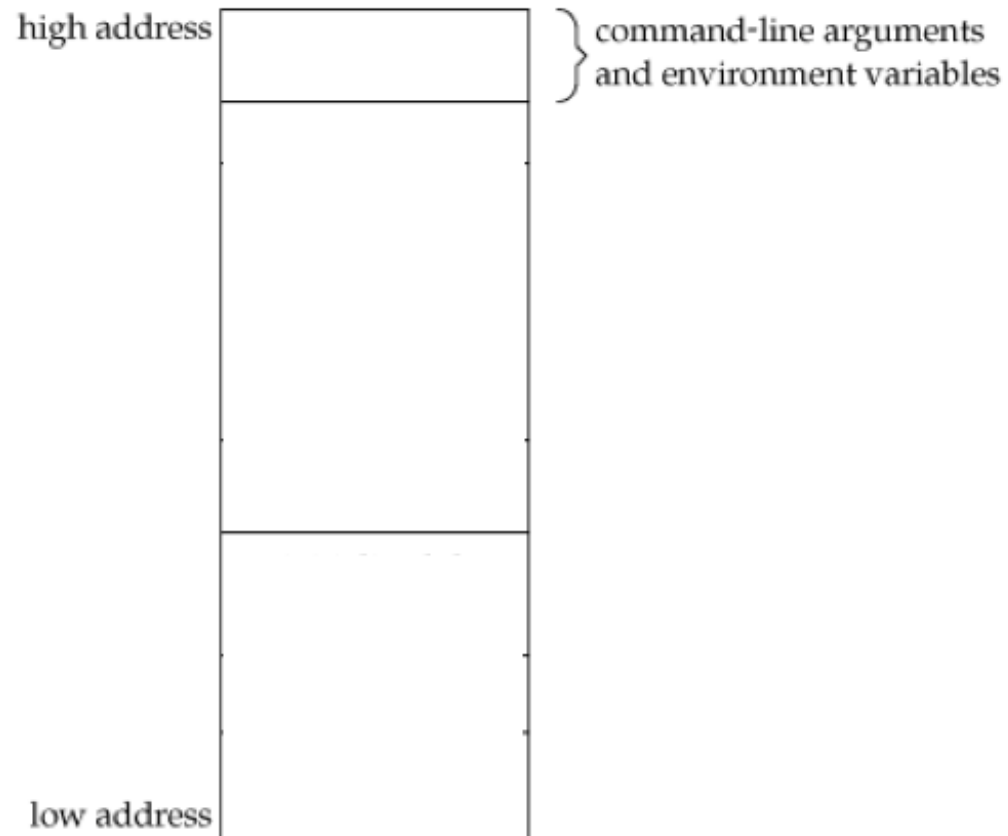
```
#include <stdlib.h>

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

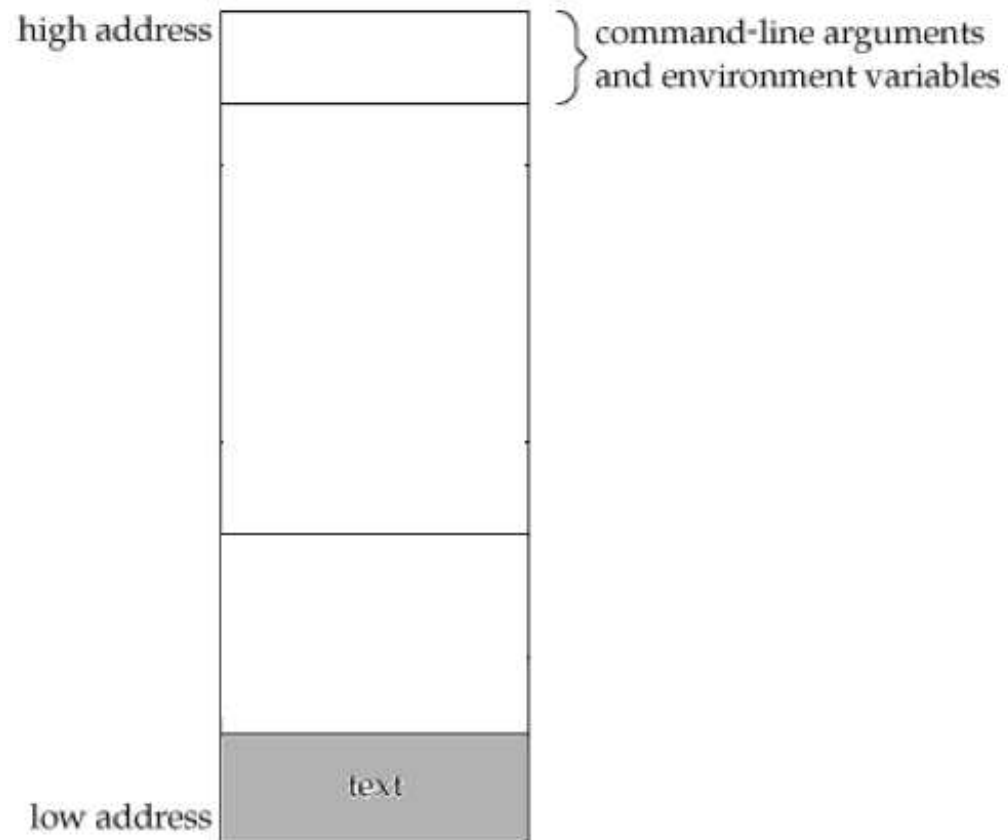
Memory Layout of a C Program



