BAHIR DAR UNIVERSITY

BAHIR DAR INSTITUE OF TECHNOLOGY

DEPARTMENT OF SOFTWARE ENGINEERING

Operating system And System Programming

Individual assignment

Name MILETE TSEGAZEAB

ID BDU 1308582

Submitted to lecturer: WENDIMU BAYE

Submitted date:18/11/2014EC

ptrace(2) - Linux

Name

ptrace - process trace

Synopsis

```
#include<sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid,
void *addr, void *data);
```

1) what ,why,how it works ?

what it is(description about it)?

The **ptrace**() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

How it works?

A tracee first needs to be attached to the tracer. Attachment and subsequent commands are per thread: in a multithreaded process, every thread can be individually attached to a (potentially different) tracer, or left not attached and thus not debugged. Therefore, "tracee" always means "(one) thread", never "a (possibly multithreaded) process". Ptrace commands

are always sent to a specific tracee using a call of the form

ptrace(PTRACE_foo, pid, ...)

where *pid* is the thread ID of the corresponding Linux thread.

(Note that in this page, a "multithreaded process" means a thread group consisting of threads created using the *clone*(2) **CLONE_THREAD** flag.)

A process can initiate a trace by calling *fork(2)* and having the resulting child do a PTRACE_TRACEME, followed (typically) by an *execve(2)*. Alternatively, one process may commence tracing another process
using PTRACE_ATTACH or PTRACE_SEIZE.

While being traced, the tracee will stop each time a signal is delivered, even if the signal is being ignored. (An exception is **SIGKILL**, which has its usual effect.) The tracer will be notified at its next call to ***waitpid****(2)* (or one of the related "wait" system calls); that call will return a *status* value containing information that indicates the cause of the stop in the tracee. While the tracee is stopped, the tracer can use various ptrace requests to inspect and modify the tracee. The tracer then causes the tracee to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

If the PTRACE_O_TRACEEXEC option is not in effect, all successful calls to ***execve****(2)* by the traced process will cause it to be sent a SIGTRAP signal, giving the parent a chance to gain control before the new program begins execution.

When the tracer is finished tracing, it can cause the tracee to continue executing in a normal, untraced mode via PTRACE_DETACH.

The value of *request* determines the action to be performed:

PTRACE_TRACEME

Indicate that this process is to be traced by its parent. A process probably shouldn't make this request if its parent isn't expecting to trace it. (*pid*, *addr*, and *data* are ignored.)

The PTRACE_TRACEME request is used only by the tracee; the remaining requests are used only by the tracer. In the following requests, *pid* specifies the thread ID of the tracee to be acted on. For requests other than PTRACE_ATTACH, PTRACE_SEIZE, PTRACE_INTERRUPT and PTRACE_KILL, the tracee must be stopped.

PTRACE_PEEKTEXT, PTRACE_PEEKDATA

Read a word at the address *addr* in the tracee's memory, returning the word as the result of the **ptrace**() call. Linux does not have separate text and data address spaces, so these two requests are currently equivalent. (*data* is ignored.)

PTRACE_PEEKUSER

Read a word at offset *addr* in the tracee's USER area, which holds the registers and other information about the process (see <*sys/user.h*>). The word is returned as the result of the **ptrace**() call. Typically, the offset must be word-aligned, though this might vary by architecture. See NOTES. (*data* is ignored.)

PTRACE_POKETEXT, PTRACE_POKEDATA

Copy the word *data* to the address *addr* in the tracee's memory. As for PTRACE_PEEKTEXT and PTRACE_PEEKDATA, these two requests are currently equivalent.

PTRACE_POKEUSER

Copy the word *data* to offset *addr* in the tracee's USER area. As for PTRACE_PEEKUSER, the offset must typically be word-aligned. In order to maintain the integrity of the kernel, some modifications to the USER area are disallowed.

PTRACE_GETREGS, PTRACE_GETFPREGS

Copy the tracee's general-purpose or floating-point registers, respectively, to the address *data* in the tracer. See <*sys/user.h*> for information on the format of this data. (*addr* is ignored.) Note that SPARC systems have the meaning of *data* and *addr* reversed; that is, *data* is ignored and the registers are copied to the address *addr*. PTRACE_GETREGS and PTRACE_GETFPREGS are not present on all architectures.

PTRACE_GETREGSET

Read the tracee's registers. *addr* specifies, in an architecture-dependent way, the type of registers to be read. NT_PRSTATUS (with numerical value 1) usually results in reading of general-purpose registers. If the CPU has, for example, floating-point and/or vector registers, they can be retrieved by setting *addr* to the corresponding NT_foo constant. *data* points to a struct iovec, which describes the destination buffer's location and length. On return, the kernel modifies iov.len to indicate the actual number of bytes returned.

PTRACE_GETSIGINFO

Retrieve information about the signal that caused the stop. Copy a *siginfo_t* structure from the tracee to the address *data* in the tracer. (*addr* is ignored.)

PTRACE_SETREGS, PTRACE_SETFPREGS

Modify the tracee's general-purpose or floating-point registers, respectively, from the address *data* in the tracer. As for PTRACE_POKEUSER, some general-purpose register modifications may be disallowed. (*addr* is ignored.) Note that SPARC systems have the meaning of *data* and *addr* reversed; that is, *data* is ignored and the registers are copied from the address *addr*. PTRACE_SETREGS and PTRACE_SETFPREGS are not present on all architectures.

PTRACE_SETREGSET

Modify the tracee's registers. The meaning of *addr* and *data* is analogous to PTRACE_GETREGSET.

PTRACE_SETSIGINFO

Set signal information: copy a *siginfo_t* structure from the address *data* in the tracer to the tracee. This will affect only signals that would normally be delivered to the tracee and were caught by the tracer. It may be difficult to tell these normal signals from synthetic signals generated by **ptrace**() itself. (*addr* is ignored.)

PTRACE_SETOPTIONS

Set ptrace options from *data*. (*addr* is ignored.) *data* is interpreted as

a bit mask of options, which are specified by the following flags:

PTRACE_GETEVENTMSG

Retrieve a message about the ptrace event that just happened, placing it at the address *data* in the tracer. For PTRACE_EVENT_EXIT, this is the tracee's exit status.

For PTRACE_EVENT_FORK, PTRACE_EVENT_VFORK, PTRACE_EVENT_VFORK_DONE, and PTRACE_EVENT_CLONE, this is the PID of the new process. (*addr* is ignored.)

PTRACE_CONT

Restart the stopped tracee process. If *data* is nonzero, it is interpreted as the number of a signal to be delivered to the tracee; otherwise, no signal is delivered. Thus, for example, the tracer can control whether a signal sent to the tracee is delivered or not. (*addr* is ignored.)

PTRACE_SYSCALL, PTRACE_SINGLESTEP

Restart the stopped tracee as for PTRACE_CONT, but arrange for the tracee to be stopped at the next entry to or exit from a system call, or after execution of a single instruction, respectively. (The tracee will also, as usual, be stopped upon receipt of a signal.) From the tracer's perspective, the tracee will appear to have been stopped by receipt of a SIGTRAP. So, for PTRACE_SYSCALL, for example, the idea is to inspect the arguments to the system call at the first stop, then do another PTRACE_SYSCALL and inspect the return value of the system call at the second stop. The *data* argument is treated as for PTRACE_CONT. (*addr* is ignored.)

PTRACE_SYSEMU, PTRACE_SYSEMU_SINGLESTEP (since Linux 2.6.14)

For PTRACE_SYSEMU, continue and stop on entry to the next system call, which will not be executed. For PTRACE_SYSEMU_SINGLESTEP, do the same but also singlestep if not a system call. This call is used by programs like User Mode Linux that want to emulate all the tracee's system calls. The *data* argument is treated as for PTRACE_CONT. The *addr* argument is ignored. These requests are currently supported only on x86.

