29260 **NAME**

29261 fork — create a new process

29262 **SYNOPSIS**

29263
```
#include <unistd.h>
```

29264
```
pid_t fork(void);
```

29265 **DESCRIPTION**

29266 The *fork*( ) function shall create a new process. The new process (child process) shall be an exact
29267 copy of the calling process (parent process) except as detailed below:

29268 • The child process shall have a unique process ID.

29269 • The child process ID also shall not match any active process group ID.

29270 • The child process shall have a different parent process ID, which shall be the process ID of
29271 the calling process.

29272 • The child process shall have its own copy of the parent's file descriptors. Each of the
29273 child's file descriptors shall refer to the same open file description with the corresponding
29274 file descriptor of the parent.

29275 • The child process shall have its own copy of the parent's open directory streams. Each
29276 open directory stream in the child process may share directory stream positioning with the
29277 corresponding directory stream of the parent.

29278 • The child process shall have its own copy of the parent's message catalog descriptors.

29279 • The child process values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* shall be set to
29280 0.

29281 • The time left until an alarm clock signal shall be reset to zero, and the alarm, if any, shall be
29282 canceled; see *alarm*( ).

29283 XSI • All *semadj* values shall be cleared.

29284 • File locks set by the parent process shall not be inherited by the child process.

29285 • The set of signals pending for the child process shall be initialized to the empty set.

29286 XSI • Interval timers shall be reset in the child process.

29287 • Any semaphores that are open in the parent process shall also be open in the child process.

29288 ML • The child process shall not inherit any address space memory locks established by the
29289 parent process via calls to *mlockall*( ) or *mlock*( ).

29290 • Memory mappings created in the parent shall be retained in the child process.
29291 MAP_PRIVATE mappings inherited from the parent shall also be MAP_PRIVATE
29292 mappings in the child, and any modifications to the data in these mappings made by the
29293 parent prior to calling *fork*( ) shall be visible to the child. Any modifications to the data in
29294 MAP_PRIVATE mappings made by the parent after *fork*( ) returns shall be visible only to
29295 the parent. Modifications to the data in MAP_PRIVATE mappings made by the child shall
29296 be visible only to the child.

29297 PS • For the SCHED_FIFO and SCHED_RR scheduling policies, the child process shall inherit
29298 the policy and priority settings of the parent process during a *fork*( ) function. For other
29299 scheduling policies, the policy and priority settings on *fork*( ) are implementation-defined.

- Per-process timers created by the parent shall not be inherited by the child process.

MSG

- The child process shall have its own copy of the message queue descriptors of the parent. Each of the message descriptors of the child shall refer to the same open message queue description as the corresponding message descriptor of the parent.

- No asynchronous input or asynchronous output operations shall be inherited by the child process. Any use of asynchronous control blocks created by the parent produces undefined behavior.

- A process shall be created with a single thread. If a multi-threaded process calls *fork*( ), the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal-safe operations until such time as one of the *exec* functions is called. Fork handlers may be established by means of the *pthread_atfork*( ) function in order to maintain application invariants across *fork*( ) calls.

  When the application calls *fork*( ) from a signal handler and any of the fork handlers registered by *pthread_atfork*( ) calls a function that is not async-signal-safe, the behavior is undefined.

OB TRC TRI

- If the Trace option and the Trace Inherit option are both supported:

  If the calling process was being traced in a trace stream that had its inheritance policy set to POSIX_TRACE_INHERITED, the child process shall be traced into that trace stream, and the child process shall inherit the parent's mapping of trace event names to trace event type identifiers. If the trace stream in which the calling process was being traced had its inheritance policy set to POSIX_TRACE_CLOSE_FOR_CHILD, the child process shall not be traced into that trace stream. The inheritance policy is set by a call to the *posix_trace_attr_setinherited*( ) function.

OB TRC

- If the Trace option is supported, but the Trace Inherit option is not supported:

  The child process shall not be traced into any of the trace streams of its parent process.

OB TRC

- If the Trace option is supported, the child process of a trace controller process shall not control the trace streams controlled by its parent process.