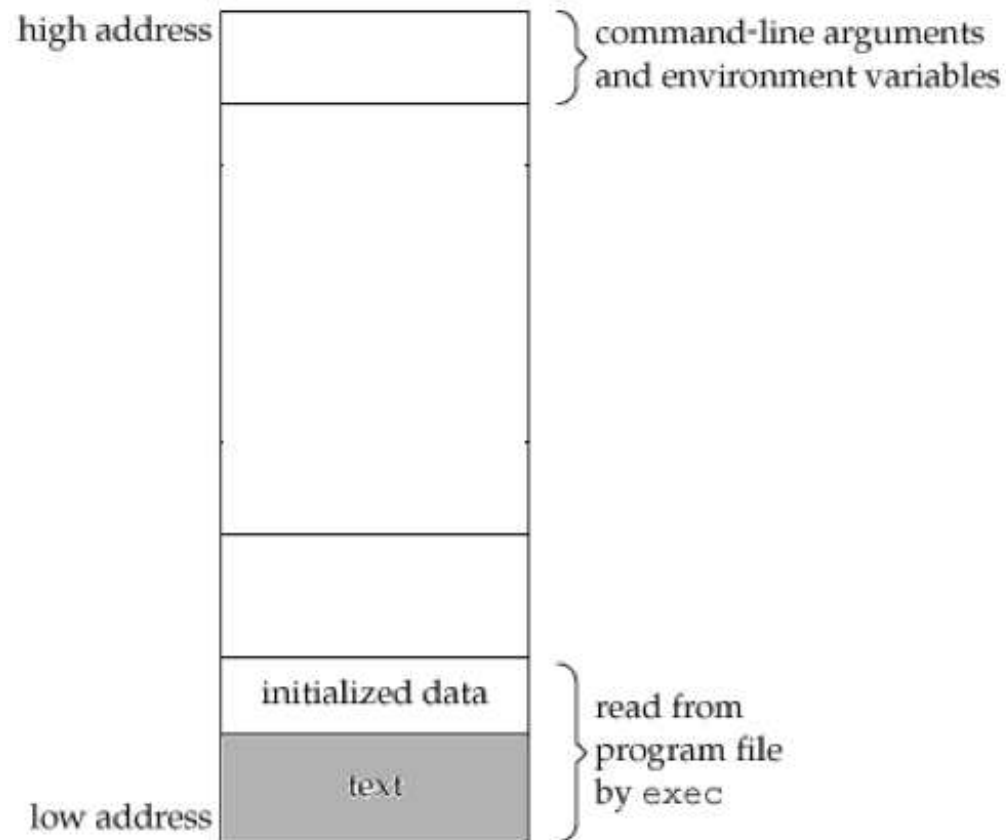
Memory Layout of a C Program



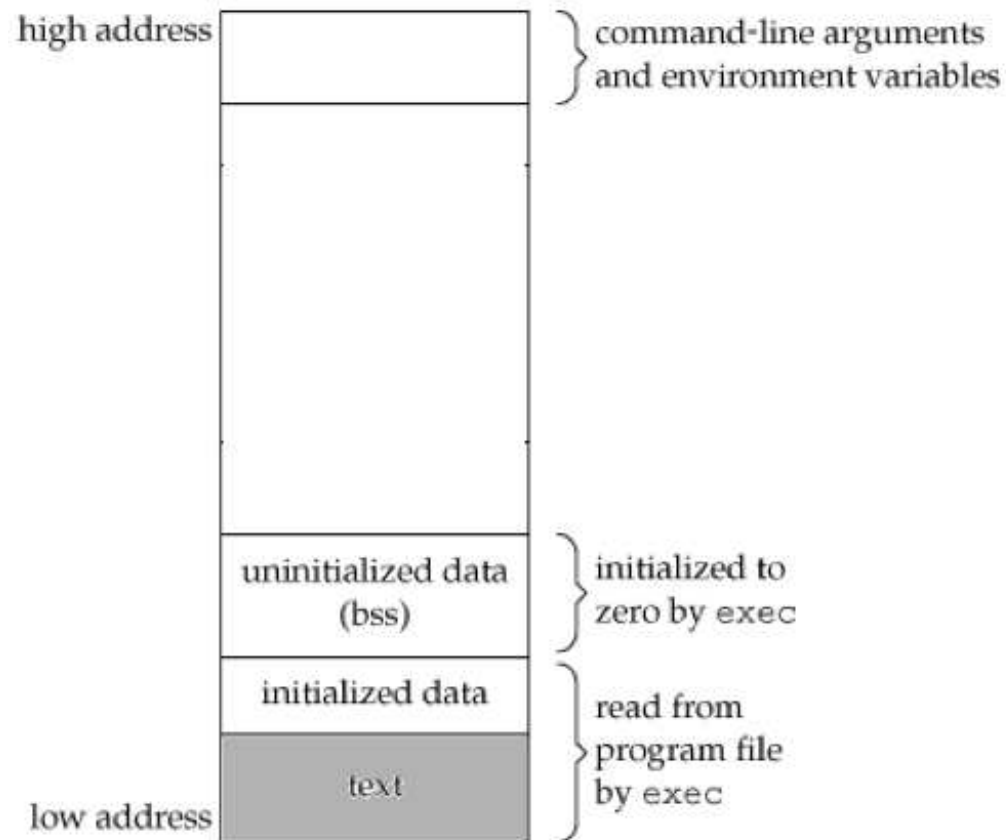
Memory Layout of a C Program



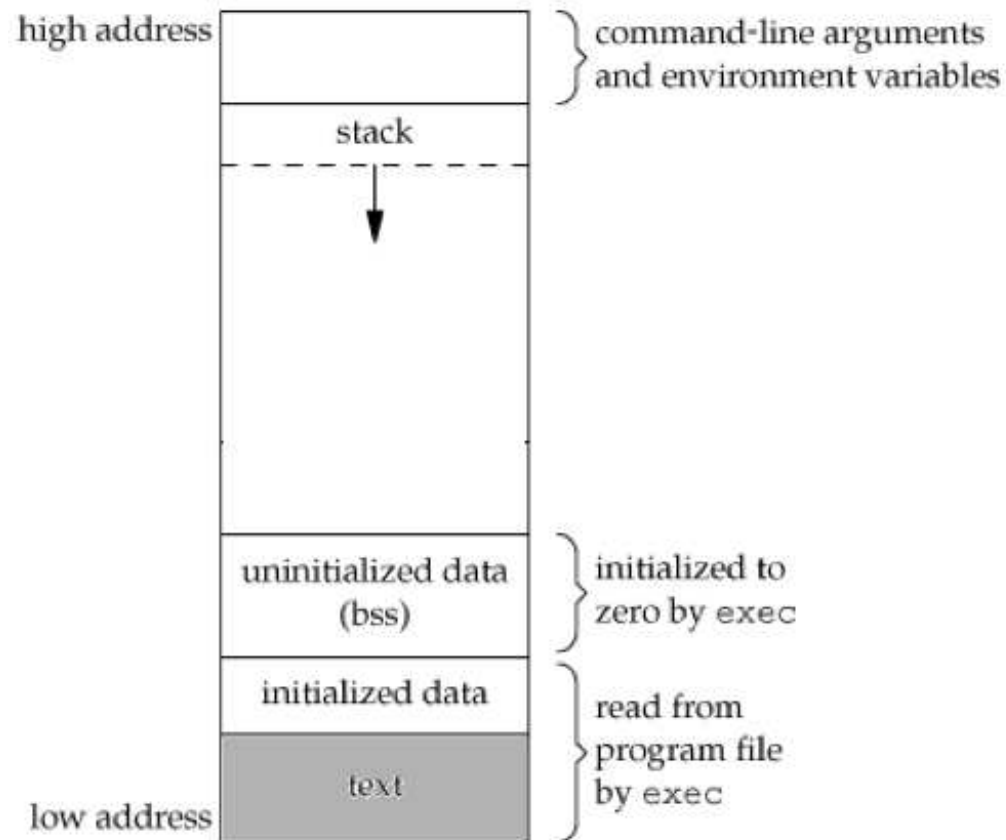