PTRACE_LISTEN

Restart the stopped tracee, but prevent it from executing. The resulting state of the tracee is similar to a process which has been stopped by a SIGSTOP (or other stopping signal). See the "group-stop" subsection for additional information. PTRACE_LISTEN only works on tracees attached by PTRACE_SEIZE.

PTRACE_KILL

Send the tracee a SIGKI**LL** to terminate it. (*addr* and *data* are ignored.)

This operation is deprecated; do not use it!

Instead, send a SIGKILL directly using ***kill***(*2*) or ***tgkill***(*2*). The problem with PTRACE_KILL is that it requires the tracee to be in signal-delivery-stop, otherwise it may not work (i.e., may complete successfully but won't kill the tracee). By contrast, sending a SIGKILL directly has no such limitation.

PTRACE_INTERRUPT

Stop a tracee. If the tracee is running, it will stop with PTRACE_EVENT_STOP. If the tracee is already stopped by a signal, or receives a signal in parallel with PTRACE_INTERRUPT, it may report a group-stop or a signal-delivery-stop instead of PTRACE_EVENT_STOP. PTRACE_INTERRUPT only works on tracees attached by PTRACE_SEIZE.

PTRACE_ATTACH

Attach to the process specified in *pid*, making it a tracee of the calling process. The tracee is sent a **SIGSTOP**, but will not necessarily have stopped by the completion of this call; use ***waitpid***(*2*) to wait for the tracee to stop. See the "Attaching and detaching" subsection for additional information. (*addr* and *data* are ignored.)

PTRACE_SEIZE (since Linux 3.4)

Attach to the process specified in *pid*, making it a tracee of the calling process. Unlike **PTRACE_ATTACH**, **PTRACE_SEIZE** does not stop the process. Only a **PTRACE_SEIZE**d process can accept **PTRACE_INTERRUPT** and **PTRACE_LISTEN** commands. *addr* must be zero. *data* contains a bit mask of ptrace options to activate immediately.

PTRACE_DETACH

Restart the stopped tracee as for **PTRACE_CONT**, but first detach from it. Under Linux, a tracee can be detached in this way regardless of which method was used to initiate tracing. (*addr* is ignored.)

Death under ptrace

When a (possibly multithreaded) process receives a killing signal (one whose disposition is set to SIG_DFL and whose default action is to kill the process), all threads exit. Tracees report their death to their tracer(s). Notification of this event is delivered via *waitpid(2)*.

Note that the killing signal will first cause signal-delivery-stop (on one tracee only), and only after it is injected by the tracer (or after it was dispatched to a thread which isn't traced), will death from the signal happen on *all* tracees within a multithreaded process. (The term "signal-delivery-stop" is explained below.)

SIGKILL does not generate signal-delivery-stop and therefore the tracer can't suppress it. SIGKILL kills even within system calls (syscall-exit-stop is not generated prior to death by SIGKILL). The net effect is that SIGKILL always kills the process (all its threads), even if some threads of the process are ptraced.

When the tracee calls **_exit**(2), it reports its death to its tracer. Other threads are not affected.

When any thread executes ***exit_group****(2)*, every tracee in its thread group reports its death to its tracer.

If the PTRACE_O_TRACEEXIT option is on, PTRACE_EVENT_EXIT will happen before actual death. This applies to exits via *exit(2)*, ***exit_group(2),*** and signal deaths (except SIGKILL), and when threads are torn down on ***execve****(2)* in a multithreaded process.

The tracer cannot assume that the ptrace-stopped tracee exists. There are many scenarios when the tracee may die while stopped (such as SIGKILL). Therefore, the tracer must be prepared to handle an ESRCH error on any ptrace operation. Unfortunately, the same error is returned if the tracee exists but is not ptrace-stopped (for commands which require a stopped tracee), or if it is not traced by the process which issued the ptrace call. The tracer needs to keep track of the stopped/running state of the tracee, and interpret ESRCH as "tracee died unexpectedly" only if it knows that the tracee has been observed to enter ptrace-stop. Note that there is no guarantee that *waitpid(WNOHANG)* will reliably report the tracee's death status if a ptrace operation returned ESRCH. *waitpid(WNOHANG)* may return 0 instead. In other words, the tracee may be "not yet fully dead", but already refusing ptrace requests.

The tracer can't assume that the tracee *always* ends its life by reporting *WIFEXITED(status)* or *WIFSIGNALED(status)*; there are cases where this does not occur. For example, if a thread other than thread group leader does an ***execve****(2)*, it disappears; its PID will never be seen again, and any subsequent ptrace stops will be reported under the thread group leader's PID.

Stopped states

A tracee can be in two states: running or stopped.

There are many kinds of states when the tracee is stopped, and in ptrace discussions they are often conflated. Therefore, it is important to use precise terms.

In this manual page, any stopped state in which the tracee is ready to accept ptrace commands from the tracer is called *ptrace-stop*. Ptrace-stops can be further subdivided into *signal-delivery-stop*, *group-stop*, *syscall-stop*, and so on. These stopped states are described in detail below.

When the running tracee enters ptrace-stop, it notifies its tracer using ***waitpid***(*2*) (or one of the other "wait" system calls). Most of this manual page assumes that the tracer waits with:

pid = waitpid(pid_or_minus_1, &status, __WALL);

Ptrace-stopped tracees are reported as returns with *pid* greater than 0 and *WIFSTOPPED(status)* true.

The **__**WALL flag does not include the WSTOPPED and WEXITED flags, but implies their functionality.

Setting the WCONTINUED flag when calling *waitpid*(*2*) is not recommended: the "continued" state is per-process and consuming it can confuse the real parent of the tracee.

Use of the WNOHANG flag may cause ***waitpid***(*2*) to return 0 ("no wait results available yet") even if the tracer knows there should be a notification. Example:

```
errno = 0;
ptrace(PTRACE_CONT, pid, 0L, 0L);
if (errno == ESRCH) {
    /* tracee is dead */
    r = waitpid(tracee, &status, __WALL | WNOHANG);
    /* r can still be 0 here! */
}
```

The following kinds of ptrace-stops exist: signal-delivery-stops, group-stops, PTRACE_EVENT stops, syscall-stops. They all are reported by ***waitpid***(*2*) with *WIFSTOPPED(status)* true. They may be differentiated by examining the value *status>>8*, and if there is ambiguity in that value, by querying PTRACE_GETSIGINFO.

Signal-delivery-stop

When a (possibly multithreaded) process receives any signal except SIGKILL, the kernel selects an arbitrary thread which handles the signal. (If the signal is generated with ***tgkill****(2)*, the target thread can be explicitly selected by the caller.) If the selected thread is traced, it enters signal-delivery-stop. At this point, the signal is not yet delivered to the process, and can be suppressed by the tracer. If the tracer doesn't suppress the signal, it passes the signal to the tracee in the next ptrace restart request. This second step of signal delivery is called *signal injection* in this manual page. Note that if the signal is blocked, signal-delivery-stop doesn't happen until the signal is unblocked, with the usual exception that SIGSTOP can't be blocked.

Signal-delivery-stop is observed by the tracer as ***waitpid****(2)* returning with *WIFSTOPPED(status)* true, with the signal returned by *WSTOPSIG(status)*. If the signal is SIGTRAP, this may be a different kind of ptrace-stop; see the "Syscall-stops" and "execve" sections below for details. If *WSTOPSIG(status)* returns a stopping signal, this may be a group-stop; see below.

Signal injection and suppression

After signal-delivery-stop is observed by the tracer, the tracer should restart the tracee with the call

ptrace(PTRACE_restart, pid, 0, sig)

where PTRACE_restart is one of the restarting ptrace requests. If *sig* is 0, then a signal is not delivered. Otherwise, the signal *sig* is delivered. This operation is called *signal injection* in this manual page, to distinguish it from signal-delivery-stop.