CPT

- The initial value of the CPU-time clock of the child process shall be set to zero.

TCT

- The initial value of the CPU-time clock of the single thread of the child process shall be set to zero.

All other process characteristics defined by POSIX.1-2008 shall be the same in the parent and child processes. The inheritance of process characteristics not defined by POSIX.1-2008 is unspecified by POSIX.1-2008.

After *fork*( ), both the parent and the child processes shall be capable of executing independently before either one terminates.

RETURN VALUE

Upon successful completion, *fork*( ) shall return 0 to the child process and shall return the process ID of the child process to the parent process. Both processes shall continue to execute from the *fork*( ) function. Otherwise, –1 shall be returned to the parent process, no child process shall be created, and *errno* shall be set to indicate the error.

29341 **ERRORS**

29342 The *fork*( ) function shall fail if:

29343 [EAGAIN] The system lacked the necessary resources to create another process, or the
29344 system-imposed limit on the total number of processes under execution
29345 system-wide or by a single user {CHILD_MAX} would be exceeded.

29346 The *fork*( ) function may fail if:

29347 [ENOMEM] Insufficient storage space is available.

29348 **EXAMPLES**

29349 None.

29350 **APPLICATION USAGE**

29351 None.

29352 **RATIONALE**

29353 Many historical implementations have timing windows where a signal sent to a process group
29354 (for example, an interactive SIGINT) just prior to or during execution of *fork*( ) is delivered to the
29355 parent following the *fork*( ) but not to the child because the *fork*( ) code clears the child's set of
29356 pending signals. This volume of POSIX.1-2008 does not require, or even permit, this behavior.
29357 However, it is pragmatic to expect that problems of this nature may continue to exist in
29358 implementations that appear to conform to this volume of POSIX.1-2008 and pass available
29359 verification suites. This behavior is only a consequence of the implementation failing to make
29360 the interval between signal generation and delivery totally invisible. From the application's
29361 perspective, a *fork*( ) call should appear atomic. A signal that is generated prior to the *fork*( )
29362 should be delivered prior to the *fork*( ). A signal sent to the process group after the *fork*( ) should
29363 be delivered to both parent and child. The implementation may actually initialize internal data
29364 structures corresponding to the child's set of pending signals to include signals sent to the
29365 process group during the *fork*( ). Since the *fork*( ) call can be considered as atomic from the
29366 application's perspective, the set would be initialized as empty and such signals would have
29367 arrived after the *fork*( ); see also **<signal.h>**.

29368 One approach that has been suggested to address the problem of signal inheritance across *fork*( )
29369 is to add an [EINTR] error, which would be returned when a signal is detected during the call.
29370 While this is preferable to losing signals, it was not considered an optimal solution. Although it
29371 is not recommended for this purpose, such an error would be an allowable extension for an
29372 implementation.

29373 The [ENOMEM] error value is reserved for those implementations that detect and distinguish
29374 such a condition. This condition occurs when an implementation detects that there is not enough
29375 memory to create the process. This is intended to be returned when [EAGAIN] is inappropriate
29376 because there can never be enough memory (either primary or secondary storage) to perform
29377 the operation. Since *fork*( ) duplicates an existing process, this must be a condition where there is
29378 sufficient memory for one such process, but not for two. Many historical implementations
29379 actually return [ENOMEM] due to temporary lack of memory, a case that is not generally
29380 distinct from [EAGAIN] from the perspective of a conforming application.

29381 Part of the reason for including the optional error [ENOMEM] is because the SVID specifies it
29382 and it should be reserved for the error condition specified there. The condition is not applicable
29383 on many implementations.

29384 IEEE Std 1003.1-1988 neglected to require concurrent execution of the parent and child of *fork*( ).
29385 A system that single-threads processes was clearly not intended and is considered an
29386 unacceptable ''toy implementation'' of this volume of POSIX.1-2008. The only objection
29387 anticipated to the phrase ''executing independently'' is testability, but this assertion should be

testable. Such tests require that both the parent and child can block on a detectable action of the other, such as a write to a pipe or a signal. An interactive exchange of such actions should be possible for the system to conform to the intent of this volume of POSIX.1-2008.

