

NAME

core – core dump file

DESCRIPTION

The default action of certain signals is to cause a process to terminate and produce a *core dump file*, a file containing an image of the process's memory at the time of termination. This image can be used in a debugger (e.g., **gdb**(1)) to inspect the state of the program at the time that it terminated. A list of the signals which cause a process to dump core can be found in **signal**(7).

A process can set its soft **RLIMIT_CORE** resource limit to place an upper limit on the size of the core dump file that will be produced if it receives a "core dump" signal; see **getrlimit**(2) for details.

There are various circumstances in which a core dump file is not produced:

- The process does not have permission to write the core file. (By default, the core file is called *core* or *core.pid*, where *pid* is the ID of the process that dumped core, and is created in the current working directory. See below for details on naming.) Writing the core file fails if the directory in which it is to be created is not writable, or if a file with the same name exists and is not writable or is not a regular file (e.g., it is a directory or a symbolic link).
- A (writable, regular) file with the same name as would be used for the core dump already exists, but there is more than one hard link to that file.
- The filesystem where the core dump file would be created is full; or has run out of inodes; or is mounted read-only; or the user has reached their quota for the filesystem.
- The directory in which the core dump file is to be created does not exist.
- The **RLIMIT_CORE** (core file size) or **RLIMIT_FSIZE** (file size) resource limits for the process are set to zero; see **getrlimit**(2) and the documentation of the shell's *ulimit* command (*limit* in **cs**(1)). However, **RLIMIT_CORE** will be ignored if the system is configured to pipe core dumps to a program.
- The binary being executed by the process does not have read permission enabled. (This is a security measure to ensure that an executable whose contents are not readable does not produce a—possibly readable—core dump containing an image of the executable.)
- The process is executing a set-user-ID (set-group-ID) program that is owned by a user (group) other than the real user (group) ID of the process, or the process is executing a program that has file capabilities (see **capabilities**(7)). (However, see the description of the **prctl**(2) **PR_SET_DUMPABLE** operation, and the description of the */proc/sys/fs/suid_dumpable* file in **proc**(5).)
- */proc/sys/kernel/core_pattern* is empty and */proc/sys/kernel/core_uses_pid* contains the value 0. (These files are described below.) Note that if */proc/sys/kernel/core_pattern* is empty and */proc/sys/kernel/core_uses_pid* contains the value 1, core dump files will have names of the form *.pid*, and such files are hidden unless one uses the **ls**(1) *-a* option.
- (Since Linux 3.7) The kernel was configured without the **CONFIG_COREDUMP** option.

In addition, a core dump may exclude part of the address space of the process if the **madvise**(2) **MADV_DONTDUMP** flag was employed.

On systems that employ **systemd**(1) as the *init* framework, core dumps may instead be placed in a location determined by **systemd**(1). See below for further details.

Naming of core dump files

By default, a core dump file is named *core*, but the */proc/sys/kernel/core_pattern* file (since Linux 2.6 and 2.4.21) can be set to define a template that is used to name core dump files. The template can contain % specifiers which are substituted by the following values when a core file is created:

%%

A single % character.

- %c Core file size soft resource limit of crashing process (since Linux 2.6.24).
- %d Dump mode—same as value returned by **prctl(2)** **PR_GET_DUMPABLE** (since Linux 3.7).
- %e The process or thread's *comm* value, which typically is the same as the executable filename (without path prefix, and truncated to a maximum of 15 characters), but may have been modified to be something different; see the discussion of */proc/pid/comm* and */proc/pid/task/tid/comm* in **proc(5)**.
- %E Pathname of executable, with slashes (/) replaced by exclamation marks (!) (since Linux 3.0).
- %F PIDFD of dumped process (since Linux 6.16).
- %g Numeric real GID of dumped process.
- %h Hostname (same as *nodename* returned by **uname(2)**).
- %i TID of thread that triggered core dump, as seen in the PID namespace in which the thread resides (since Linux 3.18).
- %I TID of thread that triggered core dump, as seen in the initial PID namespace (since Linux 3.18).
- %p PID of dumped process, as seen in the PID namespace in which the process resides.
- %P PID of dumped process, as seen in the initial PID namespace (since Linux 3.12).
- %s Number of signal causing dump.
- %t Time of dump, expressed as seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).
- %u Numeric real UID of dumped process.