The *sig* value may be different from the *WSTOPSIG(status)* value: the tracer can cause a different signal to be injected.

Note that a suppressed signal still causes system calls to return prematurely. In this case system calls will be restarted: the tracer will observe the tracee to reexecute the interrupted system call (or ***restart_syscall****(2)* system call for a few syscalls which use a different mechanism for restarting) if the tracer uses PTRACE_SYSCALL. Even system calls (such as ***poll****(2)*) which are not restartable after signal are restarted after signal is suppressed; however, kernel bugs exist which cause some syscalls to fail with EINTR even though no observable signal is injected to the tracee.

Restarting ptrace commands issued in ptrace-stops other than signal-delivery-stop are not guaranteed to inject a signal, even if *sig* is nonzero. No error is reported; a nonzero *sig* may simply be ignored. Ptrace users should not try to "create a new signal" this way: use *tgkill(2)* instead.

The fact that signal injection requests may be ignored when restarting the tracee after ptrace stops that are not signal-delivery-stops is a cause of confusion among ptrace users. One typical scenario is that the tracer observes group-stop, mistakes it for signal-delivery-stop, restarts the tracee with

ptrace(PTRACE_restart, pid, 0, stopsig)

with the intention of injecting *stopsig*, but *stopsig* gets ignored and the tracee continues to run.

The SIGCONT signal has a side effect of waking up (all threads of) a group-stopped process. This side effect happens before signal-delivery-stop. The tracer can't suppress this side effect (it can only suppress signal injection, which only causes the SIGCONT handler to not be executed in the tracee, if such a handler is installed). In fact, waking up from group-stop may be followed by signal-delivery-stop for signal(s) *other* than SIGCONT, if they were pending when SIGCONT was delivered. In other words, SIGCONT may be not the first signal observed by the tracee after it was sent.

Stopping signals cause (all threads of) a process to enter group-stop. This side effect happens after signal injection, and therefore can be suppressed by the tracer.

In Linux 2.4 and earlier, the SIGSTOP signal can't be injected.

PTRACE_GETSIGINFO can be used to retrieve a *siginfo_t* structure which corresponds to the delivered signal. PTRACE_SETSIGINFO may be used to modify it. If PTRACE_SETSIGINFO has been used to alter *siginfo_t*, the *si_signo* field and the *sig* parameter in the restarting command must match, otherwise the result is undefined.

Group-stop

When a (possibly multithreaded) process receives a stopping signal, all threads stop. If some threads are traced, they enter a group-stop. Note that the stopping signal will first cause signal-delivery-stop (on one tracee only), and only after it is injected by the tracer (or after it was dispatched to a thread which isn't traced), will group-stop be initiated on *all* tracees within the multithreaded process. As usual, every tracee reports its group-stop separately to the corresponding tracer.

Group-stop is observed by the tracer as *waitpid(2)* returning with *WIFSTOPPED(status)* true, with the stopping signal available via *WSTOPSIG(status)*. The same result is returned by some other classes of ptrace-stops, therefore the recommended practice is to perform the call

ptrace(PTRACE_GETSIGINFO, pid, 0, &siginfo)

The call can be avoided if the signal is not SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU; only these four signals are stopping signals. If the tracer sees something else, it can't be a group-stop. Otherwise, the tracer needs to call PTRACE_GETSIGINFO. If PTRACE_GETSIGINFO fails with EINVAL, then it is definitely a group-stop. (Other failure codes are possible, such as ESRCH ("no such process") if a SIGKILL killed the tracee.)

As of Linux 2.6.38, after the tracer sees the tracee ptrace-stop and until it restarts or kills it, the tracee will not run, and will not send notifications (except SIGKILL death) to the tracer, even if the tracer enters into another ***waitpid***(*2*) call.

The kernel behavior described in the previous paragraph causes a problem with transparent handling of stopping signals. If the tracer restarts the tracee after group-stop, the stopping signal is effectively ignored--the tracee doesn't remain stopped, it runs. If the tracer doesn't restart the tracee before entering into the next ***waitpid***(*2*), future SIGCONT signals will not be reported to the tracer; this would cause the SIGCONT signals to have no effect on the tracee.

Since Linux 3.4, there is a method to overcome this problem: instead of PTRACE_CONT, a PTRACE_LISTEN command can be used to restart a tracee in a way where it does not execute, but waits for a new event which it can report via ***waitpid***(*2*) (such as when it is restarted by a SIGCONT).

PTRACE_EVENT stops

If the tracer sets PTRACE_O_TRACE_* options, the tracee will enter ptrace-stops called PTRACE_EVENT stops.

PTRACE_EVENT stops are observed by the tracer as ***waitpid***(2) returning with *WIFSTOPPED(status)*, and *WSTOPSIG(status)* returns SIGTRAP. An additional bit is set in the higher byte of the status word: the value *status>>8* will be

(SIGTRAP | PTRACE_EVENT_foo << 8).

The following events exist:

PTRACE_EVENT_VFORK
: Stop before return from ***vfork***(2) or ***clone***(2) with the **CLONE_VFORK** flag. When the tracee is continued after this stop, it will wait for child to exit/exec before continuing its execution (in other words, the usual behavior on ***vfork***(2)).

PTRACE_EVENT_FORK
: Stop before return from ***fork***(2) or ***clone***(2) with the exit signal set to **SIGCHLD**.

PTRACE_EVENT_CLONE
: Stop before return from ***clone***(2).

PTRACE_EVENT_VFORK_DONE
: Stop before return from ***vfork***(2) or ***clone***(2) with the CLONE_VFORK flag, but after the child unblocked this tracee by exiting or execing.

For all four stops described above, the stop occurs in the parent (i.e., the tracee), not in the newly created thread. PTRACE_GETEVENTMSG can be used to retrieve the new thread's ID.

PTRACE_EVENT_EXEC
: Stop before return from ***execve***(2). Since Linux 3.0, PTRACE_GETEVENTMSG returns the former thread ID.

PTRACE_EVENT_EXIT
: Stop before exit (including death from ***exit_group***(2)), signal death, or exit caused by ***execve***(2) in a multithreaded process. PTRACE_GETEVENTMSG returns the exit status. Registers can be examined (unlike when "real" exit happens). The tracee is still alive; it needs to be PTRACE_CONTed or PTRACE_DETACHed to finish exiting.

PTRACE_EVENT_STOP
: Stop induced by PTRACE_INTERRUPT command.

PTRACE_GETSIGINFO on PTRACE_EVENT stops returns **SIGTRAP** in *si_signo*, with *si_code* set to *(event<<8)* | *SIGTRAP*.

Syscall-stops

If the tracee was restarted by PTRACE_SYSCALL, the tracee enters syscall-enter-stop just prior to entering any system call. If the tracer restarts the tracee with PTRACE_SYSCALL, the tracee enters syscall-exit-stop when the system call is finished, or if it is interrupted by a signal. (That is, signal-delivery-stop never happens between syscall-enter-stop and syscall-exit-stop; it happens *after* syscall-exit-stop.)

Other possibilities are that the tracee may stop in a PTRACE_EVENT stop, exit (if it entered **_exit**(2) or ***exit_group****(2)*), be killed by **SIGKILL**, or die silently (if it is a thread group leader, the ***execve****(2)* happened in another thread, and that thread is not traced by the same tracer; this situation is discussed later).

Syscall-enter-stop and syscall-exit-stop are observed by the tracer as ***waitpid****(2)* returning with *WIFSTOPPED(status)* true, and *WSTOPSIG(status)* giving SIGTRAP. If the PTRACE_O_TRACESYSGOOD option was set by the tracer, then *WSTOPSIG(status)* will give the value *(SIGTRAP | 0x80)*.

