# 26 Processes

*Processes* are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.

Processes are organized hierarchically. Each process has a *parent process* which explicitly arranged to create it. The processes created by a given parent are called its *child processes*. A child inherits many of its attributes from the parent process.

This chapter describes how a program can create, terminate, and control child processes. Actually, there are three distinct operations involved: creating a new child process, causing the new process to execute a program, and coordinating the completion of the child process with the original program.

The `system` function provides a simple, portable mechanism for running another program; it does all three steps automatically. If you need more control over the details of how this is done, you can use the primitive functions to do each step individually instead.

## 26.1 Running a Command

The easy way to run another program is to use the `system` function. This function does all the work of running a subprogram, but it doesn't give you much control over the details: you have to wait until the subprogram terminates before you can do anything else.

`int system` (*const char *`command`*) [Function]

Preliminary: | MT-Safe | AS-Unsafe plugin heap lock | AC-Unsafe lock mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function executes *command* as a shell command. In the GNU C Library, it always uses the default shell `sh` to run the command. In particular, it searches the directories in `PATH` to find programs to execute. The return value is `-1` if it wasn't possible to create the shell process, and otherwise is the status of the shell process. See Section 26.6 [Process Completion], page 753, for details on how this status code can be interpreted.

If the *command* argument is a null pointer, a return value of zero indicates that no command processor is available.

This function is a cancellation point in multi-threaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `system` is called. If the thread gets canceled these resources stay allocated until the program ends. To avoid this calls to `system` should be protected using cancellation handlers.

The `system` function is declared in the header file `stdlib.h`.

**Portability Note:** Some C implementations may not have any notion of a command processor that can execute other programs. You can determine whether a command processor exists by executing `system (NULL)`; if the return value is nonzero, a command processor is available.

The `popen` and `pclose` functions (see Section 15.2 [Pipe to a Subprocess], page 424) are closely related to the `system` function. They allow the parent process to communicate with the standard input and output channels of the command being executed.

## 26.2 Process Creation Concepts

This section gives an overview of processes and of the steps involved in creating a process and making it run another program.

Each process is named by a *process ID* number. A unique process ID is allocated to each process when it is created. The *lifetime* of a process ends when its termination is reported to its parent process; at that time, all of the process resources, including its process ID, are freed.

Processes are created with the `fork` system call (so the operation of creating a new process is sometimes called *forking* a process). The *child process* created by `fork` is a copy of the original *parent process*, except that it has its own process ID.

After forking a child process, both the parent and child processes continue to execute normally. If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling `wait` or `waitpid` (see Section 26.6 [Process Completion], page 753). These functions give you limited information about why the child terminated—for example, its exit status code.

A newly forked child process continues to execute the same program as its parent process, at the point where the `fork` call returns. You can use the return value from `fork` to tell whether the program is running in the parent process or the child.

Having several processes run the same program is only occasionally useful. But the child can execute another program using one of the `exec` functions; see Section 26.5 [Executing a File], page 750. The program that the process is executing is called its *process image*. Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image.

## 26.3 Process Identification

The `pid_t` data type represents process IDs. You can get the process ID of a process by calling `getpid`. The function `getppid` returns the process ID of the parent of the current process (this is also known as the *parent process ID*). Your program should include the header files `unistd.h` and `sys/types.h` to use these functions.

`pid_t` [Data Type]

The `pid_t` data type is a signed integer type which is capable of representing a process ID. In the GNU C Library, this is an `int`.

`pid_t getpid` (*void*) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getpid` function returns the process ID of the current process.

`pid_t getppid (void)` [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getppid` function returns the process ID of the parent of the current process.

## 26.4 Creating a Process

The `fork` function is the primitive for creating a process. It is declared in the header file `unistd.h`.

`pid_t fork (void)` [Function]

Preliminary: | MT-Safe | AS-Unsafe plugin | AC-Unsafe lock | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `fork` function creates a new process.

If the operation is successful, there are then both parent and child processes and both see `fork` return, but with different values: it returns a value of `0` in the child process and returns the child's process ID in the parent process.

