Volkmar SIEH

Fault-Injector using UNIX `ptrace` Interface

Internal Report No.: 11/93

1. Introduction

The goal of this internal report is to summarize the experiences with the fault injector based on the Unix ptrace function. It should be explained how to use the fault injector for special experiments and how to interpret the results.

In the first chapter (“ptrace(2) Interface“) an overview on the facilities of the Unix ptrace interface is given to see what faults/errors may be simulated by this interface. The limitations of this function are shown.

The currently implemented faults which can be injected by the fault injector are enlisted in the second part of this paper (“Fault Model“). It is described how to implement other fault injecting functions to the fault injector program.

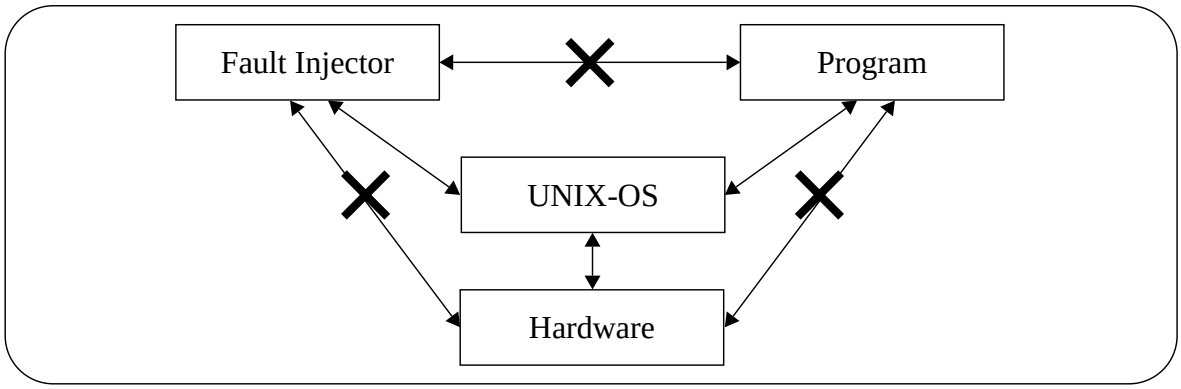
The next chapter (“Measurements using the Fault Injector“) gives hints to the user how to interpret the results of the fault injector used in experiments. Many effects may influence the measurements. Some of them are discussed in this part of the paper.

The syntax of the input and output of the fault injector is presented in the section “Command Language and Output Format“. Some examples are given to ease the use of this program.

The fault injector was used to evaluate fault coverage and fault latency of some methods of fault tolerance. The experiences of this experiments and a summary of the advantages and disadvantages are given in the last section of this paper (“Experience“).

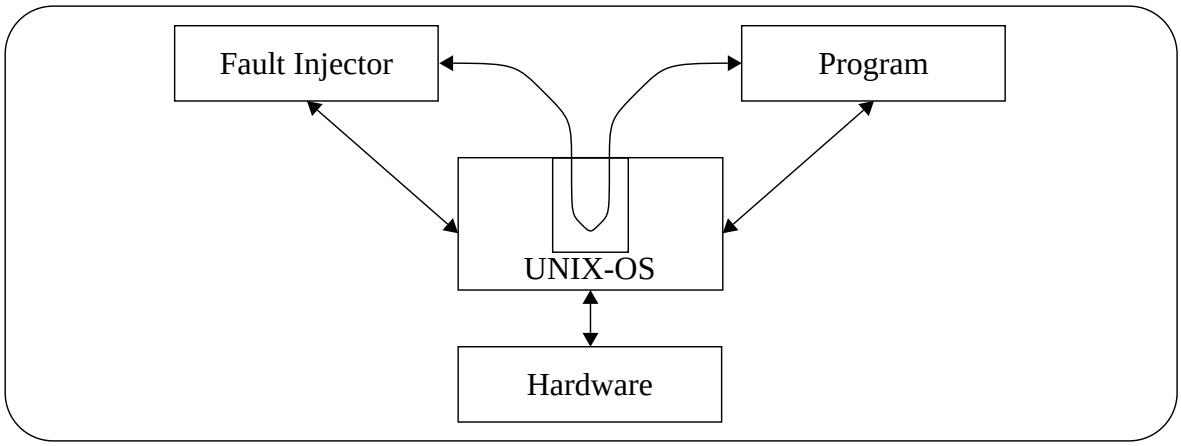
2. `ptrace`(2) Interface

Normally no Unix process has access to the internal state of another process. The processes are separated from each other by using different physical address ranges protected by the MMU and time shared use of the CPU. Direct access to hardware registers is prohibited for every process.



