Process System Calls

[**Process Creation Concepts**](http://www.chemie.fu-berlin.de/chemnet/use/info/libc/libc_toc.html#TOC405)

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. 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. 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.

[**Creating a Process**](http://www.chemie.fu-berlin.de/chemnet/use/info/libc/libc_toc.html#TOC407)

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

Function: pid\_t fork *(void)*

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.

[**Executing a File**](http://www.chemie.fu-berlin.de/chemnet/use/info/libc/libc_toc.html#TOC408)

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.

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'.

Function: int execlp *(const char \*filename, const char \*arg0, ...)*

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. In the GNU system, 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 syntax errors

If execution of the new file succeeds, it updates the access time field of the file as if the file had been read.

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.

[**Process Completion**](http://www.chemie.fu-berlin.de/chemnet/use/info/libc/libc_toc.html#TOC409)

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'.

Function: pid\_t wait *(int \*status-ptr)*

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)

File System Calls

**File descriptors**

The operating system assigns internally to each opened file a descriptor or an identifier (usually this is a positive integer). When opening or creating a new file the system returns a file descriptor to the process that executed the call. Each application has its own file descriptors. By convention, the first three file descriptors are opened at the beginning of each process. The 0 file descriptor identifies the standard input, 1 identifies the standard output and 2 the standard output for errors. The rest of the descriptors are used by the processes when opening an ordinary, pipe or special file, or directories. There are five system calls that generate file descriptors:*create*, *open*, *fcntl*, *dup* and *pipe*.

**System call OPEN**

Opening or creating a file can be done using the system call open. The syntax is:

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open(const char \*path,

   int flags,... /\* mode\_t mod \*/);

This function returns the file descriptor or in case of an error -1. The number of arguments that this function can have is two or three. The third argument is used only when creating a new file. When we want to open an existing file only two arguments are used. The function returns the smallest available file descriptor. This can be used in the following system calls: *read*, *write*, *lseek* and *close*. The effective UID or the effective GID of the process that executes the call has to have read/write rights, based on the value of the argument *flags*. The file pointer is places on the first byte in the file. The argument *flags* is formed by a bitwise OR operation made on the constants defined in the *fcntl.h* header.

O\_RDONLY

            Opens the file for reading.

O\_WRONLY

            Opens the file for writing.

O\_RDWR

            The file is opened for reading and writing.

O\_APPEND

            It writes successively to the end of the file.

O\_CREAT

            The file is created in case it didn’t already exist.

O\_EXCL

            If the file exists and O\_CREAT is positioned, calling *open* will fail.

O\_NONBLOCK

In the case of pipes and special files, this causes the open system call and any other future I/O operations to never block.

O\_TRUNC

            If the file exists all of its content will be deleted.

O\_SYNC

            It forces to write on the disk with function *write*. Though it slows down all the system, it can be useful in critical situations.

The third argument, *mod*, is a bitwise OR made between a combination of two from the following list:

S\_IRUSR, S\_IWUSR, S\_IXUSR

Owner: *read*, *write*, *execute*.

S\_IRGRP, S\_IWGRP, S\_IXGRP

Group: *read*, *write*, *execute*.

S\_IROTH, S\_IWOTH, S\_IXOTH

Others: *read*, *write*, *execute.*

The above define the access rights for a file and they are defined in the *sys/stat.h* header.

**System call CREAT**

A new file can be created by:

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int creat(const char \*path, mode\_t mod);

The function returns the file descriptor or in case of an error it returns the value -1. This call is equivalent with:

     open(path, O\_WRONLY | O\_CREAT | O\_TRUNC, mod);

The argument *path* specifies the name of the file, while *mod* defines the access rights. If the created file doesn’t exist, a new i-node is allocated and a link is made to this file from the directory it was created in. The owner of the process that executes the call - given by the effective UID and the effective GUID - must have writing permission in the directory. The open file will have the access rights that were specified in the second argument (see*umask*, too). The call returns the smallest file descriptor available. The file is opened for writing and its initial size is 0. The access time and the modification time are updated in the i-node. If the file exists (permission to search the directory is needed), it looses its contents and it will be opened for writing. The ownership and the access permissions won’t be modified. The second argument is ignored.

**System call READ**

When we want to read a certain number of bytes starting from the current position in a file, we use the *read* call. The syntax is:

#include <unistd.h>

ssize\_t read(int fd, void\* buf, size\_t noct);

The function returns the number of bytes read, 0 for end of file (EOF) and -1 in case an error occurred. It reads *noct* bytes from the open file referred by the *fd* descriptor and it puts it into a buffer *buf*. The pointer (current position) is incremented automatically after a reading that certain amount of bytes. The process that executes a read operation waits until the system puts the data from the disk into the buffer.

**System call WRITE**

For writing a certain number of bytes into a file starting from the current position we use the *write* call. Its syntax is:

#include <unistd.h>

ssize\_t write(int fd, const void\* buf, size\_t noct);

The function returns the number of bytes written and the value -1 in case of an error. It writes *noct* bytes from the buffer *buf* into the file that has as its descriptor *fd*. It is interesting to note that the actual writing onto the disk is delayed. This is done at the initiative of the root, without informing the user when it is done. If the process that did the call or an other process reads the data that haven’t been written on the disk yet, the system reads all this data out from the cache buffers. The delayed writing is faster, but it has three disadvantages:

a)      a disk error or a system error may cause loosing all the data

b)      a process that had the initiative of a write operation cannot be informed in case a writing error occurred

c)      the physical order of the write operations cannot be controlled.

To eliminate these disadvantages, in some cases the O\_SYNC is used. But as this slows down the system and considering the reliability of today’s systems it is better to use the mechanism which includes using cache buffers.

**System call CLOSE**

For closing a file and thus eliminating the assigned descriptor we use the system call *close*.

#include <unistd.h>

int close(int fd);

The function returns 0 in case of success and -1 in case of an error. At the termination of a process an open file is closed anyway.