fork - Man Page

create a new process

Пролог

Эта страница руководства является частью Руководства программиста POSIX. Реализация этого интерфейса в Linux может отличаться (обратитесь к соответствующей странице руководства Linux для получения подробной информации о поведении Linux), или интерфейс может быть не реализован в Linux.

Краткое описание

```
#include <unistd.h>

pid_t fork (void);
```

Описание

Функция *fork*() должна создать новый процесс. Новый процесс (дочерний процесс) должен быть точной копией вызывающего процесса (родительского процесса), за исключением случаев, описанных ниже:

* Дочерний процесс должен иметь уникальный идентификатор процесса.

* Идентификатор дочернего процесса также не должен совпадать с идентификатором активной группы процессов.

* Дочерний процесс должен иметь другой идентификатор родительского процесса, который должен быть идентификатором процесса вызывающего процесса.

* Дочерний процесс должен иметь свою собственную копию файловых дескрипторов родителя. Каждый из дочерних файловых дескрипторов должен ссылаться на одно и то же описание открытого файла с соответствующим файловым дескриптором родителя.

* Дочерний процесс должен иметь свою собственную копию потоков открытых каталогов родителя. Каждый открытый поток каталогов в дочернем процессе может совместно использовать позиционирование потока каталогов с соответствующим потоком каталогов родительского процесса.

* Дочерний процесс должен иметь свою собственную копию дескрипторов родительского каталога сообщений.

* Значения дочерних процессов *tms_utime*, *tms_stime*, *tms_cutime*и *tms_cstime* должны быть равны 0.

* Время, оставшееся до сигнала будильника, сбрасывается на ноль, а сигнал тревоги, если таковой имеется, отменяется; см. *alarm*() .

fork - Man Page

^ Блокировки файлов, установленные родительским процессом, не наследуются дочерним процессом.

* Набор сигналов, ожидающих дочернего процесса, должен быть инициализирован пустым набором.

* Интервальные таймеры должны быть сброшены в дочернем процессе.

* Любые семафоры, открытые в родительском процессе, также должны быть открыты в дочернем процессе.

* Дочерний процесс не должен наследовать блокировки памяти адресного пространства, установленные родительским процессом с помощью вызовов *mlockall*() или *mlock*() .

* Сопоставления памяти, созданные в родительском процессе, должны быть сохранены в дочернем процессе. Сопоставления MAP_PRIVATE, унаследованные от родителя, также должны быть сопоставлениями MAP_PRIVATE в дочернем элементе, и любые изменения данных в этих сопоставлениях, сделанные родителем до вызова *fork*(), должны быть видны дочернему элементу. Любые изменения данных в сопоставлениях MAP_PRIVATE, сделанные родителем после *возврата fork*(), должны быть видны только родителю. Изменения данных в сопоставлениях MAP_PRIVATE, внесенные дочерним элементом, должны быть видны только дочернему элементу.

* Для политик планирования SCHED_FIFO и SCHED_RR дочерний процесс наследует настройки политики и приоритета родительского процесса во *время функции fork*(). Для других политик планирования параметры политики и приоритета *fork*() определяются реализацией.

* Таймеры для каждого процесса, созданные родителем, не наследуются дочерним процессом.

* Дочерний процесс должен иметь свою собственную копию дескрипторов очереди сообщений родительского. Каждый дескриптор сообщения дочернего элемента должен ссылаться на то же описание открытой очереди сообщений, что и соответствующий дескриптор сообщения родительского элемента.

* Дочерний процесс не должен наследовать операции асинхронного ввода или асинхронного вывода. Любое использование асинхронных блоков управления, созданных родителем, приводит к неопределенному поведению.

* Процесс должен быть создан с одним потоком. Если многопоточный процесс вызывает *fork*() , новый процесс должен содержать реплику вызывающего потока и все его адресное пространство, возможно, включая состояния мьютексов и других ресурсов. Следовательно, чтобы избежать ошибок, дочерний процесс может выполнять операции async-signal-safe только до тех пор, пока не будет вызвана одна из функций exec.

  Когда приложение вызывает *fork*() из обработчика сигнала и любой из обработчиков fork, зарегистрированных *pthread_atfork*(), вызывает функцию, которая не является асинхронно-сигнальной, поведение не определено.

fork - Man Page

Если вызывающий процесс отслеживался в потоке трассировки, для которого была установлена политика наследования POSIX_TRACE_INHERITED, дочерний процесс должен быть прослежен в этом потоке трассировки, и дочерний процесс должен наследовать родительское сопоставление имен событий трассировки с идентификаторами типов событий трассировки. Если для потока трассировки, в котором отслеживался вызывающий процесс, была установлена политика наследования POSIX_TRACE_CLOSE_FOR_CHILD , дочерний процесс не должен отслеживаться в этом потоке трассировки. Политика наследования задается вызовом функции *posix_trace_attr_setinherited*() .

