Add Support for pidfd File Descriptors

Mentors: Alexander Mikhalitsyn, Christian Brauner

May 11, 2025

Contents

1 Abstract

The Linux kernel uses **process IDs (PID)** to identify processes. A **PID** is just an integer (its maximum value is `4194304` on my machine). Consider a process (let's say process **X**) that uses a PID to identify a process (let's say process **A**). Suppose that **A** dies. If a large number of processes are being created, the PID of **A** might get reassigned to a different process. Now, process **X** ends up communicating with the wrong process. This reassignment of the PID of **A** is known as PID recycling.

PID recycling creates a problem for any tool using PIDs to identify processes since we cannot *always* ensure that a PID points to the same process. To solve the problem of PID recycling, **pidfds** were introduced in the Linux kernel. A pidfd is a file descriptor that refers to a process

and we can use it to send signals to that process. pidfds ensure that the signal is being sent to the correct process. CRIU cannot dump/restore on processes that use **pidfds**. This solution is expected to add support for pidfds.

2 Technical Details

2.1 `pidfd` Details

A `pidfd` references the kernel's `struct pid`.
Here's how an entry of a `pidfd` looks like in `/proc/$pid/fdinfo/$number`:

```
pos:    0
flags:  02000002
mnt_id: 16
ino:    2102
Pid:    71937
NSpid:  71937
```

Here Pid and NSpid are *unique* to `pidfds`. So, while parsing `/proc/$pid/fdinfo/` if we find a Pid and/or NSpid entry we can assume that it is a pidfd.

- `Pid:` Refers to the `pid` of the process the `pidfd` points to.
- `NSpid:` Refers to the `pid` of the process the `pidfd` points to in its respective namespace.

There are three system calls relating to `pidfds`:

1. `pidfd_open`: `int syscall(SYS_pidfd_open, pid_t pid, int flags)`
   creates a `pidfd` using a existing `pid`, `returns -1` if no process exists with that `pid`.
2. `int pidfd_send_signal: syscall(SYS_pidfd_send_signal, pid_t pid, int flags)`
   sends a signal to the process specified by the `pidfd`.
3. `pidfd_getfd: int syscall(SYS_pidfd_get_fd, pid_t pid, int targetfd, int flags)`
   obtain a duplicate of another process's file descriptor.

We can also use `poll`, `select` or `epoll` to wait on a `pidfd`.

2.2 Checkpointing

During Checkpointing, CRIU saves all the information related to the process tree (memory maps, file descriptors, pipe paramaters, etc). It gets all this information from the `/proc` file system. There are three steps for checkpointing:

1. Collect process tree and freeze it
2. Collect tasks' resources and dump them (Dumping `fds, VMAs, etc`)
3. Cleanup

All this data is stored in a set of image files. These image files can be of 3 types:

- CRIU specific files in google protobuf format
- CRIU specific files with binary data in it
- image files in third party format

2.2.1 Checkpointing Files

For information related to `fds`, it parses the `/proc/$pid/fdinfo/$number` and `/proc/$pid/fd/$number` directories. We can do this for `pidfds` as well.
Information about open file descriptors for each process is stored in a `fdinfo-$id.img` file (it has PB data in it). Information from `/proc/$pid/fdinfo/$number` is stored in a `files.img` file. We can extend this to store information related to `pidfds` as well.

All the information CRIU needs to store about files:

- FD numbers
- File Sharing (A child process might share a fd with it's parent)
- Determining Inode Type
- State of File and Inode

2.3 Restoring

The restore procedure is done by CRIU morphing itself into the tasks it restores. It involves four steps:

1. Resolve shared resources
2. Fork the process tree
3. Restore basic tasks resources
4. Switch to restorer context, restore the rest and continue

2.3.1 Restoring Files

For restoring opened files, CRIU needs to first open the file and then assign it the desired file descriptor. This is **not** as simple as it sounds (opening a file and assigning it the right file descriptor).