If process creation failed, `fork` returns a value of `-1` in the parent process. The following `errno` error conditions are defined for `fork`:

| | |
|---|---|
| `EAGAIN` | There aren't enough system resources to create another process, or the user already has too many processes running. This means exceeding the `RLIMIT_NPROC` resource limit, which can usually be increased; see Section 22.2 [Limiting Resource Usage], page 630. |
| `ENOMEM` | The process requires more space than the system can supply. |

The specific attributes of the child process that differ from the parent process are:

- The child process has its own unique process ID.
- The parent process ID of the child process is the process ID of its parent process.
- The child process gets its own copies of the parent process's open file descriptors. Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa. See Section 13.11 [Control Operations on Files], page 358. However, the file position associated with each descriptor is shared by both processes; see Section 11.1.2 [File Position], page 243.
- The elapsed processor times for the child process are set to zero; see Section 21.3.2 [Processor Time Inquiry], page 596.
- The child doesn't inherit file locks set by the parent process. See Section 13.11 [Control Operations on Files], page 358.
- The child doesn't inherit alarms set by the parent process. See Section 21.5 [Setting an Alarm], page 623.
- The set of pending signals (see Section 24.1.3 [How Signals Are Delivered], page 660) for the child process is cleared. (The child process inherits its mask of blocked signals and signal actions from the parent process.)

`pid_t vfork (void)` [Function]

Preliminary: | MT-Safe | AS-Unsafe plugin | AC-Unsafe lock | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `vfork` function is similar to `fork` but on some systems it is more efficient; however, there are restrictions you must follow to use it safely.

While `fork` makes a complete copy of the calling process's address space and allows both the parent and child to execute independently, `vfork` does not make this copy. Instead, the child process created with `vfork` shares its parent's address space until it calls `_exit` or one of the `exec` functions. In the meantime, the parent process suspends execution.

You must be very careful not to allow the child process created with `vfork` to modify any global data or even local variables shared with the parent. Furthermore, the child process cannot return from (or do a long jump out of) the function that called `vfork`! This would leave the parent process's control information very confused. If in doubt, use `fork` instead.

Some operating systems don't really implement `vfork`. The GNU C Library permits you to use `vfork` on all systems, but actually executes `fork` if `vfork` isn't available. If you follow the proper precautions for using `vfork`, your program will still work even if the system uses `fork` instead.

## 26.5 Executing a File

This section describes the `exec` family of functions, for executing a file as a process image. You can use these functions to make a child process execute a new program after it has been forked.

To see the effects of `exec` from the point of view of the called program, see Chapter 25 [The Basic Program/System Interface], page 703.

The functions in this family differ in how you specify the arguments, but otherwise they all do the same thing. They are declared in the header file `unistd.h`.

`int execv (const char *filename, char *const argv[])` [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `execv` function executes the file named by *filename* as a new process image.

The *argv* argument is an array of null-terminated strings that is used to provide a value for the `argv` argument to the `main` function of the program to be executed. The last element of this array must be a null pointer. By convention, the first element of this array is the file name of the program sans directory names. See Section 25.1 [Program Arguments], page 703, for full details on how programs can access these arguments.

The environment for the new process image is taken from the `environ` variable of the current process image; see Section 25.4 [Environment Variables], page 736, for information about environments.

`int execl (const char *filename, const char *arg0, . . .)` [Function]

Preliminary: | MT-Safe | AS-Unsafe heap | AC-Unsafe mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This is similar to `execv`, but the *argv* strings are specified individually instead of as an array. A null pointer must be passed as the last such argument.

`int` **`execve`** (*const char *`filename`, char *const `argv`[], char *const `env`[]*) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This is similar to `execv`, but permits you to specify the environment for the new program explicitly as the *env* argument. This should be an array of strings in the same format as for the `environ` variable; see Section 25.4.1 [Environment Access], page 736.

`int` **`execle`** (*const char *`filename`, const char *`arg0`, ..., char *const `env`[]*) [Function]

Preliminary: | MT-Safe | AS-Unsafe heap | AC-Unsafe mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This is similar to `execl`, but permits you to specify the environment for the new program explicitly. The environment argument is passed following the null pointer that marks the last *argv* argument, and should be an array of strings in the same format as for the `environ` variable.