* Если опция трассировки поддерживается, но опция наследования трассировки не поддерживается:

  Дочерний процесс не должен отслеживаться ни в одном из потоков трассировки его родительского процесса.

* Если опция трассировки поддерживается, дочерний процесс процесса контроллера трассировки не должен контролировать потоки трассировки, контролируемые его родительским процессом.

* Начальное значение часов процессорного времени дочернего процесса должно быть равно нулю.

* Начальное значение часов процессорного времени одного потока дочернего процесса должно быть равно нулю.

Все остальные характеристики процесса, определенные POSIX.1-2008, должны быть одинаковыми в родительском и дочернем процессах. Наследование характеристик процесса, не определенных POSIX.1-2008, не определено POSIX.1-2008.

После *fork*() и родительский, и дочерний процессы должны быть способны выполняться независимо друг от друга до завершения любого из них.

Возвращаемое значение

После успешного завершения *fork*() возвращает 0 дочернему процессу и возвращает идентификатор процесса дочернего процесса родительскому процессу. Оба процесса должны продолжать выполняться из функции *fork*() . В противном случае родительскому процессу возвращается значение -1, дочерний процесс не создается, а *errno* указывает на ошибку.

Ошибки

Функция *fork*() завершится ошибкой, если:

EAGAIN

Системе не хватало необходимых ресурсов для создания другого процесса, или установленный системой лимит на общее количество процессов,

fork - Man Page

Функция *fork*() может выйти из строя, если:

ENOMEM

Недостаточно места для хранения.

Следующие разделы являются информативными.

Примеры

Нет.

Использование приложений

Нет.

Обоснование

Многие исторические реализации имеют временные окна, в которых сигнал, отправленный группе процессов (например, интерактивному SIGINT) непосредственно перед или во время выполнения *fork*(), доставляется родителю после *fork*(), но не дочернему элементу, потому *что fork*() код очищает дочерний набор ожидающих сигналов. Этот том POSIX.1-2017 не требует и даже не разрешает такое поведение. Однако прагматично ожидать, что проблемы такого рода могут продолжать существовать в реализациях, которые, как представляется, соответствуют этому объему POSIX.1-2017 и проходят доступные наборы проверки. Такое поведение является лишь следствием того, что реализация не смогла сделать интервал между генерацией и доставкой сигнала полностью невидимым. С точки зрения приложения *вызов fork*() должен выглядеть атомарным. Сигнал, который генерируется до *fork*() должен быть доставлен до *fork*() . Сигнал, отправленный в группу процессов после *fork*(), должен быть доставлен как родительскому, так и дочернему элементу. Реализация может фактически инициализировать внутренние структуры данных, соответствующие дочернему набору ожидающих сигналов, чтобы включить сигналы, отправленные группе процессов во время *fork*(). Поскольку *вызов fork*() можно рассматривать как атомарный с точки зрения приложения, набор будет инициализирован как пустой, и такие сигналы поступят после *fork*(); см. Также *<signal.h>*.

Один из подходов, который был предложен для решения проблемы наследования сигнала через *fork*(), заключается в добавлении **ошибки [EINTR]** , которая будет возвращена при обнаружении сигнала во время вызова. Хотя это предпочтительнее потери сигналов, это не считалось оптимальным решением. Хотя это не рекомендуется для этой цели, такая ошибка будет допустимым расширением для реализации.

Значение **ошибки [ENOMEM]** зарезервировано для тех реализаций, которые обнаруживают и различают такое условие. Это условие возникает, когда

fork - Man Page

выполнения операции. Поскольку *fork*() дублирует существующий процесс, это должно быть условие, при котором достаточно памяти для одного такого процесса, но не для двух. Многие исторические реализации фактически возвращают **[ENOMEM]** из-за временной нехватки памяти случай, который обычно не отличается от **[EAGAIN]** с точки зрения соответствующего приложения.

Часть причины включения необязательной ошибки **[ENOMEM]** заключается в том, что SVID указывает ее, и она должна быть зарезервирована для указанного там условия ошибки. Условие неприменимо во многих реализациях.

IEEE Std 1003.1-1988 пренебрег требованием одновременного выполнения родителя и потомка *fork*(). Система, которая обрабатывает однопоточные процессы, явно не предназначалась и считается неприемлемой “игрушечной реализацией” этого тома POSIX.1-2017. Единственное возражение, ожидаемое от фразы “выполнение независимо”, - это тестируемость, но это утверждение должно быть проверяемым. Такие тесты требуют, чтобы и родитель, и потомок могли блокировать обнаруживаемое действие другого, например запись в канал или сигнал. Интерактивный обмен такими действиями должен быть возможен, чтобы система соответствовала намерениям этого тома POSIX.1-2017.