A single % at the end of the template is dropped from the core filename, as is the combination of a % followed by any character other than those listed above. All other characters in the template become a literal part of the core filename. The template may include ' characters, which are interpreted as delimiters for directory names. The maximum size of the resulting core filename is 128 bytes (64 bytes before Linux 2.6.19). The default value in this file is "core". For backward compatibility, if */proc/sys/kernel/core_pattern* does not include %p and */proc/sys/kernel/core_uses_pid* (see below) is nonzero, then .PID will be appended to the core filename.

Paths are interpreted according to the settings that are active for the crashing process. That means the crashing process's mount namespace (see **mount_namespaces(7)**), its current working directory (found via **getcwd(2)**), and its root directory (see **chroot(2)**).

Since Linux 2.4, Linux has also provided a more primitive method of controlling the name of the core dump file. If the */proc/sys/kernel/core_uses_pid* file contains the value 0, then a core dump file is simply named *core*. If this file contains a nonzero value, then the core dump file includes the process ID in a name of the form *core.PID*.

Since Linux 3.6, if */proc/sys/fs/suid_dumpable* is set to 2 ("suidsafes"), the pattern must be either an absolute pathname (starting with a leading / character) or a pipe, as defined below.

Piping core dumps to a program

Since Linux 2.6.19, Linux supports an alternate syntax for the */proc/sys/kernel/core_pattern* file. If the first character of this file is a pipe symbol (|), then the remainder of the line is interpreted as the command-line for a user-space program (or script) that is to be executed.

Since Linux 5.3.0, the pipe template is split on spaces into an argument list *before* the template parameters are expanded. In earlier kernels, the template parameters are expanded first and the resulting string is split on spaces into an argument list. This means that in earlier kernels executable names added by the %e and %E template parameters could get split into multiple arguments. So the core dump handler needs to put the executable names as the last argument and ensure it joins all parts of the executable name using spaces. Executable names with multiple spaces in them are not correctly represented in earlier kernels, meaning that the core dump handler needs to use mechanisms to find the executable name.

Instead of being written to a file, the core dump is given as standard input to the program. Note the following points:

- The program must be specified using an absolute pathname (or a pathname relative to the root directory, /), and must immediately follow the '|' character.

- The command-line arguments can include any of the % specifiers listed above. For example, to pass the PID of the process that is being dumped, specify %p in an argument.
- The process created to run the program runs as user and group *root*.
- Running as *root* does not confer any exceptional security bypasses. Namely, LSMs (e.g., SELinux) are still active and may prevent the handler from accessing details about the crashed process via */proc/pid*.
- The program pathname is interpreted with respect to the initial mount namespace as it is always executed there. It is not affected by the settings (e.g., root directory, mount namespace, current working directory) of the crashing process.
- The process runs in the initial namespaces (PID, mount, user, and so on) and not in the namespaces of the crashing process. One can utilize specifiers such as %P to find the right */proc/pid* directory and probe/enter the crashing process's namespaces if needed.
- The process starts with its current working directory as the root directory. If desired, it is possible change to the working directory of the dumping process by employing the value provided by the %P specifier to change to the location of the dumping process via */proc/pid/cwd*.
- Command-line arguments can be supplied to the program (since Linux 2.6.24), delimited by white space (up to a total line length of 128 bytes).
- The **RLIMIT_CORE** limit is not enforced for core dumps that are piped to a program via this mechanism.

