

## fork() — Create a New Process

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

### Format

```
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t fork(void);
```

### Note:

Although POSIX.1 does not require that the `<sys/types.h>` include file be included, XPG4 has it as an optional header. Therefore, it is recommended that you include it for portability.

### General Description

Creates a new process. The new process (the *child process*) is an exact duplicate of the process that calls `fork()` (the *parent process*), except for the following:

- The child process has a unique process ID (PID) that does not match any active process group ID.
- The child has a different parent process ID, that is, the process ID of the process that called `fork()`.
- The child has its own copy of the parent's file descriptors. Each file descriptor in the child refers to the same open file description as the corresponding file descriptor in the parent.
- The child has its own copy of the parent's open directory streams. Each child's open directory stream can share directory stream positioning with the corresponding parent's directory stream.
- The following elements in the `tms` structure are set to 0 in the child:
  - `tms_utime`
  - `tms_stime`
  - `tms_cutime`
  - `tms_cstime`

For more information about these elements, see [times\(\) — Get Process and Child Process Times](#).

- The child does not inherit any file locks previously set by the parent.
- The child process has no alarms set (similar to the results of a call to `alarm()` with an argument value of 0).
- The child has no pending signals.
- The child process has only a single thread. That thread is a copy of the thread in the parent that called `fork()`. The child process has a different thread ID. If the parent process was multi-threaded (invoked `pthread_create()` at least once), the child process can only safely invoke async-signal-safe functions before it invokes an `exec()` family function. (This

restriction also applies to any process created as the result of the child invoking `fork()` before it invokes an `exec()` family function because the child process is still considered multi-threaded.) The child process does not inherit pthread attributes or pthread security environment. See [Table](#) for a list of async-signal-safe functions.

In all other respects, the child is identical to the parent. Because the child is a duplicate, it contains the same call to `fork()` that was in the parent. Execution begins with this `fork()` call, which returns a value of 0; the child then proceeds with normal execution.

The child address space inherits the following address space attributes of the parent address space:

- Region size
- Time limit

If the parent process is multi-threaded, it is the responsibility of the application to ensure that the application data is in a consistent state when the `fork()` occurs. For example, mutexes that are used to serialize updates to application data may need to be locked before the `fork()` and unlocked afterwards.

For more information on `fork()`, refer to [z/OS UNIX System Services Programming: Assembler Callable Services Reference](#).

You can use MVS memory files from a z/OS UNIX program. However, use of the `fork()` function from the program removes access from a hiperspace memory file for the child process. Use of an `exec` function from the program clears a memory file when the process address space is cleared.

The child process that results from a `fork()` in a multi-threaded environment can only invoke async-signal-safe functions.

An async-signal-safe function is defined as a function that may be invoked, without restriction, from signal-catching functions. All supported async-signal-safe functions are listed in [Table](#).

Table. Async-signal-safe library functions

<code>abort()</code>	<code>fpathconf()</code>	<code>raise()</code>	<code>sigpending()</code>
<code>accept()</code>	<code>fstat()</code>	<code>read()</code>	<code>sigprocmask()</code>
<code>access()</code>	<code>fsync()</code>	<code>readlink()</code>	<code>sigqueue()</code>
<code>aio_error()</code>	<code>ftruncate()</code>	<code>recv()</code>	<code>sigset()</code>
<code>aio_return()</code>	<code>getegid()</code>	<code>recvfrom()</code>	<code>sigsuspend()</code>
<code>aio_suspend()</code>	<code>geteuid()</code>	<code>recvmsg()</code>	<code>socket()</code>
<code>alarm()</code>	<code>getgid()</code>	<code>rename()</code>	<code>socketpair()</code>
<code>bind()</code>	<code>getgroups()</code>	<code>rmdir()</code>	<code>stat()</code>
<code>cfgetispeed()</code>	<code>getpeername()</code>	<code>select()</code>	<code>symlink()</code>
<code>cfgetospeed()</code>	<code>getpgrp()</code>	<code>send()</code>	<code>sysconf()</code>
<code>cfsetispeed()</code>	<code>getpid()</code>	<code>sendmsg()</code>	<code>tcdrain()</code>
<code>cfsetospeed()</code>	<code>getppid()</code>	<code>sendto()</code>	<code>tcflow()</code>
<code>chdir()</code>	<code>getsockname()</code>	<code>setgid()</code>	<code>tcflush()</code>
<code>chmod()</code>	<code>getsockopt()</code>	<code>setpgid()</code>	<code>tcgetattr()</code>
<code>chown()</code>	<code>getuid()</code>	<code>setsid()</code>	<code>tcgetpgrp()</code>
<code>close()</code>	<code>kill()</code>	<code>setsockopt()</code>	<code>tcsendbreak()</code>
<code>connect()</code>	<code>link()</code>	<code>setuid()</code>	<code>tcsetattr()</code>
<code>creat()</code>	<code>listen()</code>	<code>shutdown()</code>	<code>tcsetpgrp()</code>
<code>dup()</code>	<code>lseek()</code>	<code>sigaction()</code>	<code>time()</code>
<code>dup2()</code>	<code>lstat()</code>	<code>sigaddset()</code>	<code>times()</code>

execle()	mkdir()	sigdelset()	umask()
execve()	mkfifo()	sigemptyset()	uname()
_Exit()	open()	sigfillset()	unlink()
_exit()	pathconf()	sigismember()	utime()
fchmod()	pause()	sleep()	wait()
fchown()	pipe()	signal()	waitpid()
fcntl()	poll()	sigpause()	write()
fork()			

### Interoperability Restriction

For POSIX resources, fork() behaves as just described. But in general, MVS resources that existed in the parent do *not* exist in the child. This is true for open streams in MVS data sets and assembler-accessed MVS facilities, such as TIMERS. In addition, MVS allocations (through JCL, SVC99, or ALLOCATE) are not passed to the child process.

### Special Behavior for z/OS UNIX Services

Notes:

1. A prior loaded copy of an HFS program in the same address space is reused under the same circumstances that apply to the reuse of a prior loaded MVS unauthorized program from an unauthorized library by the MVS XCTL service with the following exceptions:
  - If the calling process is in Ptrace debug mode, a prior loaded copy is not reused.
  - If the calling process is not in Ptrace debug mode, but the only prior loaded usable copy found of the HFS program is in storage modifiable by the caller, the prior copy is not reused.
2. If the specified file name represents an external link or a sticky bit file, the program is loaded from the caller's MVS load library search order. For an external link, the external name is only used if the name is eight characters or less, otherwise the caller receives an error from the loadhfs service. For a sticky bit program, the file name is used if it is eight characters or less. Otherwise, the program is loaded from the HFS.
3. If the calling task is in a WLM enclave, the resulting task in the new process image is joined to the same WLM enclave. This allows WLM to manage the old and new process images as one 'business unit of work' entity for system accounting and management purposes.

### Returned Value

If successful, fork() returns 0 to the child process and the process ID of the newly created child to the parent process.

If unsuccessful, fork() fails to create a child process, returns -1 to the parent, and sets errno to one of the following values:

Error Code

Description

EAGAIN

There are insufficient resources to create another process, or the process has already reached the maximum number of processes you can run.

ELEMSGERR

Language Environment message file not available.

ELEMULTITHREAD

Application contains a language that does not support fork() in a multithreaded environment, or the multithreaded fork() is being attempted while running in a Language Environment preinitialization (CEEPIPI) environment.

ELENOFORK

Application contains a language that does not support fork().

ENOMEM

The process requires more space than is available.