Syscall-stops can be distinguished from signal-delivery-stop with SIGTRAP by querying PTRACE_GETSIGINFO for the following cases:

si_code <= 0
: SIGTRAP was delivered as a result of a user-space action, for example, a system call (***tgkill****(2)*, ***kill****(2)*, ***sigqueue****(3)*, etc.), expiration of a POSIX timer, change of state on a POSIX message queue, or completion of an asynchronous I/O request.

si_code == SI_KERNEL (0x80)
: SIGTRAP was sent by the kernel.

si_code == SIGTRAP or *si_code* == (SIGTRAP|0x80)
: This is a syscall-stop.

However, syscall-stops happen very often (twice per system call), and performing PTRACE_GETSIGINFO for every syscall-stop may be somewhat expensive.

Some architectures allow the cases to be distinguished by examining registers. For example, on x86, *rax* == -ENOSYS in syscall-enter-stop. Since SIGTRAP (like any other signal) always happens *after* syscall-exit-stop, and at this point *rax* almost never contains -ENOSYS, the SIGTRAP looks like "syscall-stop which is not syscall-enter-stop"; in other words, it looks like a "stray syscall-exit-stop" and can be detected this way. But such detection is fragile and is best avoided.

Using the PTRACE_O_TRACESYSGOOD option is the recommended method to distinguish syscall-stops from other kinds of ptrace-stops, since it is reliable and does not incur a performance penalty.

Syscall-enter-stop and syscall-exit-stop are indistinguishable from each other by the tracer. The tracer needs to keep track of the sequence of ptrace-stops in order to not misinterpret syscall-enter-stop as syscall-exit-stop or vice versa. The rule is that syscall-enter-stop is always followed by syscall-exit-stop, PTRACE_EVENT stop or the tracee's death; no other kinds of ptrace-stop can occur in between.

If after syscall-enter-stop, the tracer uses a restarting command other than PTRACE_SYSCALL, syscall-exit-stop is not generated.

PTRACE_GETSIGINFO on syscall-stops returns SIGTRAP in *si_signo*, with *si_code* set to SIGTRAP .

PTRACE_SINGLESTEP, PTRACE_SYSEMU, PTRACE_SYSEMU_SINGLESTEP stops

Informational and restarting ptrace commands

Mostptracecommands (allexcept PTRACE_ATTACH, PTRACE_SEIZE, PTRACE_TRACEME, PTRACE_INTERRUPT, and PTRACE_KILL) require the tracee to be in a ptrace-stop, otherwise they fail with ESRCH.

When the tracee is in ptrace-stop, the tracer can read and write data to the tracee using informational commands. These commands leave the tracee in ptrace-stopped state:

```
ptrace(PTRACE_PEEKTEXT/PEEKDATA/PEEKUSER, pid, addr, 0);
ptrace(PTRACE_POKETEXT/POKEDATA/POKEUSER, pid, addr, long_val);
ptrace(PTRACE_GETREGS/GETFPREGS, pid, 0, &struct);
ptrace(PTRACE_SETREGS/SETFPREGS, pid, 0, &struct);
ptrace(PTRACE_GETREGSET, pid, NT_foo, &iov);
ptrace(PTRACE_SETREGSET, pid, NT_foo, &iov);
ptrace(PTRACE_GETSIGINFO, pid, 0, &siginfo);
ptrace(PTRACE_SETSIGINFO, pid, 0, &siginfo);
ptrace(PTRACE_GETEVENTMSG, pid, 0, &long_var);
ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_flags);
```

Note that some errors are not reported. For example, setting signal information (*siginfo*) may have no effect in some ptrace-stops, yet the call may succeed (return 0 and not set *errno*);
querying PTRACE_GETEVENTMSG may succeed and return some random value if current ptrace-stop is not documented as returning a meaningful event message.

The call

ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_flags);

affects one tracee. The tracee's current flags are replaced. Flags are inherited by new tracees created and "auto-attached" via active PTRACE_O_TRACEFORK, PTRACE_O_TRACEVFORK, or PTRACE_O_TRACECLONE options.

Another group of commands makes the ptrace-stopped tracee run. They have the form:

ptrace(cmd, pid, 0, sig);

where *cmd* is PTRACE_CONT, PTRACE_LISTEN, PTRACE_DETACH, PTRACE_SYSCALL, PTRACE_SINGLESTEP, PTRACE_SYSEMU, or PTRACE_SYSEMU_SINGLESTEP. If the tracee is in signal-delivery-stop, *sig* is the signal to be injected (if it is nonzero). Otherwise, *sig* may be ignored. (When restarting a tracee from a ptrace-stop other than signal-delivery-stop, recommended practice is to always pass 0 in *sig*.)

Attaching and detaching

A thread can be attached to the tracer using

the call

ptrace(PTRACE_ATTACH, pid, 0, 0);

or

ptrace(PTRACE_SEIZE, pid, 0, PTRACE_O_flags);

PTRACE_ATTACH sends SIGSTOP to this thread. If the tracer wants this SIGSTOP to have no effect, it needs to suppress it. Note that if other signals are concurrently sent to this thread during attach, the tracer may see the tracee enter signal-delivery-stop with other signal(s) first! The usual practice is to reinject these signals until SIGSTOP is seen, then suppress SIGSTOP injection. The design bug here is that a ptrace attach and a concurrently delivered SIGSTOP may race and the concurrent SIGSTOP may be lost.

Since attaching sends SIGSTOP and the tracer usually suppresses it, this may cause a stray EINTR return from the currently executing system call in the tracee, as described in the "Signal injection and suppression" section.

Since Linux 3.4, PTRACE_SEIZE can be used instead of PTRACE_ATTACH. PTRACE_SEIZE does not stop the attached process. If you need to stop it after attach (or at any other time) without sending it any signals, use PTRACE_INTERRUPT command.

The request

ptrace(PTRACE_TRACEME, 0, 0, 0);

turns the calling thread into a tracee. The thread continues to run (doesn't enter ptrace-stop). A common practice is to follow the PTRACE_TRACEME with

raise(SIGSTOP);

and allow the parent (which is our tracer now) to observe our signal-delivery-stop.

If the PTRACE_O_TRACEFORK, PTRACE_O_TRACEVFORK, or PTRACE_O_TRACECLONE options are in effect, then children created by, respectively, *vfork(2)* or *clone(2)* with the CLONE_VFORK flag, ***fork****(2)* **or *clone(2***) with the exit signal set to SIGCHLD, and other kinds of ***clone****(2)*, are automatically attached to the same tracer which traced their parent. SIGSTOP is delivered to the children, causing them to enter signal-delivery-stop after they exit the system call which created them.

Detaching of the tracee is performed by:

ptrace(PTRACE_DETACH, pid, 0, sig);

PTRACE_DETACH

is a restarting operation; therefore it requires the tracee to be in ptrace-stop. If the tracee is in signal-delivery-stop, a signal can be injected. Otherwise, the *sig* parameter may be silently ignored.

If the tracee is running when the tracer wants to detach it, the usual solution is to send SIGSTOP (using *tgkill(2)*, to make sure it goes to the correct thread), wait for the tracee to stop in signal-delivery-stop for SIGSTOP and then detach it (suppressing **SIGSTOP** injection). A design bug is that this can race with concurrent SIGSTOPs. Another complication is that the tracee may enter other ptrace-stops and needs to be restarted and waited for again, until SIGSTOP is seen. Yet another complication is to be sure that the tracee is not already ptrace-stopped, because no signal delivery happens while it is--not even SIGSTOP.