3 Implementation

3.1 Checkpointing a `pidfd`

3.1.1 Detecting a `pidfd`

`pidfd` are similiar to `signalfd`, `timerfd`, etc in the sense that they also use the `anonymous inode` infrastructure (But, this is changing?). We can use `readlink` to detect a `pidfd`.

```
// Declared in criu/include/files.h
static int dump_one_file(struct pid *pid, int fd, int lfd, struct fd_opts *opts,
        struct parasite_ctl *ctl, FdinfoEntry *e, struct parasite_drain_fd *dfds)
{
        /*...*/
        if (p.fs_type == ANON_INODE_FS_MAGIC) {
                char link[32];

                if (read_fd_link(lfd, link, sizeof(link)) < 0)
                        return -1;

                if (is_pidfd_link(link))
                        ops = &pidfd_dump_ops;
        }
        /*...*/
}

int is_pidfd_link(char *link)
{
        return is_anon_link_type(link, "[pidfd]");
}
```

3.1.2 Dumping a `pidfd`

CRIU has the `dump_one_file` method which is used dump a single file descriptor of a process. Every different type of `fd` has the following structure associated with it:

```
struct fdtype_ops {
        unsigned int type;
        int (*dump)(int lfd, u32 id, const struct fd_parms *p);
        int (*pre_dump)(int pid, int lfd);
};
```

A type must be defined for `pidfds`. The `dump` function pointer can point to function which does the following things:

- `Declare a struct defined using a protobuf message for pidfds`
- `Parse fdinfo and fill the struct with details`
- `Assign it the write ID`
- `Write that struct into a img file`

The dumping of a `timerfd` can be used as a reference implementation.

3.1.3 Defining image files for `pidfds`

To add a entry which stores data from `/proc/$pid/fdinfo/$number` we create a `images/pidfd.proto` file which defines the following message:

```
message pidfd_entry {
    optional uint32         id              = 1;
    optional uint64         pos             = 2;
    optional uint32         flags           = 3;
    optional uint32         mnt_id          = 4;
    optional uint64         inode           = 5;
    optional int64          pid             = 6;
    optional int64          nspid           = 7;
    optional fown_entry     fown            = 8;
}
```

This struct is also added in the `fd_types` enum and as an optional value for the `file_entry` message in `images/fdinfo.proto`. Now, if a `pidfd` is open a entry containing the above information will appear in `files.img`.

3.1.4 Parsing `/proc` for `pidfds`

Information related to file descriptors is present in `/proc/$pid/fdinfo/$number`. A `pidfd` has two unique entries namely, `Pid` and `NSpid`. If these entries are present we can say that a process has a open `pidfd`.

CRIU has a function `parse_fdinfo_fd_s(int pid, int fd, int type, void* arg)` in the file `criu/proc_parse.c` which makes `arg` argument point to a struct storing info for the specific `fd`. We can use the `fdinfo_field` macro to check for the fields `Pid` and `NSPid` if they are present we create a `PidfdEntry`.

3.2 Restoring a `pidfd`

3.2.1 Collect `pidfd` Data from Images

Each file descriptor type has the following struct associated with it.

```
struct collect_image_info {
        int fd_type;
        int pb_type;
        unsigned int priv_size;
        int (*collect)(void *, ProtobufCMessage *, struct cr_img *);
        unsigned flags;
};
```

The `collect` function pointer should point to a function that can collects information from the `ProtobufCMessage` pointer and transfers it to a struct pointed to by the `void *`. This pattern is common in CRIU and its implementation should be very straightforward.

3.2.2 `file_desc_ops`

```
struct file_desc_ops {
        /* fd_types from images/fdinfo.proto */
        unsigned int type;
        /*
         * Opens a file by whatever syscall is required for that.
         * The returned descriptor may be closed (dup2-ed to another)
         * so it shouldn't be saved for any post-actions.
         */
        int (*open)(struct file_desc *d, int *new_fd);
        char *(*name)(struct file_desc *, char *b, size_t s);
};
```

Each type of file descriptor has this struct associated with it.

3.2.3 Opening `pidfd(s)` for Process(es) Outside the Process Tree Being Restored

In this case, we can simply reopen the fd (using `pidfd_open`) during restore using the pid taken from `/proc/$pid/fdinfo/$number`.
But, there are two problems with this approach:

- The process (whose `pidfd` was open) might have exited between checkpoint and restore.
- The `pid` we have checkpointed might now refer a different process.

3.2.4 Opening `pidfd(s)` for a Process(es) Inside the Process Tree Being Restored

A pidfd can also refer to a process in the same process tree. A parent can open one for a child, a child can for the parent or a child can open one for a different child (A process might also open a `pidfd` that refers to itself). This presents a problem for CRIU. We will have to ensure that a process exists before trying to open a pidfd for it.