/proc/sys/kernel/core_pipe_limit

When collecting core dumps via a pipe to a user-space program, it can be useful for the collecting program to gather data about the crashing process from that process's */proc/pid* directory. In order to do this safely, the kernel must wait for the program collecting the core dump to exit, so as not to remove the crashing process's */proc/pid* files prematurely. This in turn creates the possibility that a misbehaving collecting program can block the reaping of a crashed process by simply never exiting.

Since Linux 2.6.32, the */proc/sys/kernel/core_pipe_limit* can be used to defend against this possibility. The value in this file defines how many concurrent crashing processes may be piped to user-space programs in parallel. If this value is exceeded, then those crashing processes above this value are noted in the kernel log and their core dumps are skipped.

A value of 0 in this file is special. It indicates that unlimited processes may be captured in parallel, but that no waiting will take place (i.e., the collecting program is not guaranteed access to */proc/<crashing-PID>*). The default value for this file is 0.

Controlling which mappings are written to the core dump

Since Linux 2.6.23, the Linux-specific */proc/pid/coredump_filter* file can be used to control which memory segments are written to the core dump file in the event that a core dump is performed for the process with the corresponding process ID.

The value in the file is a bit mask of memory mapping types (see **mmap(2)**). If a bit is set in the mask, then memory mappings of the corresponding type are dumped; otherwise, they are not dumped. The bits in this file have the following meanings:

- bit 0 Dump anonymous private mappings.
- bit 1 Dump anonymous shared mappings.
- bit 2 Dump file-backed private mappings.
- bit 3 Dump file-backed shared mappings.
- bit 4 (since Linux 2.6.24)
 Dump ELF headers.
- bit 5 (since Linux 2.6.28)
 Dump private huge pages.
- bit 6 (since Linux 2.6.28)
 Dump shared huge pages.

- bit 7 (since Linux 4.4)
Dump private DAX pages.
- bit 8 (since Linux 4.4)
Dump shared DAX pages.

By default, the following bits are set: 0, 1, 4 (if the **CONFIG_CORE_DUMP_DEFAULT_ELF_HEADERS** kernel configuration option is enabled), and 5. This default can be modified at boot time using the *coredump_filter* boot option.

The value of this file is displayed in hexadecimal. (The default value is thus displayed as 33.)

Memory-mapped I/O pages such as frame buffer are never dumped, and virtual DSO (**vdso(7)**) pages are always dumped, regardless of the *coredump_filter* value.

A child process created via **fork(2)** inherits its parent's *coredump_filter* value; the *coredump_filter* value is preserved across an **execve(2)**.

It can be useful to set *coredump_filter* in the parent shell before running a program, for example:

```
$ echo 0x7 > /proc/self/coredump_filter
$ ./some_program
```

This file is provided only if the kernel was built with the **CONFIG_ELF_CORE** configuration option.

Core dumps and systemd

On systems using the **systemd(1)** *init* framework, core dumps may be placed in a location determined by **systemd(1)**. To do this, **systemd(1)** employs the *core_pattern* feature that allows piping core dumps to a program. One can verify this by checking whether core dumps are being piped to the **systemd-coredump(8)** program:

```
$ cat /proc/sys/kernel/core_pattern;
|/usr/lib/systemd/systemd-coredump %P %u %g %s %t %c %e
```

In this case, core dumps will be placed in the location configured for **systemd-coredump(8)**, typically as **lz4(1)** compressed files in the directory */var/lib/systemd/coredump/*. One can list the core dumps that have been recorded by **systemd-coredump(8)** using **coredumpctl(1)**:

```
$ coredumpctl list | tail -5;
Wed 2017-10-11 22:25:30 CEST 2748 1000 1000 3 present /usr/bin/sleep
Thu 2017-10-12 06:29:10 CEST 2716 1000 1000 3 present /usr/bin/sleep
Thu 2017-10-12 06:30:50 CEST 2767 1000 1000 3 present /usr/bin/sleep
Thu 2017-10-12 06:37:40 CEST 2918 1000 1000 3 present /usr/bin/cat
Thu 2017-10-12 08:13:07 CEST 2955 1000 1000 3 present /usr/bin/cat
```

The information shown for each core dump includes the date and time of the dump, the PID, UID, and GID of the dumping process, the signal number that caused the core dump, and the pathname of the executable that was being run by the dumped process. Various options to **coredumpctl(1)** allow a specified coredump file to be pulled from the **systemd(1)** location into a specified file. For example, to extract the core dump for PID 2955 shown above to a file named *core* in the current directory, one could use:

```
$ coredumpctl dump 2955 -o core;
```

For more extensive details, see the **coredumpctl(1)** manual page.