The `ptrace`(2) interface of the Unix operating system was originally designed to allow debugging of single processes. A debugger must be able to read and modify the local and global variables and the program counter of his debuggee. The data flow between the debugger and the

tested program is checked by the kernel. So changes in the state of the debugged process cannot affect other running processes. The normal multi-user and multi-tasking facilities of the UNIX operating system are still provided.



Access to the internal state of a process is granted only if the following conditions are met:

1. The tested process must agree to be debugged (see `PTRACE_TRACEME` below).

2. The debugged process must be stopped. Every signal (except `SIGKILL`) sent to the process stops it. The kernel procedures `wait`(2), `waitpid`(2), `wait3`(2) or `wait4`(2) can be used to determine if the debugged process is stopped, signaled or exited.

The functionality of the `ptrace`(2) interface is nearly identical on most Unix systems. Only the types of the parameters and the names of the constants differ slightly (see the manual entry for `ptrace` in the actual systems on-line manual for the exact syntax and semantic of `ptrace`).

In the following a description of the SunOS `ptrace`(2) is given. The names used by the operating systems Motorola-System-V (Memsos), Linux and Dynix are given in the appendix (“Constants for ptrace“).

Syntax of the `ptrace`(2) function:

```
#include <sys/types.h>
#include <sys/ptrace.h>

int
ptrace(request, pid, addr1, data, addr2)
    int request, pid;
    int data;
    void *addr1, *addr2;
```

`request` defines the sub-function to be performed by `ptrace`. *`pid`* is the process id of the traced process. The meaning of the parameters *`addr1`*, *`data`* and *`addr2`*[1] varies depending on the *`request`* parameter (see below).

1. The *`addr2`* parameter is only used by SunOS with the `PTRACE_READTEXT`, `PTRACE_WRITETEXT`, `PTRACE_READDATA` and `PTRACE_WRITEDATA` calls.

The following constants are defined in `<sys/ptrace.h>` and can be used as *`request`* parameter:

- `PTRACE_TRACEME`
  This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by func (see `signal`(2)). The *`pid`*, *`addr1`*, *`data`* and *`addr2`* arguments are ignored, and a return value is not defined for this request. If the process calls `execve`(2) it will load the new program image and stop before executing the first statement of the loaded program. Peculiar results will ensue if the parent does not expect to trace the child.

- `PTRACE_CONT`
  Continue the stopped process. If *`data`* is not 0 it will be interpreted as a signal number. This signal is given to the debugged process before it is restarted.

- `PTRACE_SINGLESTEP`
  Executes exactly one instruction and stops again afterwards.

- `PTRACE_SYSCALL` (SunOS only)
  Continue, but stop before calling a system function and stop after executing it.

- `PTRACE_KILL`
  The debugged process is terminated. It terminates with the same consequences as if it has called `_exit`().

- `PTRACE_DUMPCORE` (SunOS only)
  Same as `PTRACE_KILL` but write core file before terminating process.

- `PTRACE_PEEKTEXT`
  One word of the text segment is given to the debugger. *`addr1`* is the address of the requested instruction.

- `PTRACE_READTEXT`
  Read data bytes from the text segment from address *`addr2`* to address *`addr1`* in the debugger address space

- `PTRACE_PEEKDATA`
  Same as `PTRACE_PEEKTEXT` but for a data segment (data, bss and stack segment)

- `PTRACE_READDATA` (SunOS only)
  Same as `PTRACE_READTEXT` but for a data segment.

- `PTRACE_PEEKUSER`
  `ptrace`(2) will return the value at offset *`addr1`* in the user structure (see `<sys/user.h>`) of the debugged process. With this function it is possible to determine the values of single CPU and FPU registers, open files and chosen signal reactions.

