

CS810D - Advanced Programming in the UNIX Environment

—

Process Environment, Process Control

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann

`jschauma@cs.stevens.edu`

`http://www.cs.stevens.edu/~jschauma/810D-APUE/`

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
- `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`
- 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`
- 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);

#include <unistd.h>

void _exit(int status);
```

- `_exit` returns to the kernel immediately.
- `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.

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 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 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.

All three return pointer suitably aligned for any kind of data.

setjmp(3) and longjmp(3)

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int longjmp(jmp_buf env, int val);
```

- “non-local goto”
- handle error conditions in deeply nested functions
- longjmp may not be called after the routine which called setjmp

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*

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.

The `exec(3)` functions

```
#include <unistd.h>

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

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

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

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

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, it takes the `argv`'s as a vector: `const * char argv[]`
- if it has an `l` in its name, it takes the `argv`'s as 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

vfork(2) and COW

While Unix has no notion of an atomic `spawn` operation, the sequence: `fork()`, `exec()` is still extremely common. A special purpose `fork()`, called `vfork()` was added in early versions of Unix. `vfork()`'s purpose was to create an additional process which promised to immediately call `exec()`. Because of this promise, `vfork()` was able to get away with not copying the heap, data, stack, bss, etc. of the parent.

However, in the modern, more mature VM model, this “copying” is never done until an explicit write to a segment is made. Even then, only the page (usually around 4096 or 8192 bytes) which contains the data is copied. This strategy is known as “Copy On Write” or “COW”. This model essentially obsoletes `vfork()`.

wait(2) and waitpid(2)

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

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

Difference between these two functions:

- `wait(2)` can 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

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.

Notes and Homework

Section `setjmp/longjmp` taken from

<http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/Setjmp/lecture.html>

Reading:

- Stevens, Chapter 7 and 8

Other:

- work on your midterm project!