Memory Layout of a C Program



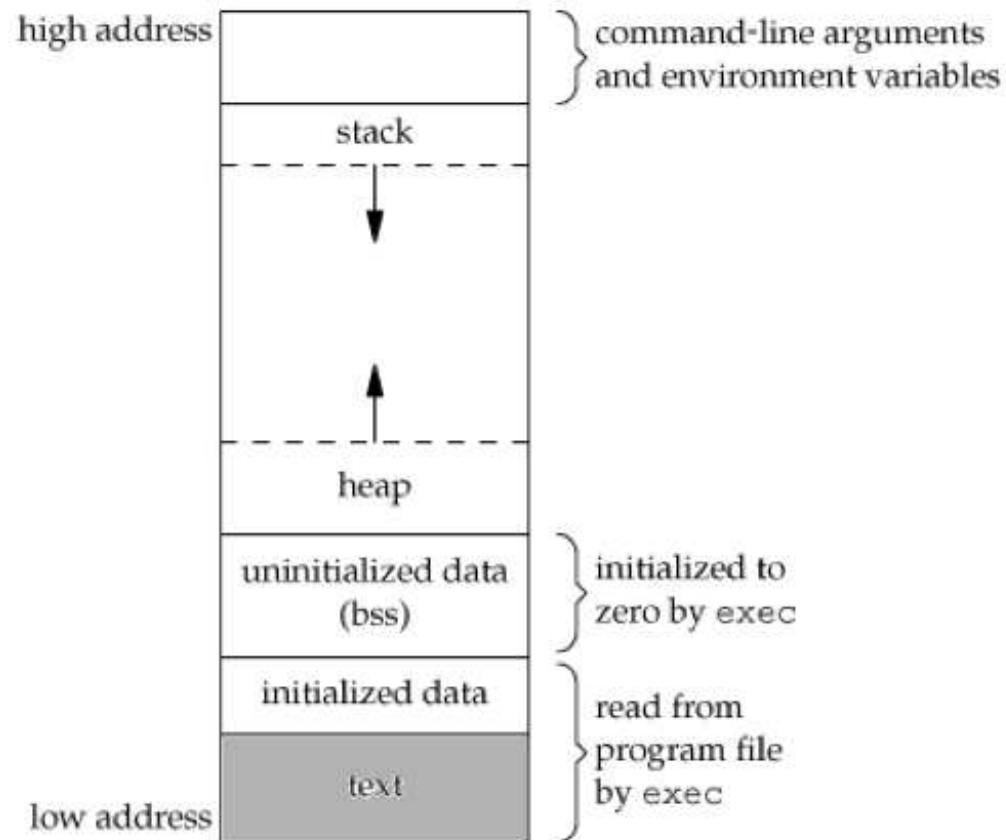
Memory Layout of a C Program



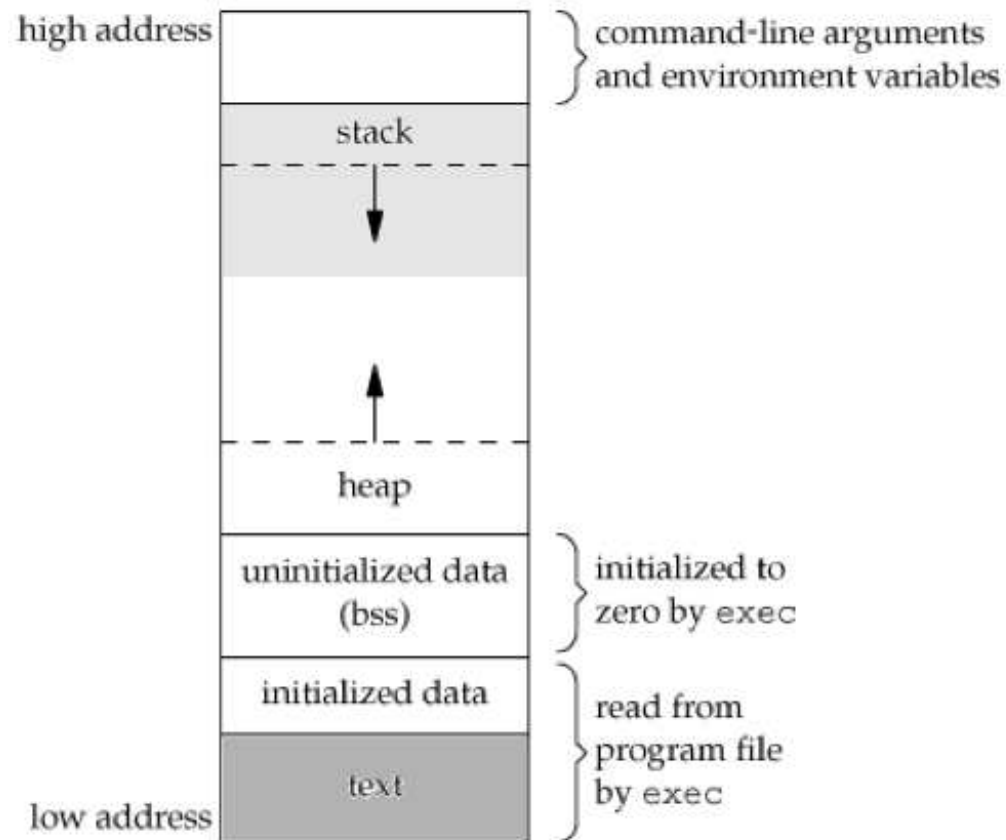
Memory Layout of a C Program