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

Preliminary: | MT-Safe env | AS-Unsafe heap | AC-Unsafe mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `execvp` function is similar to `execv`, except that it searches the directories listed in the `PATH` environment variable (see Section 25.4.2 [Standard Environment Variables], page 739) to find the full file name of a file from *filename* if *filename* does not contain a slash.

This function is useful for executing system utility programs, because it looks for them in the places that the user has chosen. Shells use it to run the commands that users type.

`int` **`execlp`** (*const char *`filename`, const char *`arg0`, ...*) [Function]

Preliminary: | MT-Safe env | AS-Unsafe heap | AC-Unsafe mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is like `execl`, except that it performs the same file name searching as the `execvp` function.

The size of the argument list and environment list taken together must not be greater than `ARG_MAX` bytes. See Section 32.1 [General Capacity Limits], page 836. On GNU/Hurd systems, the size (which compares against `ARG_MAX`) includes, for each string, the number of characters in the string, plus the size of a `char *`, plus one, rounded up to a multiple of the size of a `char *`. Other systems may have somewhat different rules for counting.

These functions normally don't return, since execution of a new program causes the currently executing program to go away completely. A value of `-1` is returned in the event of a failure. In addition to the usual file name errors (see Section 11.2.3 [File Name Errors], page 245), the following `errno` error conditions are defined for these functions:

| | |
|---|---|
| `E2BIG` | The combined size of the new program's argument list and environment list is larger than `ARG_MAX` bytes. GNU/Hurd systems have no specific limit on the argument list size, so this error code cannot result, but you may get `ENOMEM` instead if the arguments are too big for available memory. |
| `ENOEXEC` | The specified file can't be executed because it isn't in the right format. |
| `ENOMEM` | Executing the specified file requires more storage than is available. |

If execution of the new file succeeds, it updates the access time field of the file as if the file had been read. See Section 14.9.9 [File Times], page 412, for more details about access times of files.

The point at which the file is closed again is not specified, but is at some point before the process exits or before another process image is executed.

Executing a new process image completely changes the contents of memory, copying only the argument and environment strings to new locations. But many other attributes of the process are unchanged:

- The process ID and the parent process ID. See Section 26.2 [Process Creation Concepts], page 748.
- Session and process group membership. See Section 28.1 [Concepts of Job Control], page 761.
- Real user ID and group ID, and supplementary group IDs. See Section 30.2 [The Persona of a Process], page 789.
- Pending alarms. See Section 21.5 [Setting an Alarm], page 623.
- Current working directory and root directory. See Section 14.1 [Working Directory], page 376. On GNU/Hurd systems, the root directory is not copied when executing a setuid program; instead the system default root directory is used for the new program.
- File mode creation mask. See Section 14.9.7 [Assigning File Permissions], page 409.
- Process signal mask; see Section 24.7.3 [Process Signal Mask], page 692.
- Pending signals; see Section 24.7 [Blocking Signals], page 690.
- Elapsed processor time associated with the process; see Section 21.3.2 [Processor Time Inquiry], page 596.

If the set-user-ID and set-group-ID mode bits of the process image file are set, this affects the effective user ID and effective group ID (respectively) of the process. These concepts are discussed in detail in Section 30.2 [The Persona of a Process], page 789.

Signals that are set to be ignored in the existing process image are also set to be ignored in the new process image. All other signals are set to the default action in the new process image. For more information about signals, see Chapter 24 [Signal Handling], page 659.

File descriptors open in the existing process image remain open in the new process image, unless they have the `FD_CLOEXEC` (close-on-exec) flag set. The files that remain open inherit all attributes of the open file description from the existing process image, including file locks. File descriptors are discussed in Chapter 13 [Low-Level Input/Output], page 322.

Streams, by contrast, cannot survive through `exec` functions, because they are located in the memory of the process itself. The new process image has no streams except those it creates afresh. Each of the streams in the pre-`exec` process image has a descriptor inside it, and these descriptors do survive through `exec` (provided that they do not have `FD_CLOEXEC` set). The new process image can reconnect these to new streams using `fdopen` (see Section 13.4 [Descriptors and Streams], page 332).

## 26.6 Process Completion

The functions described in this section are used to wait for a child process to terminate or stop, and determine its status. These functions are declared in the header file `sys/wait.h`.

