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 disk 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 will fail if the directory in which it is to be created is nonwritable, 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 **csh**(1)).
* The binary being executed by the process does not have read permission enabled.
* 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. (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).)
* (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.

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	executable filename (without path prefix)
%E	pathname of executable, with slashes ('/') replaced by exclamation marks ('!') (since Linux 3.0).
%g	(numeric) real GID of dumped process
%h	hostname (same as *nodename* returned by **uname**(2))
%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 in kernels before 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.

Since version 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 ("suidsafe"), 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 kernel 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 a program to be executed. Instead of being written to a disk 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 process created to run the program runs as user and group *root*.
* 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 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.

Controlling which mappings are written to the core dump

Since kernel 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.

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. 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 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.

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.)

EXAMPLE

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
^\                    # type control-backslash
Quit (core dumped)
$ 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[])
{
    int tot, j;
    ssize_t nread;
    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 (j = 0; j < argc; j++)
        fprintf(fp, "argc[%d]=<%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: %d\n", tot);

    exit(EXIT_SUCCESS);
}
```

SEE ALSO

bash(1), **gdb**(1), **getrlimit**(2), **mmap**(2), **prctl**(2), **sigaction**(2), **elf**(5), **proc**(5), **pthreads**(7), **signal**(7)