- `PTRACE_POKETEXT`
  Write the word data into the text segment at address *`addr1`*. Can be used to insert breakpoints into the program text.

- `PTRACE_WRITETEXT` (SunOS only)
  Write data bytes from address *`addr1`* in the debugger address space to *`addr2`* in the debuggee address space.

- `PTRACE_POKEDATA`
  Same as `PTRACE_POKETEXT` but for a data segment.

- `PTRACE_WRITEDATA` (SunOS only)
  Same as `PTRACE_WRITETEXT` but for a data segment.

- `PTRACE_POKEUSER`
  Write word *`data`* at offset *`addr1`* into the user structure of the debugged process. Only a few values of the user structure may be changed. Most are read-only.

- `PTRACE_GETREGS` (SunOS only)
  Get all CPU registers at once. *`addr1`* is an address, where to store the registers contents.

- `PTRACE_SETREGS` (SunOS only)
  Set all CPU registers.

- `PTRACE_GETFPREGS` (SunOS only)
  Get all FPU registers

- `PTRACE_SETFPREGS` (SunOS only)
  Set all FPU registers

3. Fault Model

Using the `ptrace`(2) interface described in Section 2 a process is able to inject faults to another process. The fault injector has to fork a child which runs the program under test. The state of this child can be observed and modified with the `ptrace`(2) function.

Examining the different facilities of `ptrace`(2), one can see that faults/errors can be injected into the following parts of the state of the tested process:

- Into most of the CPU registers (program counter, stack pointer, data and address registers and into the condition code registers). Some bits remain unaltered, because a change in this bits may cause a failure of the whole Unix system (e.g. the interrupt mask or supervisor bits). This is enforced by the system without causing an error to be indicated.

- All FPU and FPA registers may be changed.

- Any word in the address space of the tested process may be changed (text, data, bss, shared memory and stack segment). The stack will be expanded if necessary.

Until now a variety of different functions has been implemented in the fault injector. Each of these functions injects one special fault/error into the target program. This set of fault injection functions can be easily extended to special needs.

Implemented are:

- Transient single bit, double bit or word errors in one of the CPU registers.
- Transient single bit, double bit or word errors in one of the FPU registers.
- Transient single bit, double bit, single word, double word, single page or double page errors in the text, data, bss or stack segment.

(The bit number in the registers, the number of the modified register and the addresses of the altered word in memory are calculated with the standard random function `rand`(3) of the standard C-library.)

The different names of the fault injecting functions are listed in the appendix. There are different functions for each individual architecture (as different processors have different register sets).

Possible extensions and modifications are (for example):

- Permanent and intermittent faults/errors in one of the registers or memory cells using single step mode.
- Simulation of incorrectly functioning arithmetic or logical units.
- More complex faults/errors.
- Modification of the random function.

4. Measurements using the Fault Injector

The fault injector is able to start a program under test, to inject errors into its internal state and to observe its behavior.

4.1 Program Exit Status

When the tested program terminates it is possible to get information on the process exit cause. There are two main events which will terminate a process:

- The program calls `_exit`(2).
- The program receives a signal and dies (with or without a core dump depending on the signal).

The user has to determine whether the program terminated normally or not, whether the results are correct and done in the correct range of time. This cannot be done by the fault injector because the fault injector doesn't know the correct results and the correct timing of the tested program.

To allow automatic statistical testing a time-out value can be defined for the fault injector. After this time (or after this number of executed instructions) the tested program is terminated and so another new test run can be started. For this purpose the tested program must not alter the reac-

tion upon the `SIGALRM` signal because the fault injector sends this time out signal to its child to terminate it. All other reactions upon receipt of a signal may be altered by the tested program. The delivery of signals other than the alarm signal is not affected by the fault injector.

4.2 Time Resolution

In the following some of the effects are described which can affect the measurements of time.

- Unfortunately most Unix systems have a clock resolution of only 1/60 or 1/100 second, so it is difficult to measure time intervals in the range of only a few milliseconds. Only the Sun Sparc station has a real-time clock with a resolution of one microsecond.