Ошибка **[EAGAIN]** существует для предупреждения приложений о том, что такое условие может возникнуть. Независимо от того, происходит это или нет, это не в каком-либо практическом смысле под контролем приложения, потому что условие обычно является следствием использования пользователем системы, а не кода приложения. Таким образом, ни одно приложение не может и не должно полагаться на его появление ни при каких обстоятельствах, и при этом точная семантика того, какое понятие “пользователь” используется, не должна беспокоить разработчика приложения. Авторы проверки должны знать об этом ограничении.

Есть две причины, по которым программисты POSIX вызывают *fork*() . Одна из причин - создать новый поток управления в той же программе (что изначально было возможно только в POSIX путем создания нового процесса); другая - создать новый процесс, выполняющий другую программу. В последнем случае вызов *fork*() вскоре сопровождается вызовом одной из функций *exec*.

Общая проблема с работой *fork*() в многопоточном мире заключается в том, что делать со всеми потоками. Есть две альтернативы. Один из них - скопировать все потоки в новый процесс. Это приводит к тому, что программист или реализация имеют дело с потоками, которые приостановлены на системных вызовах или которые могут выполнять системные вызовы, которые не должны выполняться в новом процессе. Другой альтернативой является копирование только потока, который вызывает *fork*() . Это создает трудность в том, что состояние локальных ресурсов процесса обычно хранится в памяти процесса. Если поток, который не вызывает *fork*() содержит ресурс, который никогда не освобождается в дочернем процессе, потому что поток, задачей которого является освобождение ресурса, не существует в дочернем процессе.

fork - Man Page

обеспечивается функцией *pthread_create*() . Таким образом, функция fork() используется только для запуска новых программ, а эффекты вызова функций, требующих определенных ресурсов между вызовом *fork*() и вызовом функции exec, не определены.

Добавление функции *forkall*() в стандарт было рассмотрено и отклонено. Функция *forkall*() позволяет дублировать все потоки в родительском файле в дочернем. Это по существу дублирует состояние родителя в потомке. Это позволяет потокам в дочернем элементе продолжать обработку и позволяет сохранять блокировки и состояние без явного *кода pthread_atfork*(). Вызывающий процесс должен убедиться, что состояние обработки потоков, которое является общим между родителем и дочерним (то есть файловыми дескрипторами или MAP_SHARED памятью), ведет себя правильно после *forkall*() . Например, если поток читает дескриптор файла в родителе при *вызове forkall*() , то два потока (один в родителе и один в потомке) читают дескриптор файла после *forkall*() . Если это нежелательное поведение, родительский процесс должен синхронизироваться с такими потоками перед вызовом *forkall*() .

Хотя функция *fork*() является асинхронно-сигнальной, реализация не может определить, являются ли обработчики fork, установленные *pthread_atfork*(), асинхронно-сигнальными. Обработчики fork могут пытаться выполнить части реализации, которые не являются асинхронными сигналами, например те, которые защищены мьютексами, что приводит к условию взаимоблокировки. Поэтому не определено, чтобы обработчики fork выполняли функции, которые не являются async-signal-safe, когда *fork*() вызывается из обработчика сигнала.

При *вызове forkall*() потоки, отличные от вызывающего потока, которые находятся в функциях, которые могут возвращаться с **ошибкой [EINTR]**, могут возвращать эти функции **[EINTR]**, если реализация не может гарантировать, что функция ведет себя правильно в родительском и дочернем элементах. В частности, *pthread_cond_wait*() и *pthread_cond_timedwait*() необходимо вернуть, чтобы убедиться, что условие не изменилось. Эти функции могут быть пробуждены ложным пробуждением условия, а не возвращением **[EINTR]**.

Будущие направления

Нет.

См. Также

сигнализация(), исполнитель, fcntl(), posix_trace_attr_getinherited(), posix_trace_eventid_equal(), pthread_atfork(), semop(), сигнал(), times()

Том базовых определений POSIX.1-2017, *Раздел 4.12*, *Синхронизация памяти*, <sys_types.h>, <unistd.h>

fork - Man Page

Ссылка на

aio_error(3p), aio_read(3p), aio_return(3p), aio_write(3p), alarm(3p), close(3p), exec(3p), getpgid(3p), getpgrp(3p), getpid(3p), getppid(3p), getrlimit(3p), getsid(3p), lio_listio(3p), mlock(3p), mlockall(3p), mmap(3p), pclose(3p), popen(3p), posix_spawn(3p), posix_trace_attr_getinherited(3p), pthread_atfork(3p), pthread_create(3p), semop(3p), setpgrp(3p), sh(1p), shmat(3p), shmdt(3p), times(3p), unistd.h(0p), wait(3p).

2017 IEEE/The Open Group POSIX Programmer's Manual

fork - Man Page