Process Creation and Termination

fork()

#include <unistd.h>

pid_t fork(void);

In parent: returns process ID of child on success, or –1 on error;
in successfully created child: always returns 0

The *fork()* system call creates a new process, the *child*, which is an almost exact duplicate of the calling process, the *parent*.

vfork()

#include <unistd.h>

pid_t vfork(void);

In parent: returns process ID of child on success, or –1 on error;
in successfully created child: always returns

Like *fork()*, *vfork()* is used by the calling process to create a new child process. However, *vfork()* is expressly designed to be used in programs where the child per-forms an immediate *exec()* call.

Two features distinguish the *vfork()* system call from fork() and make it more efficient:

- No duplication of virtual memory pages or page tables is done for the child process. Instead, the child shares the parent's memory until it either performs a successful *exec()* or calls *_exit()* to terminate.
- Execution of the parent process is suspended until the child has performed an *exec()* or *_exit()*.

exit()

#include <stdlib.h>

void exit(int status)

The *status* argument given to *exit()* defines the termination status of the process, which is available to the parent of this process when it calls *wait()*. Although defined as an int, only the bottom 8 bits of *status* are actually made available to the parent. By convention, a termination status of 0 indicates that a process completed successfully, and a nonzero status value indicates that the process terminated unsuccessfully. There are no fixed rules about how nonzero status values are to be interpreted; different applications follow their own conventions, which should be described in their documentation.

The following actions are performed by *exit()*:

- Exit handlers (functions registered with *atexit()* and *on_exit()*) are called, in reverse order of their registration.
- The *stdio* stream buffers are flushed.
- The *_exit()* system call is invoked, using the value supplied in status.

  Unlike *_exit()*, which is UNIX-specific, *exit()* is defined as part of the standard C library; that is, it is available with every C implementation. One other way in which a process may terminate is to return from main(), either explicitly, or implicitly, by falling off the end of the main() function. Performing an explicit *return n* is generally equivalent to calling *exit(n)*, since the run-time function that invokes *main()* uses the return value from *main()* in a call to *exit()*.

atexit()

```
#include <stdlib.h>

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

Returns 0 on success, or nonzero on error

The *atexit()* function adds *func* to a list of functions that are called when the process terminates. The function *func* should be defined to take no arguments and return no value, thus having the following general form:

```
void func(void)
{
   /* Perform some actions */
}
```

Note that *atexit()* returns a nonzero value (not necessarily –1) on error. It is possible to register multiple exit handlers (and even the same exit handler multiple times). When the program invokes *exit()*, these functions are called in reverse order of registration.

on_exit()

```
#define _BSD_SOURCE         /* Or: #define _SVID_SOURCE */
#include <stdlib.h>

int on_exit(void (*func)(int, void *), void *arg);
```

Returns 0 on success, or nonzero on error

Exit handlers registered with *atexit()* suffer a couple of limitations. The first is that when called, an exit handler doesn't know what status was passed to *exit()*. Occasionally, knowing the status could be useful; for example, we may like to perform different actions depending on whether the process is exiting successfully or unsuccessfully. The second limitation is that we can't specify an argument to the exithandler when it is called. Such a facility could be useful to define an exit handler that performs different actions depending on its argument, or to register a function multiple times, each time with a different argument. To address these limitations, glibc provides a (nonstandard) alternative method of registering exit handlers: *on_exit()*.

wait()

#include <sys/wait.h>

*pid_t wait(int *status);*

Returns process ID of terminated child, or –1 on error

The *wait()* system call does the following:

1. If no (previously unwaited-for) child of the calling process has yet terminated, the call blocks until one of the children terminates. If a child has already terminated by the time of the call, *wait()* returns immediately.
2. If status is not NULL, information about how the child terminated is returned in the integer to which status points.
3. The kernel adds the process CPU times and resource usage statistics to running totals for all children of this parent process.
4. As its function result, *wait()* returns the process ID of the child that has terminated.

On error, *wait()* returns –1. One possible error is that the calling process has no (previously unwaited-for) children, which is indicated by the *errno* value *ECHILD*.

waitpid()

#include <sys/wait.h>

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

Returns process ID of child, 0 (see text), or –1 on error

The *wait()* system call has a number of limitations, which *waitpid()* was designed to address:

- If a parent process has created multiple children, it is not possible to *wait()* for the completion of a specific child; we can only wait for the next child that terminates.
- If no child has yet terminated, *wait()* always blocks. Sometimes, it would be preferable to perform a nonblocking wait so that if no child has yet terminated, we obtain an immediate indication of this fact.
- Using *wait()*, we can find out only about children that have terminated. It is not possible to be notified when a child is stopped by a signal (such as SIGSTOP or SIGTTIN) or when a stopped child is resumed by delivery of a SIGCONT signal.

The return value and status arguments of *waitpid()* are the same as for *wait()*. The pid argument enables the selection of the child to be waited for, as follows:

- If pid > 0, wait for the child whose process ID equals pid.
- If pid == 0, wait for any child in the same process group as the caller (parent).
- If pid < –1, wait for any child whose process group identifier equals the absolute value of pid.
- If pid == –1, wait for any child. The call *wait(&status)* is equivalent to the call *waitpid(–1, &status, 0)*.

execl, execlp, execle, execv, execvp, execvpe - execute a file

#include <unistd.h>

*extern char **environ;*

*int execl(const char *path, const char *arg, ...);*
*int execlp(const char *file, const char *arg, ...);*
*int execle(const char *path, const char *arg,*
*..., char * const envp[]);*
*int execv(const char *path, char *const argv[]);*
*int execvp(const char *file, char *const argv[]);*
*int execvpe(const char *file, char *const argv[],*
*char *const envp[]);*

The **exec**() family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for **execve**(2). (See the manual page for **execve**(2) for further details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer, and, since these are variadic functions, this pointer must be cast *(char *) NULL*.

The **execv**(), **execvp**(), and **execvpe**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a NULL pointer.

The **execle**() and **execvpe**() functions allow the caller to specify the environment of the executed program via the argument *envp*. The *envp* argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the calling process.

Special semantics for execlp() and execvp()

The **execlp**(), **execvp**(), and **execvpe**() functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character. The file is sought in the colon-separated list of directory pathnames specified in the **PATH** environment variable. If this variable isn't defined, the path list defaults to the current directory followed by the list of directories returned by *confstr(_CS_PATH)*. (This **confstr**(3) call typically returns the value "/bin:/usr/bin".)

If the specified filename includes a slash character, then **PATH** is ignored, and the file at the specified pathname is executed.

In addition, certain errors are treated specially.

If permission is denied for a file (the attempted **execve**(2) failed with the error **EACCES**), these functions will continue searching the rest of the search path. If no other file is found, however, they will return with *errno* set to **EACCES**.

If the header of a file isn't recognized (the attempted **execve**(2) failed with the error **ENOEXEC**), these functions will execute the shell (*/bin/sh*) with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

Return Value

The **exec**() functions only return if an error has occurred. The return value is -1, and *errno* is set to indicate the error.

Errors

All of these functions may fail and set *errno* for any of the errors specified for **execve**(2).