- Furthermore the `ptrace`(2) function works very slow. Six context switches between the tested process and the fault injector are necessary to transfer a single word from one process to another (one fault injection). Every context switch takes a long time because the contents of the MMU and the contents of the cache must be updated to execute another process. If the tested process uses very much memory space swapping or paging has to be done, too. So, even if the process dies immediately after a fault injection it will take round about ten to fifty milliseconds (depending on the system) to observe its death.

- If a system is executing multiple processes than the tested process potentially is not scheduled immediately after injecting a fault and executing `ptrace(PTRACE_CONT, pid, 1, 0)`. So there might be an extra delay between fault injection and death of the tested process. This problem can be eliminated using a stand-alone system in single user mode with net interfaces configured down.

- If a process dies on a signal normally indicating a program error (`SIGQUIT`, `SIGILL`, `SIGABRT`, `SIGEMT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS` see `signal`(2)) it will write a core dump into the current working directory. Writing a core dump will delay the notification of the fault injector by about half a second up to a few seconds!

  Hint:
  The tested program doesn't dump a core file if a file exists in the current directory with the name `core` (or `core`.name-of-program on the Motorola System V Unix or Memsos) which is no plain file or which is not writable (no write permission for the current user or which is part of a read-only file system). The easiest way to prohibit core dumping is to create a symbolic link to `/dev/null` in the current working directory (`ln -s /dev/null core`).

- Nearly all Unix systems do paging. If there is insufficient physical memory some pages may be written to a secondary storage (swap space; normally a hard disk). Access to these pages is much slower than read or write operations on internal memory. So a faulty access to one of these pages will increase latency time.

  Example: At first the fault is injected and the fault time is saved, then the tested process is continued. The process will access a page not resident in memory and will be stopped until the page is transferred. This will take some time (~3 milliseconds on a sparc station). After the page is loaded the process will continue and die because of an invalid memory access. Now the time is stopped. The time between the fault and the failure will be greater than 3 milliseconds even the process has executed only one single instruction!

Because of these effects the real error latency can only roughly be estimated.

4.3 Single Step Mode

In most experiments it might be more accurate to measure the number of instructions executed before a failure instead of the latency. This may be done by using the single step mode of the CPU's. The number of executed instruction is a deterministic value whereas the latency time may be changed by various events (e.g. process switch, core dumping, system load etc. as described above).

Unfortunately single step mode is very slow on most systems. Only a few thousands of instructions are executed in one second. Because of this only small applications of a short run-time may be executed step by step. Processes executing realistic, long running programs in single step mode will take an realistically long time. But it is still possible to mix single step mode and normal execution of instructions (see example in section "Command Language and Output Format").

Hint:
Using the `-Bstatic` option of the compiler on a Sun-3 or Sun-Sparcstation to generate static linked programs will decrease the number of steps necessary to execute start up and signal handling code.

5. Command Language and Output Format

In this section a description is given how to use the fault injector. The first part describes the calling syntax of the program "`fault_injector`", the next part explains the format of the control file and the last part gives an explanation on the output of the fault injector.

Calling syntax:

```
fault_injector control-file program [program-args ...]
```

`control-file` is the name of a file which contains statements (see below) to be executed by the fault injector. `program` is the name of that program that should be tested. `program-args` are arguments given to the program `program` before starting it.

The control file for the fault injector contains lines in pure ASCII code. Every line of this file contains one single instruction for the injector. It consists of a delay (time or steps) and an action to perform. The syntax is summarized in the following diagram (extended Backus-Naur form; spaces and tabs may be inserted between keywords):

```
control-file::=
      (delay action '\n')*
    | delay 'timeout' '\n'

delay::=
      number 'microsecond' ['s']
    | number 'step' ['s']

number::=
```

```
    ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')+

action::=
      'nop'
    | '1_bit_pc'
    | '1_bit_sp'
    ...
    | 'word_data'
```

The keywords given as "`action`" vary on different system architectures (see the appendix "Currently implemented fault/error injecting functions" for all allowed actions on a given system). If an error in the control file is recognized by the injector the program is not started and an error message is written to `stderr`.