If the tracer dies, all tracees are automatically detached and restarted, unless they were in group-stop. Handling of restart from group-stop is currently buggy, but the "as planned" behavior is to leave tracee stopped and waiting for SIGCONT. If the tracee is restarted from signal-delivery-stop, the pending signal is injected.

execve(2) under ptrace

When one thread in a multithreaded process calls ***execve****(2)*, the kernel destroys all other threads in the process, and resets the thread ID of the execing thread to the thread group ID (process ID). (Or, to put things another way, when a multithreaded process does an ***execve****(2)*, at completion of the call, it appears as though the ***execve****(2)* occurred in the thread group leader, regardless of which thread did the ***execve****(2)*.) This resetting of the thread ID looks very confusing to tracers:

* All other threads stop in PTRACE_EVENT_EXIT stop, if the PTRACE_O_TRACEEXIT option was turned on. Then all other threads except the thread group leader report death as if they exited via _exit(2) with exit code 0.
* The execing tracee changes its thread ID while it is in the ***execve****(2)*. (Remember, under ptrace, the "pid" returned from ***waitpid****(2)*, or fed into ptrace calls, is the tracee's thread ID.) That is, the tracee's thread ID is reset to be the same as its process ID, which is the same as the thread group leader's thread ID.
* Then a PTRACE_EVENT_EXEC stop happens, if the PTRACE_O_TRACEEXEC option was turned on.
*

If the thread group leader has reported its this time, it appears to the tracer that the dead thread leader PTRACE_EVENT_EXIT stop by "reappears from nowhere". (Note: the thread group leader does not report death via *WIFEXITED(status)* until there is at least one other live thread. This eliminates the possibility that the tracer will see it dying and then reappearing.) If the thread group leader was still alive, for the tracer this may look as if thread group leader returns from a different system call than it entered, or even "returned from a system call even though it was not in any system call". If the thread group leader was not traced (or was traced by a different tracer), then during ***execve****(2)* it will appear as if it has become a tracee of the tracer of the execing tracee.

All of the above effects are the artifacts of the thread ID change in the tracee.

The PTRACE_O_TRACEEXEC option is the recommended tool for dealing with this situation. First, it enables PTRACE_EVENT_EXEC stop, which occurs before ***execve****(2)* returns. In this stop, the tracer can use PTRACE_GETEVENTMSG to retrieve the tracee's former thread ID. (This feature was introduced in Linux 3.0). Second, the PTRACE_O_TRACEEXEC option disables legacy SIGTRAP generation on ***execve****(2)*.

When the tracer receives PTRACE_EVENT_EXEC stop notification, it is guaranteed that except this tracee and the thread group leader, no other threads from the process are alive.

On receiving the PTRACE_EVENT_EXEC stop notification, the tracer should clean up all its internal data structures describing the threads of this process, and retain only one data structure--one which describes the single still running tracee, with

thread ID == thread group ID == process ID.

Why it works?

It works because it provides the mechanism by which the parent process control the execution of the child process inorder not to be interrupted. when there is aneed for security This call performs the above task and then return the request for the security of the process**.**

2) BRIFELY DESCRIBE ABOUT PARAMTETERS ?

- **Request**

The value of *request* determines the action to be performed:

- **Pid**

specifies the thread ID of tracee to be acted Oo.

- **addr**

ADRESSin the tracee's memory,

- **data**

IS used to define the data elements that are to be used as incoming parameters in a Natural subprogram, external subroutine or helproutine. These parameters can be defined within the statement itself Parameter Data Definition or they can be defined outside the program in a parameter data area (PDA), with the statement referencing that data area ressing this extra SIGTRAP is the recommended approach.

3) description about the flags ?

PTRACE_O_EXITKILL

If a tracer sets this flag, **a** SIGKILL signal will be sent to every tracee if the tracer exits. This option is useful for ptrace jailers that want to ensure that tracees can never escape the tracer's control.

PTRACE_O_TRACECLONE

Stop the tracee at the next ***clone***(*2*) and automatically start tracing the newly cloned process, which will start with a SIGSTOP.
A ***waitpid***(*2*)

by the tracer will return a *status* value such that

```
status>>8 == (SIGTRAP | (PTRACE_EVENT_CLONE<<8))
```

The PID of the new process can be retrieved with PTRACE_GETEVENTMSG.

This option may not catch

clone(*2*) calls in all cases. If the tracee calls ***clone***(*2*) with the CLONE_VFORK flag, PTRACE_EVENT_VFORK will be delivered instead if PTRACE_O_TRACEVFORK is set; otherwise if the tracee calls ***clone***(*2*) with the exit signal set

to SIGCHLD, PTRACE_EVENT_FORK will be delivered
if PTRACE_O_TRACEFORK is set.

PTRACE_O_TRACEEXEC

Stop the tracee at the next ***execve**(2)*. A ***waitpid**(2)* by the tracer will return a *status*

value such that

```
status>>8 == (SIGTRAP | (PTRACE_EVENT_EXEC<<8))
```

If the execing thread is not a thread group leader, the thread ID is reset to thread group leader's ID before this stop. Since Linux 3.0, the former thread ID can be retrieved with PTRACE_GETEVENTMSG.

PTRACE_O_TRACEEXIT

Stop the tracee at exit. A ***waitpid**(2)* by the tracer will return

a *status* value such that

```
status>>8 == (SIGTRAP | (PTRACE_EVENT_EXIT<<8))
```

The tracee's exit status can be retrieved with PTRACE_GETEVENT The tracee is stopped early during process exit,
when registers are still available, allowing the tracer to see where the exit occurred, whereas the normal exit notification is done after the process is finished exiting. Even though context is available, the tracer cannot prevent the exit from happening at this point.

PTRACE_O_TRACEFORK

Stop the tracee at the next ***fork**(2)* and automatically start tracing the newly forked process, which will start with a SIGSTOP.
A ***waitpid**(2)* by the tracer will return

a *status* value such that

```
status>>8 == (SIGTRAP | (PTRACE_EVENT_FORK<<8))
```

The PID of the new process can be
retrievedwith PTRACE_GETEVENTMSG.

PTRACE_O_TRACESYSGOOD

When delivering system call traps, set bit 7 in the signal number (i.e., deliver *SIGTRAP|0x80*). This makes it easy for the tracer to distinguish normal traps from those caused by a system call.
(PTRACE_O_TRACESYSGOOD may not work on all architectures.)

PTRACE_O_TRACEVFORK

Stop the tracee at the next ***vfork**(2)* and automatically start tracing the newly vforked process, which will start with a SIGSTOP.
A ***waitpid**(2)* by

the tracer will return a *status* value such that `status>>8 == (SIGTRAP | (PTRACE_EVENT_VFORK<<8))`

The PID of the new process can be retrieved with PTRACE_GETEVENTMSG.

PTRACE_O_TRACEVFORKDONE

Stop the tracee at the completion of the next ***vfork**(2)*.
A ***waitpid**(2)* by the tracer will return a *status* value such that

`status>>8 == (SIGTRAP | (PTRACE_EVENT_VFORK_DONE<<8))`

The PID of the new process can be retrieved with PTRACE_GETEVENTMSG.

After all

Return Value If succeeded

On success, **PTRACE_PEEK*** requests return the requested data, while other requests return zero.

On error, all requests return -1, and *errno* is set appropriately. Since the value returned by a successful PTRACE_PEEK* request may be -1, the caller must clear *errno* before the call, and then check it afterward to determine `

Else

Errors Occured

EBUSY

(i386 only) There was an error with allocating or freeing a debug register.

EFAULT

There was an attempt to read from or write to an invalid area in the tracer's or the tracee's memory, probably because the area wasn't mapped or accessible. Unfortunately, under Linux, different variations of this fault will return **EIO** or **EFAULT** more or less arbitrarily.

EINVAL

An attempt was made to set an invalid option.

EIO

request is invalid, or an attempt was made to read from or write to an invalid area in the tracer's or the tracee's memory, or there was a word-alignment violation, or an invalid signal was specified during a restart request.