`pid_t waitpid` (*pid_t* `pid`, *int* `*status-ptr`, *int* `options`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `waitpid` function is used to request status information from a child process whose process ID is *pid*. Normally, the calling process is suspended until the child process makes status information available by terminating.

Other values for the *pid* argument have special interpretations. A value of `-1` or `WAIT_ANY` requests status information for any child process; a value of `0` or `WAIT_MYPGRP` requests information for any child process in the same process group as the calling process; and any other negative value − *pgid* requests information for any child process whose process group ID is *pgid*.

If status information for a child process is available immediately, this function returns immediately without waiting. If more than one eligible child process has status information available, one of them is chosen randomly, and its status is returned immediately. To get the status from the other eligible child processes, you need to call `waitpid` again.

The *options* argument is a bit mask. Its value should be the bitwise OR (that is, the '`|`' operator) of zero or more of the `WNOHANG` and `WUNTRACED` flags. You can use the `WNOHANG` flag to indicate that the parent process shouldn't wait; and the `WUNTRACED` flag to request status information from stopped processes as well as processes that have terminated.

The status information from the child process is stored in the object that *status-ptr* points to, unless *status-ptr* is a null pointer.

This function is a cancellation point in multi-threaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `waitpid` is called. If the thread gets canceled these resources stay allocated until the program ends. To avoid this calls to `waitpid` should be protected using cancellation handlers.

The return value is normally the process ID of the child process whose status is reported. If there are child processes but none of them is waiting to be noticed, `waitpid` will block until one is. However, if the `WNOHANG` option was specified, `waitpid` will return zero instead of blocking.

If a specific PID to wait for was given to `waitpid`, it will ignore all other children (if any). Therefore if there are children waiting to be noticed but the child whose PID was specified is not one of them, `waitpid` will block or return zero as described above.

A value of `-1` is returned in case of error. The following `errno` error conditions are defined for this function:

`EINTR` The function was interrupted by delivery of a signal to the calling process. See Section 24.5 [Primitives Interrupted by Signals], page 685.

`ECHILD` There are no child processes to wait for, or the specified *pid* is not a child of the calling process.

`EINVAL` An invalid value was provided for the *options* argument.

These symbolic constants are defined as values for the *pid* argument to the `waitpid` function.

`WAIT_ANY`

This constant macro (whose value is `-1`) specifies that `waitpid` should return status information about any child process.

`WAIT_MYPGRP`

This constant (with value `0`) specifies that `waitpid` should return status information about any child process in the same process group as the calling process.

These symbolic constants are defined as flags for the *options* argument to the `waitpid` function. You can bitwise-OR the flags together to obtain a value to use as the argument.

`WNOHANG`

This flag specifies that `waitpid` should return immediately instead of waiting, if there is no child process ready to be noticed.

`WUNTRACED`

This flag specifies that `waitpid` should report the status of any child processes that have been stopped as well as those that have terminated.

`pid_t wait` (*int* ***status-ptr***) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This is a simplified version of `waitpid`, and is used to wait until any one child process terminates. The call:

```
wait (&status)
```

is exactly equivalent to:

```
waitpid (-1, &status, 0)
```

This function is a cancellation point in multi-threaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `wait` is called. If the thread gets canceled these resources stay allocated until the program ends. To avoid this calls to `wait` should be protected using cancellation handlers.

`pid_t wait4` (*pid_t* ***pid***, *int* ***status-ptr***, *int* ***options***, *struct rusage* ***usage***) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

If *usage* is a null pointer, `wait4` is equivalent to `waitpid (pid, status-ptr, options)`.

If *usage* is not null, `wait4` stores usage figures for the child process in `*rusage` (but only if the child has terminated, not if it has stopped). See Section 22.1 [Resource Usage], page 628.

This function is a BSD extension.