The [EAGAIN] error exists to warn applications that such a condition might occur. Whether it occurs or not is not in any practical sense under the control of the application because the condition is usually a consequence of the user's use of the system, not of the application's code. Thus, no application can or should rely upon its occurrence under any circumstances, nor should the exact semantics of what concept of "user" is used be of concern to the application developer. Validation writers should be cognizant of this limitation.

There are two reasons why POSIX programmers call *fork*( ). One reason is to create a new thread of control within the same program (which was originally only possible in POSIX by creating a new process); the other is to create a new process running a different program. In the latter case, the call to *fork*( ) is soon followed by a call to one of the *exec* functions.

The general problem with making *fork*( ) work in a multi-threaded world is what to do with all of the threads. There are two alternatives. One is to copy all of the threads into the new process. This causes the programmer or implementation to deal with threads that are suspended on system calls or that might be about to execute system calls that should not be executed in the new process. The other alternative is to copy only the thread that calls *fork*( ). This creates the difficulty that the state of process-local resources is usually held in process memory. If a thread that is not calling *fork*( ) holds a resource, that resource is never released in the child process because the thread whose job it is to release the resource does not exist in the child process.

When a programmer is writing a multi-threaded program, the first described use of *fork*( ), creating new threads in the same program, is provided by the *pthread_create*( ) function. The *fork*( ) function is thus used only to run new programs, and the effects of calling functions that require certain resources between the call to *fork*( ) and the call to an *exec* function are undefined.

The addition of the *forkall*( ) function to the standard was considered and rejected. The *forkall*( ) function lets all the threads in the parent be duplicated in the child. This essentially duplicates the state of the parent in the child. This allows threads in the child to continue processing and allows locks and the state to be preserved without explicit *pthread_atfork*( ) code. The calling process has to ensure that the threads processing state that is shared between the parent and child (that is, file descriptors or MAP_SHARED memory) behaves properly after *forkall*( ). For example, if a thread is reading a file descriptor in the parent when *forkall*( ) is called, then two threads (one in the parent and one in the child) are reading the file descriptor after the *forkall*( ). If this is not desired behavior, the parent process has to synchronize with such threads before calling *forkall*( ).

While the *fork*( ) function is async-signal-safe, there is no way for an implementation to determine whether the fork handlers established by *pthread_atfork*( ) are async-signal-safe. The fork handlers may attempt to execute portions of the implementation that are not async-signal-safe, such as those that are protected by mutexes, leading to a deadlock condition. It is therefore undefined for the fork handlers to execute functions that are not async-signal-safe when *fork*( ) is called from a signal handler.

When *forkall*( ) is called, threads, other than the calling thread, that are in functions that can return with an [EINTR] error may have those functions return [EINTR] if the implementation cannot ensure that the function behaves correctly in the parent and child. In particular, *pthread_cond_wait*( ) and *pthread_cond_timedwait*( ) need to return in order to ensure that the condition has not changed. These functions can be awakened by a spurious condition wakeup rather than returning [EINTR].

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *exec*, *fcntl*( ), *posix_trace_attr_getinherited*( ), *posix_trace_eventid_equal*( ), *pthread_atfork*( ), *semop*( ), *signal*( ), *times*( )

XBD Section 4.11 (on page 110), **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is changed for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effect of *fork*( ) on a pending alarm call in the child process is clarified.

The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/17 is applied, adding text to the DESCRIPTION and RATIONALE relating to fork handlers registered by the *pthread_atfork*( ) function and async-signal safety.

Issue 7

Austin Group Interpretation 1003.1-2001 #080 is applied, clarifying the status of asynchronous input and asynchronous output operations and asynchronous control lists in the DESCRIPTION.

Functionality relating to the Asynchronous Input and Output, Memory Mapped Files, Timers, and Threads options is moved to the Base.

Functionality relating to message catalog descriptors is moved from the XSI option to the Base.