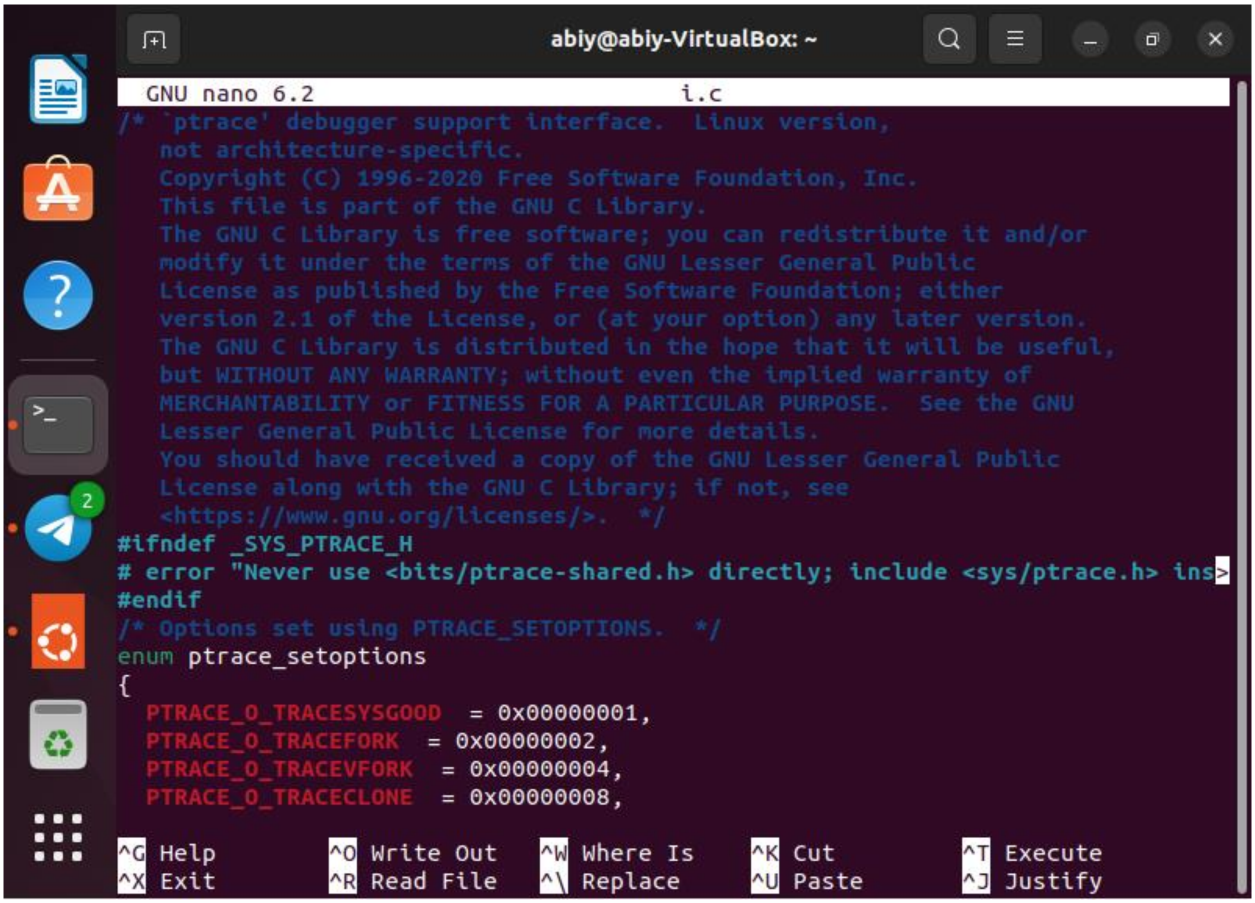
EPERM

The specified process cannot be traced. This could be because the tracer has insufficient privileges.