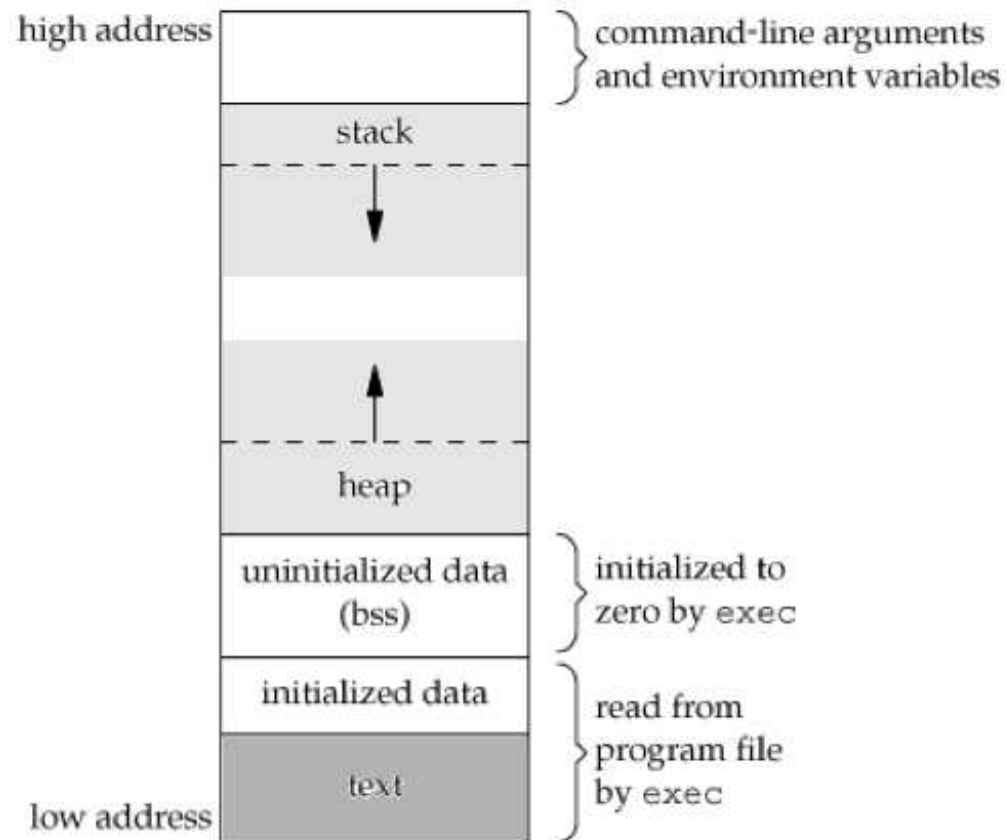
Memory Layout of a C Program



Memory Layout of a C Program



Memory Layout of a C Program



Memory Layout of a C Program

On NetBSD:

```
$ cc hw.c
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked (uses shared libs), for NetBSD 5.0, not stripped
$ ldd a.out
a.out:
        -lc.12 => /usr/lib/libc.so.12
$ size a.out
   text    data     bss      dec     hex filename
   2301     552     120     2973     b9d a.out
$ objdump -d a.out > obj
$ wc -l obj
    271 obj
$
```


Memory Layout of a C Program

On Mac OS X:

```
$ cc hw.c
$ file a.out
a.out: Mach-O 64-bit executable x86_64
$ otool -L a.out
a.out:
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0,
current version 125.2.11)
$ size a.out
__TEXT __DATA __OBJC others dec hex
4096 4096 0 4294971392 4294979584 100003000
$ otool -t -v a.out > obj
$ wc -l obj
    32 obj
$
```

Memory Layout of a C Program

On Linux:

```
$ cc hw.c
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.15, not stripped
$ ldd a.out
linux-gate.so.1 => (0x00c66000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x006b4000)
/lib/ld-linux.so.2 (0x005fe000)
$ size a.out
   text    data     bss     dec     hex filename
   918     264        8    1190    4a6 a.out
$ objdump -d a.out >obj
$ wc -l obj
225 obj
$
```

Memory Layout of a C Program

On NetBSD:

```
$ cc -static hw.c
```

```
$ file a.out
```

```
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically  
linked, for NetBSD 5.0, not stripped
```