To (persistently) disable the **systemd(1)** mechanism that archives core dumps, restoring to something more like traditional Linux behavior, one can set an override for the **systemd(1)** mechanism, using something like:

```
# echo "kernel.core_pattern=core.%p" > \;
    /etc/sysctl.d/50-coredump.conf
# /lib/systemd/systemd-sysctl;
```

It is also possible to temporarily (i.e., until the next reboot) change the *core_pattern* setting using a command such as the following (which causes the names of core dump files to include the executable name as well as the number of the signal which triggered the core dump):

```
# sysctl -w kernel.core_pattern="%e-%s.core";
```

NOTES

The **gdb**(1) *gcore* command can be used to obtain a core dump of a running process.

In Linux versions up to and including 2.6.27, if a multithreaded process (or, more precisely, a process that shares its memory with another process by being created with the **CLONE_VM** flag of **clone**(2)) dumps core, then the process ID is always appended to the core filename, unless the process ID was already included elsewhere in the filename via a *%p* specification in */proc/sys/kernel/core_pattern*. (This is primarily useful when employing the obsolete LinuxThreads implementation, where each thread of a process has a different PID.)

EXAMPLES

The program below can be used to demonstrate the use of the pipe syntax in the */proc/sys/kernel/core_pattern* file. The following shell session demonstrates the use of this program (compiled to create an executable named *core_pattern_pipe_test*):

```
$ cc -o core_pattern_pipe_test core_pattern_pipe_test.c
$ su
Password:
# echo "|$PWD/core_pattern_pipe_test %p UID=%u GID=%g sig=%s" > \
  /proc/sys/kernel/core_pattern
# exit
$ sleep 100
^\  
Quit (core dumped)          # type control-backslash
$ cat core.info
argc=5
argc[0]=</home/mtk/core_pattern_pipe_test>
argc[1]=<20575>
argc[2]=<UID=1000>
argc[3]=<GID=100>
argc[4]=<sig=3>
Total bytes in core dump: 282624
```

Program source

```
/* core_pattern_pipe_test.c */

#define _GNU_SOURCE
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUF_SIZE 1024

int
main(int argc, char *argv[])
{
    ssize_t nread, tot;
    char buf[BUF_SIZE];
    FILE *fp;
    char cwd[PATH_MAX];
```

```
/* Change our current working directory to that of the
   crashing process.  */

snprintf(cwd, PATH_MAX, "/proc/%s/cwd", argv[1]);
chdir(cwd);

/* Write output to file "core.info" in that directory.  */

fp = fopen("core.info", "w+");
if (fp == NULL)
    exit(EXIT_FAILURE);

/* Display command-line arguments given to core_pattern
   pipe program.  */

fprintf(fp, "argc=%d\n", argc);
for (size_t j = 0; j < argc; j++)
    fprintf(fp, "argc[%zu]=<%s>\n", j, argv[j]);

/* Count bytes in standard input (the core dump).  */

tot = 0;
while ((nread = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
    tot += nread;
fprintf(fp, "Total bytes in core dump: %zd\n", tot);

fclose(fp);
exit(EXIT_SUCCESS);
}
```

SEE ALSO

bash(1), **coredumpctl(1)**, **gdb(1)**, **getrlimit(2)**, **mmap(2)**, **prctl(2)**, **sigaction(2)**, **elf(5)**, **proc(5)**, **pthread(7)**, **signal(7)**, **systemd-coredump(8)**