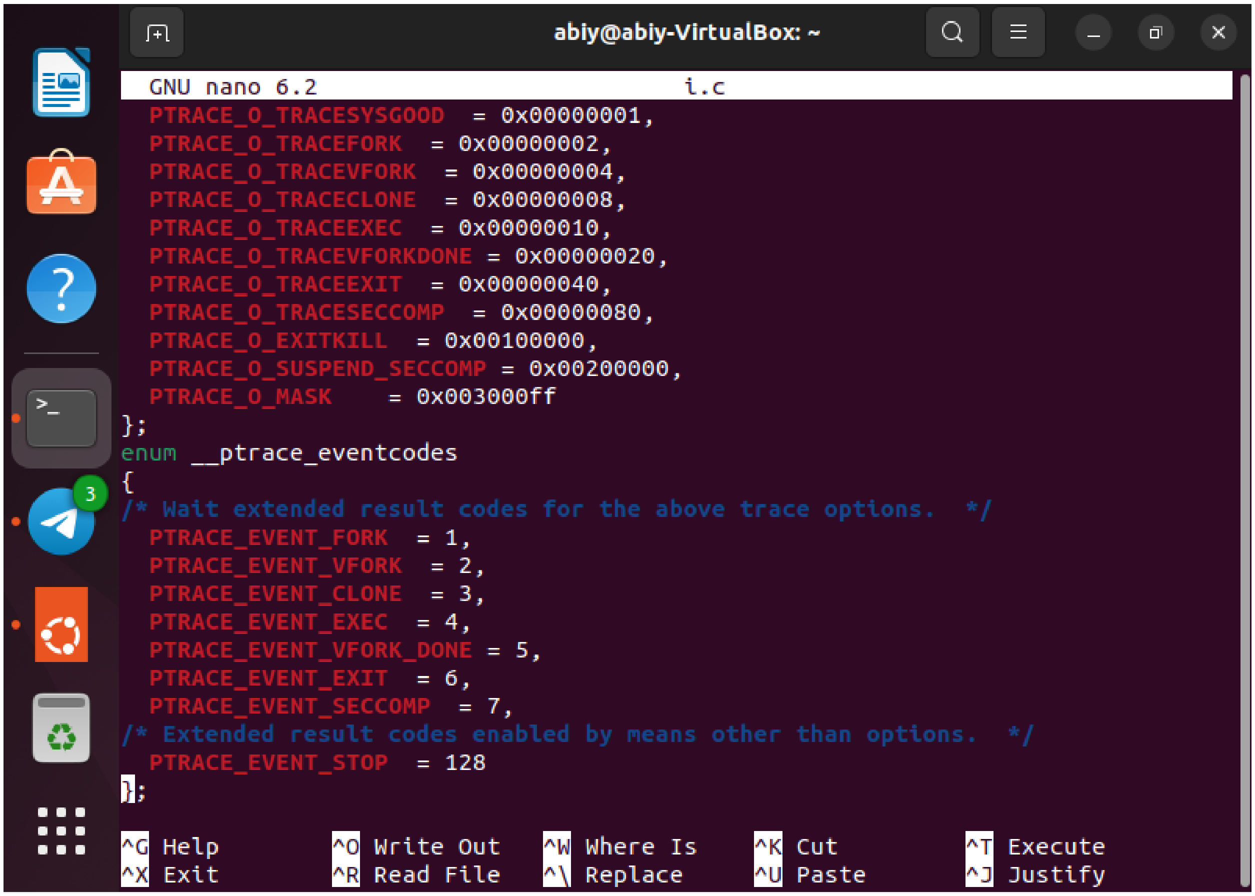
ESRCH

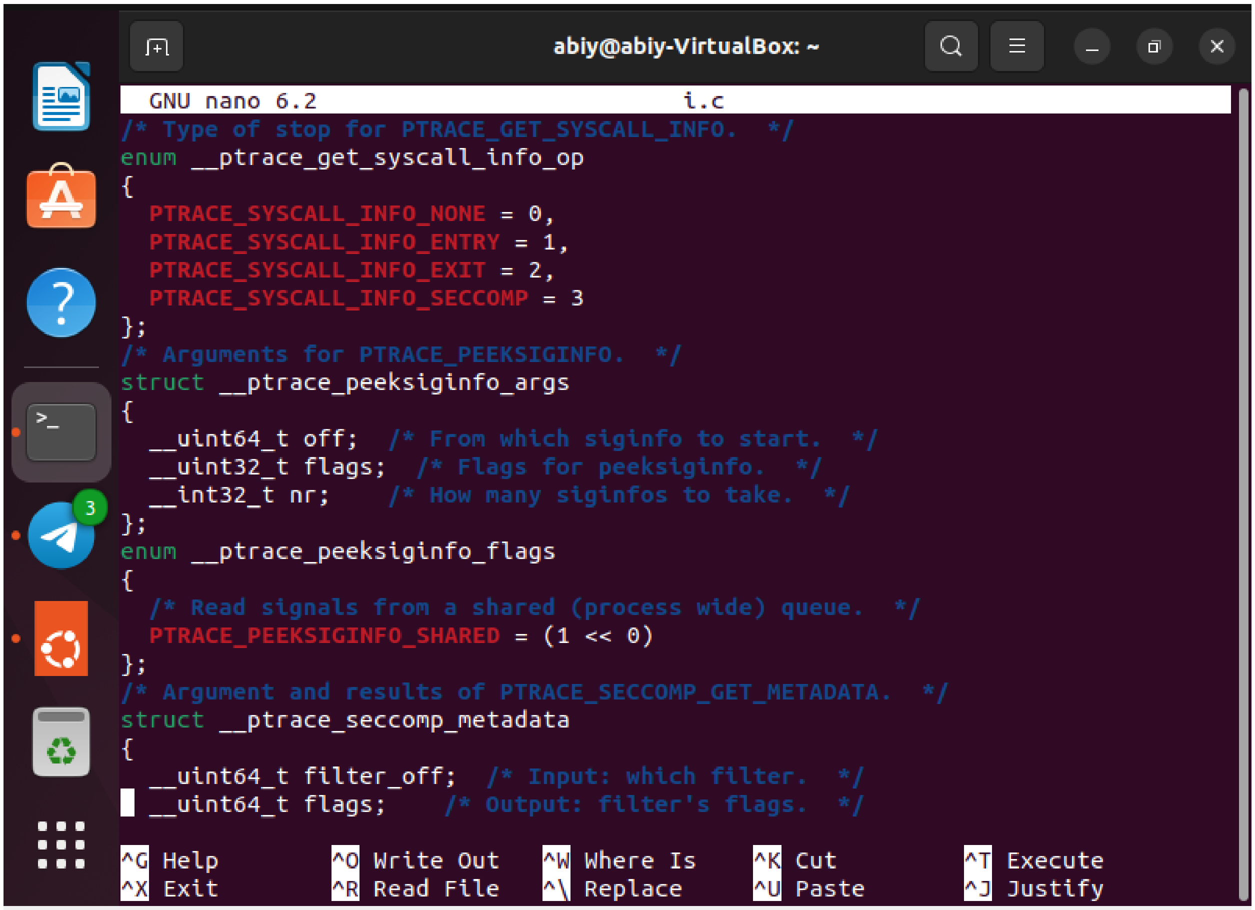
The specified process does not exist, or is not currently being traced by the caller, or is not stopped .

IMPLMENTATION



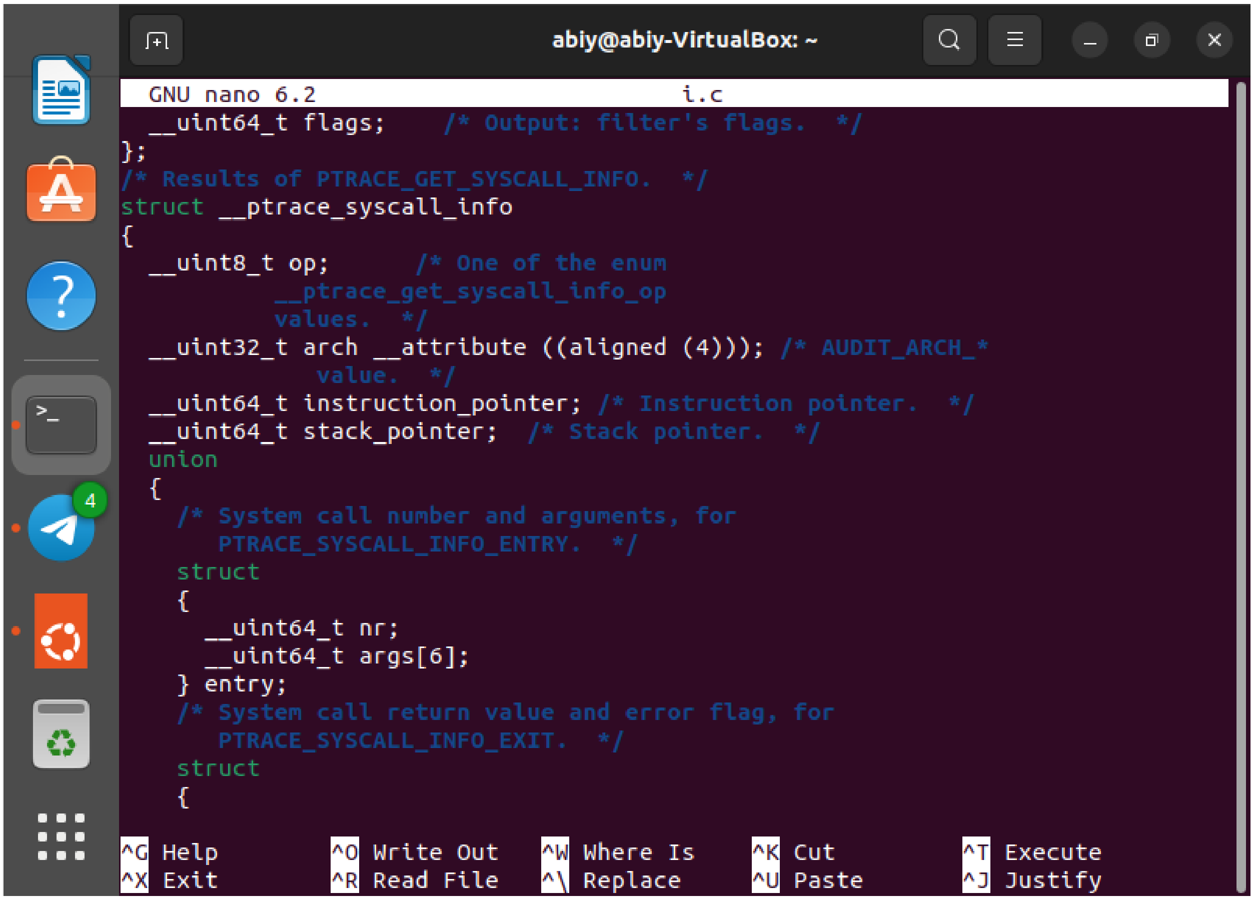

abiy@abiy-VirtualBox: ~
GNU nano 6.2
i.c
struct
{
__int64_t rval;
__uint8_t is_error;
} exit;
/* System call number, arguments and SECCOMP_RET_DATA portion of
SECCOMP_RET_TRACE return value, for
PTRACE_SYSCALL_INFO_SECCOMP. */
struct
{
__uint64_t nr;
__uint64_t args[6];
__uint32_t ret_data;
} seccomp;
};
};
/* Perform process tracing functions. REQUEST is one of the values
above, and determines the action to be taken.
For all requests except PTRACE_TRACEME, PID specifies the process to be
traced.
PID and the other arguments described above for the various requests should
appear (those that are used for the particular request) as:
pid_t PID, void *ADDR, int DATA, void *ADDR2
after REQUEST. */
^G Help
^O Write Out
^W Where Is
^K Cut
^T Execute
^X Exit
^R Read File
^\ Replace
^U Paste
^J Justify