3.3 File Descriptors created using `pidfd_getfd`

Additional work might not be required to add support for this syscall. Since, `pidfd_getfd` creates a `duplicate` file descriptor, these can be **C/R** similiar to any other file descriptor.

3.4 Adding Tests

A proper implementation of `pidfds` should be able checkpoint/restore in a variety of use cases. Tests are necessary to prove this fact.
Tests should demonstrate the following facts:

- **C/R** of `pidfds` works in the simplest case.
- **C/R** of `pidfds` works in a more involved example consisting many open `pidfds` referring to different processes.
- **C/R** of `pidfds` works for processes resembling real world use cases.

4 Timeline

4.1 2 April - 1 May

- **Experiment** and better understand the checkpointing and restoring process of CRIU. (especially for files).
- **Design and Implement** a proof of concept for adding support for pidfds.

4.2 1 May - 26 May (Community Bonding Period)

- Discuss the minor aspects of the proposal with the mentors and recognize any problems that may arise during implementation.
- Based on feedback, modify the proposal and finalise the solution.

4.3 27 May - 12 July (Phase I)

- **27 May - 9 June:** Work on checkpointing `pidfds`
- **10 June - 23 June:** Work on implementing restore for process(es) outside `dumpee's` process tree
- **24 June - 30 June:** Added Tests and discuss solutions for verifying that the correct process has been restored
- **1 July - 7 July:** Implement some verification techniques
- **8 July - 12 July:** Midterm evaluation and potential change of plans for Phase II

4.4 12 July - August 26 (Phase II)

- **15 July - 21 July:** Begin work on implementing restore for `pidfd's` of processes in the current process tree.
- **22 July - 4 August:** Finish work on implementing restore for `pidfd's` of processes in the current process tree.
- **5 Aug - 11 August:** Write tests for `pidfds` and integrate them with the zdtm test suite
- **12 Aug - 18 August:** Write documentation, clean up the pull request.
- **19 Aug - 26 August:** A buffer week to accommodate changes and rescheduling.

4.5 After August 26

- `pidfds` are still a evolving feature of the Linux Kernel. I can continue to look after this implementation and extend its functionality.
- I would also like to improve documentation for CRIU.

5 Personal Information

- **Name:** Bhavik Sachdev
- **Email:** b.sachdev1904@gmail.com
- **Phone Number:** +91 8319336255
- **GitHub:** bsach64
- **LinkedIn:** Bhavik Sachdev
- **Location:** Raipur, India
- **Timezone:** GMT +0530

5.1 About Me

I am a second-year student at IIIT Naya Raipur, India, pursuing a degree in Computer Science. I have always been fascinated with everything computers, be it hardware or software. I keep up with the latest hardware releases from Intel, Nvidia, and AMD.

My interest in systems programming and CRIU started with a simple question: **How does a shell work?** This led me down a rabbit hole, and I tried to learn everything about Linux processes and the kernel. I even built a simple shell. When I wanted to apply for GSOC, CRIU was an obvious choice as it perfectly aligns with my interests. Over the past couple of months, as I have dug deeper into the CRIU codebase, I have learned so much through that process, and I would love to contribute to it.

5.2 Open Source Activity

I have opened a PR for solving a issue in the CRIU repository:

- zdtm: Distinguish between fail and crash of dump

I have contributed to the sympy repository (Relevant PRs):

- Added a function for opportunistic subscripts used in the expint and polylog functions
- Added a test for an edge case in the plot3d function
- Enhanced documentation for the new biomechanics sympy module

5.3 Commitments During GSOC 2024

I will have my end-semester examinations in the first and second week of May (During the Community Bonding Period). Apart from that, I am completely free this summer and fully dedicate my time to GSOC (40-50 hours a week). I will also make up for the lost time in May.

Should circumstances impede the project's progress, I will promptly inform my mentors and make up for it by increasing my workload. In the event of unforeseen obstacles, I am prepared to allocate additional time to the project in the subsequent weeks.

6 References

1. Checkpoint/Restore
2. Prajwal S N GSOC 2022 Proposal
3. A draft for implementing `pidfds`
4. Fdinfo Engine
5. How hard is it to open a file?
6. How to assign needed file descriptor to a file
7. man pages for `/proc`
8. Adding the pidfd abstraction to the kernel
9. Completing the pidfd API
10. pidfd implementation
11. CRIU Images
12. Tree After Restore
13. Dumping Files