NAME

pidfd_open − obtain a file descriptor that refers to a process

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

int syscall(SYS_pidfd_open, pid_t *pid***, unsigned int** *flags***);**

Note: glibc provides no wrapper for **pidfd_open**(), necessitating the use of **syscall**(2).

DESCRIPTION

The **pidfd_open**() system call creates a file descriptor that refers to the process whose PID is specified in *pid*. The file descriptor is returned as the function result; the close-on-exec flag is set on the file descriptor.

The *flags* argument either has the value 0, or contains the following flag:

PIDFD_NONBLOCK (since Linux 5.10)
Return a nonblocking file descriptor. If the process referred to by the file descriptor has not yet terminated, then an attempt to wait on the file descriptor using **waitid**(2) will immediately return the error **EAGAIN** rather than blocking.

RETURN VALUE

On success, **pidfd_open**() returns a file descriptor (a nonnegative integer). On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EINVAL
flags is not valid.

EINVAL
pid is not valid.

EMFILE
The per-process limit on the number of open file descriptors has been reached (see the description of **RLIMIT_NOFILE** in **getrlimit**(2)).

ENFILE
The system-wide limit on the total number of open files has been reached.

ENODEV
The anonymous inode filesystem is not available in this kernel.

ENOMEM
Insufficient kernel memory was available.

ESRCH
The process specified by *pid* does not exist.

VERSIONS

pidfd_open() first appeared in Linux 5.3.

STANDARDS

pidfd_open() is Linux specific.

NOTES

The following code sequence can be used to obtain a file descriptor for the child of **fork**(2):

```
pid = fork();
if (pid > 0) {     /* If parent */
    pidfd = pidfd_open(pid, 0);
    ...
}
```

Even if the child has already terminated by the time of the **pidfd_open**() call, its PID will not have been recycled and the returned file descriptor will refer to the resulting zombie process. Note, however, that this is guaranteed only if the following conditions hold true:

- the disposition of **SIGCHLD** has not been explicitly set to **SIG_IGN** (see **sigaction**(2));
- the **SA_NOCLDWAIT** flag was not specified while establishing a handler for **SIGCHLD** or while setting the disposition of that signal to **SIG_DFL** (see **sigaction**(2)); and
- the zombie process was not reaped elsewhere in the program (e.g., either by an asynchronously executed signal handler or by **wait**(2) or similar in another thread).

If any of these conditions does not hold, then the child process (along with a PID file descriptor that refers to it) should instead be created using **clone**(2) with the **CLONE_PIDFD** flag.

Use cases for PID file descriptors

A PID file descriptor returned by **pidfd_open**() (or by **clone**(2) with the **CLONE_PID** flag) can be used for the following purposes:

- The **pidfd_send_signal**(2) system call can be used to send a signal to the process referred to by a PID file descriptor.
- A PID file descriptor can be monitored using **poll**(2), **select**(2), and **epoll**(7). When the process that it refers to terminates, these interfaces indicate the file descriptor as readable. Note, however, that in the current implementation, nothing can be read from the file descriptor (**read**(2) on the file descriptor fails with the error **EINVAL**).
- If the PID file descriptor refers to a child of the calling process, then it can be waited on using **waitid**(2).
- The **pidfd_getfd**(2) system call can be used to obtain a duplicate of a file descriptor of another process referred to by a PID file descriptor.
- A PID file descriptor can be used as the argument of **setns**(2) in order to move into one or more of the same namespaces as the process referred to by the file descriptor.
- A PID file descriptor can be used as the argument of **process_madvise**(2) in order to provide advice on the memory usage patterns of the process referred to by the file descriptor.

The **pidfd_open**() system call is the preferred way of obtaining a PID file descriptor for an already existing process. The alternative is to obtain a file descriptor by opening a */proc/[pid]* directory. However, the latter technique is possible only if the **proc**(5) filesystem is mounted; furthermore, the file descriptor obtained in this way is *not* pollable and can't be waited on with **waitid**(2).

EXAMPLES

The program below opens a PID file descriptor for the process whose PID is specified as its command-line argument. It then uses**poll**(2) to monitor the file descriptor for process e xit, as indicated by an **EPOLLIN** event.

Program source

```
#define _GNU_SOURCE
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>

static int
pidfd_open(pid_t pid, unsigned int flags)
{
    return syscall(SYS_pidfd_open, pid, flags);
}
```

```
int
main(int argc, char *argv[])
{
    int            pidfd, ready;
    struct pollfd  pollfd;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pid>\n", argv[0]);
        exit(EXIT_SUCCESS);
    }

    pidfd = pidfd_open(atoi(argv[1]), 0);
    if (pidfd == −1) {
        perror("pidfd_open");
        exit(EXIT_FAILURE);
    }

    pollfd.fd = pidfd;
    pollfd.events = POLLIN;

    ready = poll(&pollfd, 1, −1);
    if (ready == −1) {
        perror("poll");
        exit(EXIT_FAILURE);
    }

    printf("Events (%#x): POLLIN is %sset\n", pollfd.revents,
           (pollfd.revents & POLLIN) ? "" : "not ");

    close(pidfd);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

clone(2), **kill**(2), **pidfd_getfd**(2), **pidfd_send_signal**(2), **poll**(2), **process_madvise**(2), **select**(2), **setns**(2), **waitid**(2), **epoll**(7)