Here's an example of how to use `waitpid` to get the status from all child processes that have terminated, without ever waiting. This function is designed to be a handler for `SIGCHLD`, the signal that indicates that at least one child process has terminated.

```
void
sigchld_handler (int signum)
{
  int pid, status, serrno;
  serrno = errno;
  while (1)
    {
      pid = waitpid (WAIT_ANY, &status, WNOHANG);
      if (pid < 0)
        {
          perror ("waitpid");
          break;
        }
      if (pid == 0)
        break;
      notice_termination (pid, status);
    }
  errno = serrno;
}
```

## 26.7 Process Completion Status

If the exit status value (see Section 25.7 [Program Termination], page 742) of the child process is zero, then the status value reported by `waitpid` or `wait` is also zero. You can test for other kinds of information encoded in the returned status value using the following macros. These macros are defined in the header file `sys/wait.h`.

`int` **`WIFEXITED`** (*int* `status`) [Macro]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This macro returns a nonzero value if the child process terminated normally with `exit` or `_exit`.

`int` **`WEXITSTATUS`** (*int* `status`) [Macro]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

If `WIFEXITED` is true of *status*, this macro returns the low-order 8 bits of the exit status value from the child process. See Section 25.7.2 [Exit Status], page 743.

`int` **`WIFSIGNALED`** (*int* `status`) [Macro]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This macro returns a nonzero value if the child process terminated because it received a signal that was not handled. See Chapter 24 [Signal Handling], page 659.

`int WTERMSIG (`*int* `status)` [Macro]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

If `WIFSIGNALED` is true of *status*, this macro returns the signal number of the signal that terminated the child process.

`int WCOREDUMP (`*int* `status)` [Macro]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This macro returns a nonzero value if the child process terminated and produced a core dump.

`int WIFSTOPPED (`*int* `status)` [Macro]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This macro returns a nonzero value if the child process is stopped.

`int WSTOPSIG (`*int* `status)` [Macro]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

If `WIFSTOPPED` is true of *status*, this macro returns the signal number of the signal that caused the child process to stop.

## 26.8 BSD Process Wait Functions

The GNU C Library also provides these related facilities for compatibility with BSD Unix. BSD uses the `union wait` data type to represent status values rather than an `int`. The two representations are actually interchangeable; they describe the same bit patterns. The GNU C Library defines macros such as `WEXITSTATUS` so that they will work on either kind of object, and the `wait` function is defined to accept either type of pointer as its *status-ptr* argument.

These functions are declared in `sys/wait.h`.

`union wait` [Data Type]

This data type represents program termination status values. It has the following members:

`int w_termsig`
: The value of this member is the same as that of the `WTERMSIG` macro.

`int w_coredump`
: The value of this member is the same as that of the `WCOREDUMP` macro.

`int w_retcode`
: The value of this member is the same as that of the `WEXITSTATUS` macro.

`int w_stopsig`
: The value of this member is the same as that of the `WSTOPSIG` macro.

Instead of accessing these members directly, you should use the equivalent macros.

The `wait3` function is the predecessor to `wait4`, which is more flexible. `wait3` is now obsolete.

`pid_t` **`wait3`** (*union wait *`status-ptr`, int `options`, struct rusage *`usage`*) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

If *usage* is a null pointer, `wait3` is equivalent to `waitpid (-1, status-ptr, options)`.

If *usage* is not null, `wait3` stores usage figures for the child process in `*rusage` (but only if the child has terminated, not if it has stopped). See Section 22.1 [Resource Usage], page 628.

## 26.9 Process Creation Example

Here is an example program showing how you might write a function similar to the built-in `system`. It executes its *command* argument using the equivalent of '`sh -c command`'.

```
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

/* Execute the command using this shell program.  */
#define SHELL "/bin/sh"

int
my_system (const char *command)
{
  int status;
  pid_t pid;

  pid = fork ();
  if (pid == 0)
    {
      /* This is the child process.  Execute the shell command. */
      execl (SHELL, SHELL, "-c", command, NULL);
      _exit (EXIT_FAILURE);
    }
  else if (pid < 0)
    /* The fork failed.  Report failure.  */
    status = -1;
  else
    /* This is the parent process.  Wait for the child to complete.  */
    if (waitpid (pid, &status, 0) != pid)
      status = -1;
  return status;
}
```

There are a couple of things you should pay attention to in this example.

Remember that the first `argv` argument supplied to the program represents the name of the program being executed. That is why, in the call to `execl`, `SHELL` is supplied once to name the program to execute and a second time to supply a value for `argv[0]`.