```
$ ldd a.out
```

```
ldd: a.out: unrecognized file format2 [2 != 1]
```

```
$ size a.out
```

text	data	bss	dec	hex filename
151877	4416	16384	172677	2a285 a.out

```
$ size a.out.dyn
```

text	data	bss	dec	hex filename
2301	552	120	2973	b9d a.out

```
$ objdump -d a.out > obj
```

```
$ wc -l obj
```

```
35029 obj
```

```
$
```

Memory Layout of a C Program

On Mac OS X:

```
$ cc -static hw.c
ld: library not found for -lcrt0.o
collect2: ld returned 1 exit status
$
```

Memory Layout of a C Program

On Linux:

```
$ cc -static hw.c
```

```
$ file a.out
```

```
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),  
statically linked, for GNU/Linux 2.6.15, not stripped
```

```
$ ldd a.out
```

```
/usr/bin/ldd: line 161: /lib64/ld-linux-x86-64.so.2: cannot execute binary file  
not a dynamic executable
```

```
$ size a.out
```

text	data	bss	dec	hex	filename
510786	1928	7052	519766	7ee56	a.out

```
$ objdump -d a.out >obj
```

```
$ wc -l obj
```

```
114420 obj
```

```
$
```

Memory Allocation

```
#include <stdlib.h>

void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsiz);
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

Memory Allocation

```
#include <stdlib.h>

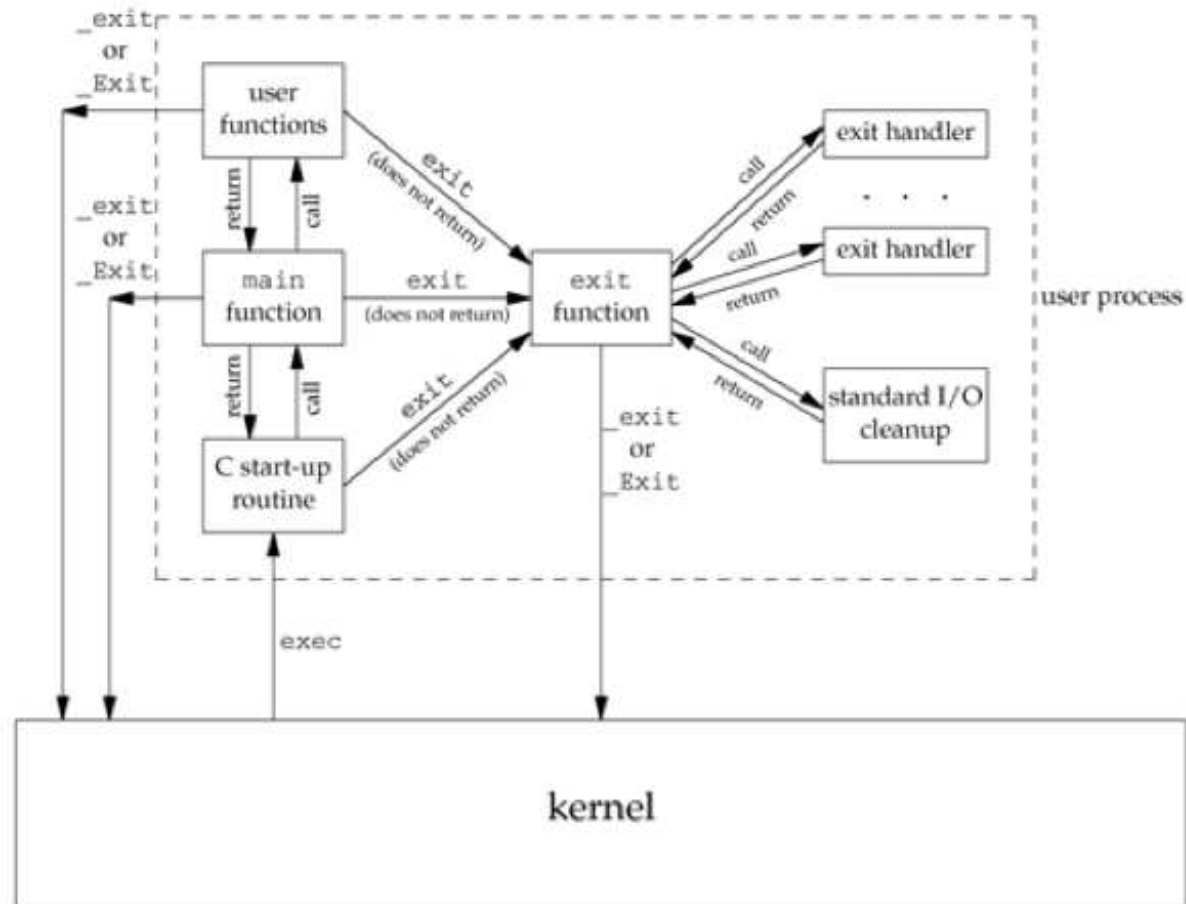
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsz);
void *alloca(size_t size);

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

Knowing this, how does manipulation of the environment work?

Lifetime of a UNIX Process



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 resource, struct rlimit *rlp);
int setrlimit(int resource, 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

getrlimit(2) and setrlimit(2)

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

int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, 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 Control

Review from our first class, the world's simplest shell:

```
int
main(int argc, char **argv)
{
    char buf[1024];
    pid_t pid;
    int status;

    while (getinput(buf, sizeof(buf))) {
        buf[strlen(buf) - 1] = '\0';

        if((pid=fork()) == -1) {
            fprintf(stderr, "shell: can't fork: %s\n",
                    strerror(errno));
            continue;
        } else if (pid == 0) {
            /* child */
            execlp(buf, buf, (char *)0);
            fprintf(stderr, "shell: couldn't exec %s: %s\n", buf,
                    strerror(errno));
            exit(EX_DATAERR);
        }

        if ((pid=waitpid(pid, &status, 0)) < 0)
            fprintf(stderr, "shell: waitpid error: %s\n",
                    strerror(errno));
    }

    exit(EX_OK);
}
```

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>

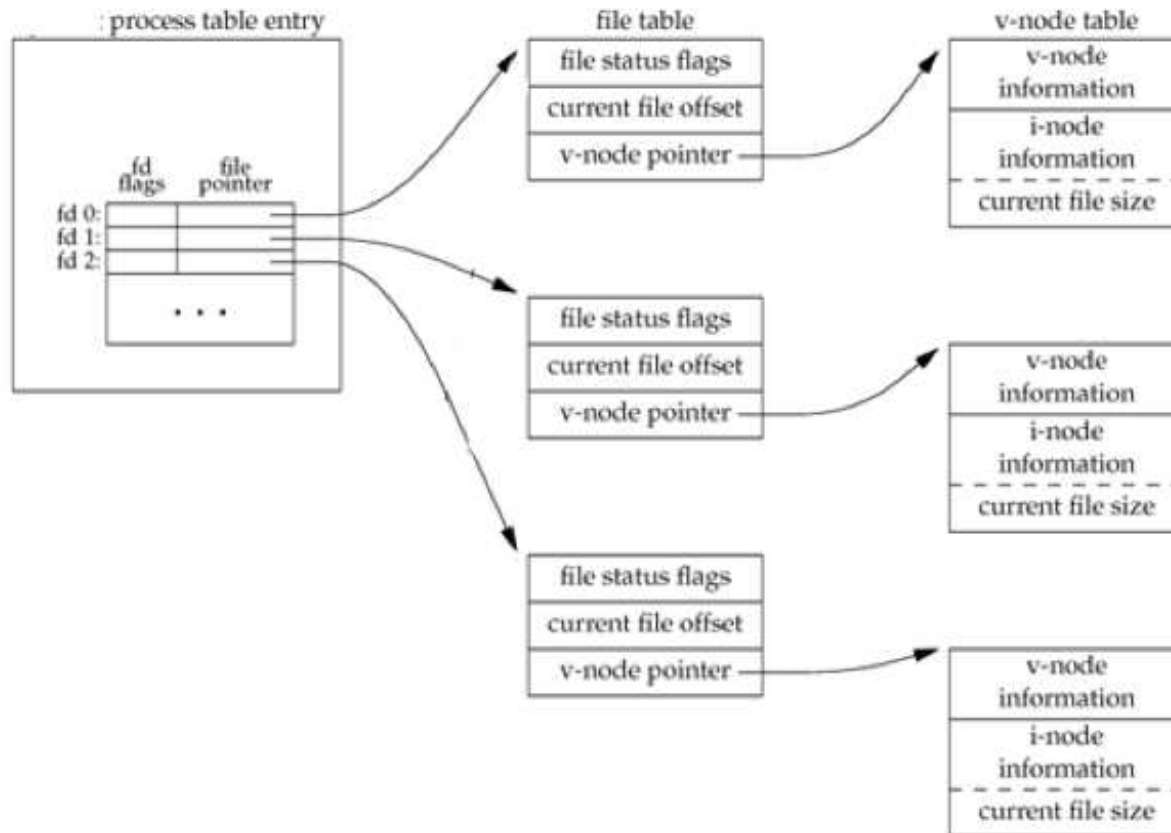
pid_t fork(void);
```

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:

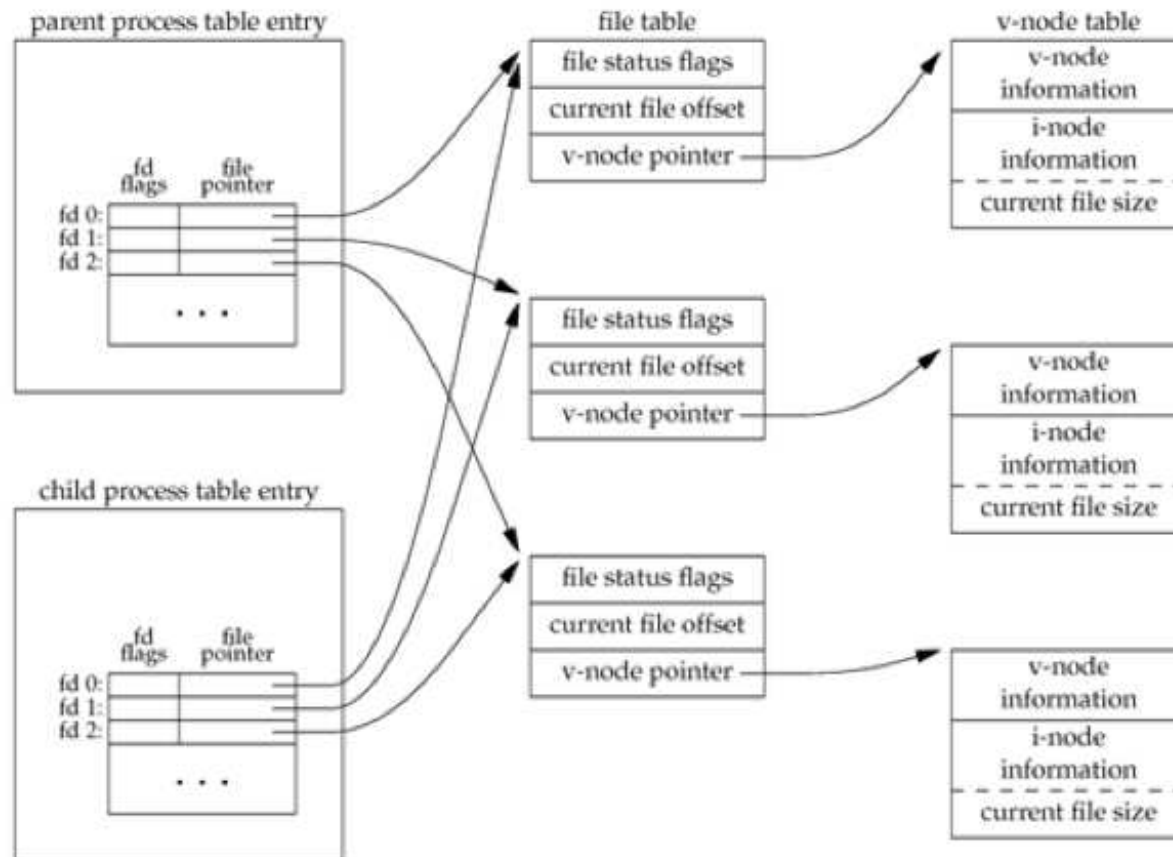
- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors.
- The child process' resource utilizations are set to 0.

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

fork(2)



fork(2)



fork(2)

```
$ cc -Wall forkflush.c
$ ./a.out
a write to stdout
before fork
pid = 12149, glob = 7, var = 89
pid = 12148, glob = 6, var = 88
$ ./a.out | cat
a write to stdout
before fork
pid = 12153, glob = 7, var = 89
before fork
pid = 12151, glob = 6, var = 88
$
```

The `exec(3)` functions

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);
int execv(const char *pathname, char * const argv[]);
int execlp(const char *pathname, const char *arg0, ... /* (char *) 0, char *const envp[] */ );
int execve(const char *pathname, char * const argv[], 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 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 determine 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). Then 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.

What if we don't wait(2)?

What if we don't wait (2)?



What if we don't wait(2)?

```
$ cc -Wall zombies.c
$ ./a.out
Let's create some zombies!
====
15603 s003 S+      0:00.00 ./a.out
15604 s003 Z+      0:00.00 (a.out)
====
15603 s003 S+      0:00.00 ./a.out
15604 s003 Z+      0:00.00 (a.out)
15608 s003 Z+      0:00.00 (a.out)
====
15603 s003 S+      0:00.00 ./a.out
15604 s003 Z+      0:00.00 (a.out)
15612 s003 Z+      0:00.00 (a.out)
====
15603 s003 S+      0:00.00 ./a.out
15604 s003 Z+      0:00.00 (a.out)
15612 s003 Z+      0:00.00 (a.out)
15616 s003 Z+      0:00.00 (a.out)
```


Notes and Homework

Reading:

- Stevens, Chapter 7 and 8

Other:

- work on your midterm project!