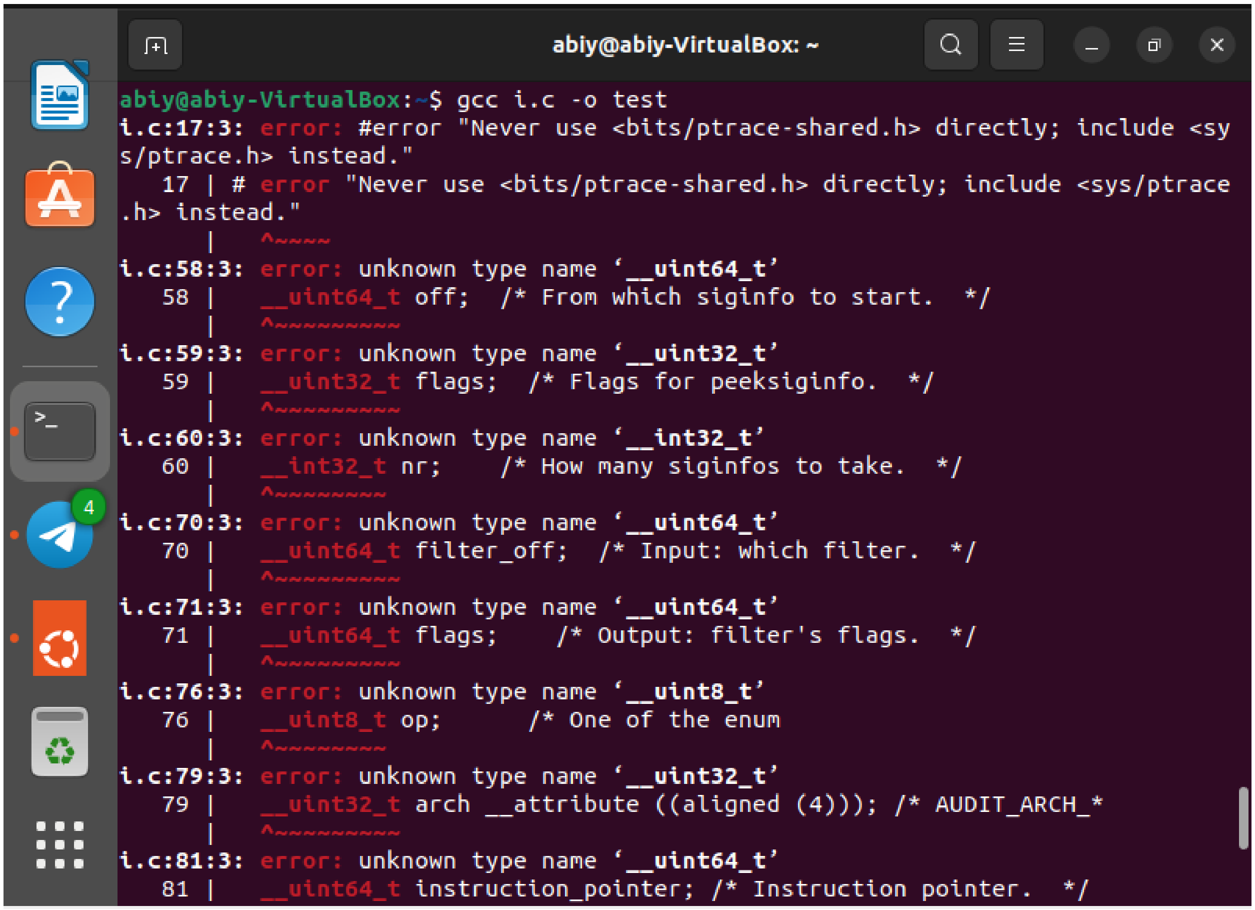

abiy@abiy-VirtualBox: ~
GNU nano 6.2 i.c
PTRACE_O_TRACESYSGOOD = 0x00000001,
PTRACE_O_TRACEFORK = 0x00000002,
PTRACE_O_TRACEVFORK = 0x00000004,
PTRACE_O_TRACECLONE = 0x00000008,
PTRACE_O_TRACEEXEC = 0x00000010,
PTRACE_O_TRACEVFORKDONE = 0x00000020,
PTRACE_O_TRACEEXIT = 0x00000040,
PTRACE_O_TRACESECCOMP = 0x00000080,
PTRACE_O_EXITKILL = 0x00100000,
PTRACE_O_SUSPEND_SECCOMP = 0x00200000,
PTRACE_O_MASK = 0x003000ff
};
enum __ptrace_eventcodes
{
/* Wait extended result codes for the above trace options. */
PTRACE_EVENT_FORK = 1,
PTRACE_EVENT_VFORK = 2,
PTRACE_EVENT_CLONE = 3,
PTRACE_EVENT_EXEC = 4,
PTRACE_EVENT_VFORK_DONE = 5,
PTRACE_EVENT_EXIT = 6,
PTRACE_EVENT_SECCOMP = 7,
/* Extended result codes enabled by means other than options. */
PTRACE_EVENT_STOP = 128
};
^G Help
^O Write Out
^W Where Is
^K Cut
^T Execute
^X Exit
^R Read File
^\ Replace
^U Paste
^J Justify



```
GNU nano 6.2                              i.c
/* Type of stop for PTRACE_GET_SYSCALL_INFO.  */
enum __ptrace_get_syscall_info_op
{
  PTRACE_SYSCALL_INFO_NONE = 0,
  PTRACE_SYSCALL_INFO_ENTRY = 1,
  PTRACE_SYSCALL_INFO_EXIT = 2,
  PTRACE_SYSCALL_INFO_SECCOMP = 3
};
/* Arguments for PTRACE_PEEKSIGINFO.  */
struct __ptrace_peeksiginfo_args
{
  __uint64_t off;  /* From which siginfo to start.  */
  __uint32_t flags;  /* Flags for peeksiginfo.  */
  __int32_t nr;    /* How many siginfos to take.  */
};
enum __ptrace_peeksiginfo_flags
{
  /* Read signals from a shared (process wide) queue.  */
  PTRACE_PEEKSIGINFO_SHARED = (1 << 0)
};
/* Argument and results of PTRACE_SECCOMP_GET_METADATA.  */
struct __ptrace_seccomp_metadata
{
  __uint64_t filter_off;  /* Input: which filter.  */
  __uint64_t flags;    /* Output: filter's flags.  */

^G Help        ^O Write Out   ^W Where Is    ^K Cut         ^T Execute
^X Exit        ^R Read File   ^\ Replace     ^U Paste       ^J Justify
```


abiy@abiy-VirtualBox: ~
GNU nano 6.2 i.c
__uint64_t flags; /* Output: filter's flags. */
};
/* Results of PTRACE_GET_SYSCALL_INFO. */
struct __ptrace_syscall_info
{
__uint8_t op; /* One of the enum
__ptrace_get_syscall_info_op
values. */
__uint32_t arch __attribute ((aligned (4))); /* AUDIT_ARCH_*
value. */
__uint64_t instruction_pointer; /* Instruction pointer. */
__uint64_t stack_pointer; /* Stack pointer. */
union
{
/* System call number and arguments, for
PTRACE_SYSCALL_INFO_ENTRY. */
struct
{
__uint64_t nr;
__uint64_t args[6];
} entry;
/* System call return value and error flag, for
PTRACE_SYSCALL_INFO_EXIT. */
struct
{
^G Help
^O Write Out
^W Where Is
^K Cut
^T Execute
^X Exit
^R Read File
^\ Replace
^U Paste
^J Justify



Reference

1 https://clickhouse.com

2 https://linux.die.net

THANKSTHE END