The output of the fault injector - which is written to `stderr` to separate it from the normal output of the tested program - consists of time stamps and log messages on the actions performed by the injector and observed reactions of the tested program. The syntax is summarized in the following diagram (extended Backus-Naur form):

```
output::=
      (time-stamp action-log)*
    | time-stamp term-mesg

time-stamp::=
      number ' microsecond(s)\n'
    | number ' step(s)\n'

action-log::=
      'nop\n'
    | 'inject fault ' number ' (' fault-desc ')\n'

number::=
    ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')+

term-mesg::=
      program-name ' exited with exit code ' number '\n'
    | program-name ' died on signal ' number ' (' signal-desc ')\n'
```

In the syntax diagram "`program-name`" is the name of the started program, "`fault-desc`" is a detailed description of the injected fault and "`signal-desc`" is a description of the signal.

5.1 Example

Contents of control file `desc.ctrl`:

```
1000 steps nop
100000 microseconds 1_bit_sp
500000 microseconds timeout
```

This means:
The started program will run 1000 steps without any modification (single step modus). Afterwards it will run 0.1 seconds (100.000 microseconds; this time may vary depending on the current work-load of the system, see “Time Resolution” of the previous chapter). Then a single bit of the stack pointer will be flipped and the program is continued again. If the program is not terminated by an `_exit`(2) call or a received signal in an interval of half of a second (500.000 microseconds) it will be terminated by the fault injector.

Execution of the fault injection experiment:

```
fault_injector desc.ctrl /bin/ls -l /
```

Output of the fault injector program `fault_injector` (`stderr`) and `/bin/ls` program (`stdout`):

```
example%> fault_injector desc.ctrl /bin/ls -l /
1000 step(s)
nop
total 3497
drwxr-xr-x  2 root          512 Nov 30 15:36 afs
drwxr-xr-x  8 root          512 Nov 30 13:49 amd
lrwxrwxrwx  1 root            7 Mar 14  1991 bin -> usr/bin
drwxr-sr-x  2 bin         11264 Nov 22 09:07 dev
drwxr-xr-x 10 bin          5632 Dec  1 13:30 etc
107133 microsecond(s)
inject fault 1 (1-BIT ERROR o6 (f7ffef08 -> d7ffef08) - HIT)
468512 microsecond(s)
/bin/ls died on signal 11
example%> _
```

Explanation
First the fault injector program is started and it starts the `/bin/ls` program. 1000 instructions of the test program are executed (1000 step(s) nop). `ls` didn’t print any message in that time (the start up code needs execution of 4000 (static binding) to 200.000 (dynamic binding) instructions!). In the next 107133 microseconds `ls` prints the first six lines of its output. Then the first fault/error is injected (`inject fault 1`). The stack pointer (register `o6` on a sparc station) is changed from `f7ffef08` to `d7ffef08` (bit 29 is flipped). After 468512 microseconds the fault injector got the message that his child died on a segmentation violation (signal 11 = `SIGSEGV`).(Writing a core dump and waking up the parent process takes a long time! Here in this example nearly half of a second!)

6. Experience

Many hardware and software fault injectors exists. Some papers describing them are: [3], [5]. They all have the disadvantage that are special purposed hardware or programs for only one test environment. Porting them to another environment is impossible without much hardware changing and/or programming work.

In the following the main advantages and disadvantages of a software fault injector using the `ptrace`(2) interface of the Unix kernel are listed:

Advantages:

- This kind of fault injector can be ported to any Unix based system with only few changes. No part of the injector uses assembler sources.
- Normal service to other users remains unaffected. The system can run in multi-user mode because the environment (e.g. the operating system and file system) of the tested program is unchanged by the injector.
- The executable code of the injector is very small (about 50 kbyte). So the running fault injector doesn't require too much memory.
- It isn't necessary to modify the programs under test. They needn't be altered neither in source code nor even be recompiled. Faults can be injected into the unaltered binaries.

Disadvantages:

- Only the reactions upon changes to the internal state can be tested. The environment of the program can't be changed. So the influence of faults in the operating system or in external devices is not checked.
- A process may only be affected by injected faults if it is not currently executing subroutines of the operating system.
- By using the `ptrace`(2) interface of the kernel one has only access to the virtual address space of the tested program. For example it is impossible to distinguish between memory cells in main memory, in the processor cache or on the swap space.
- Only one process can be tested. If that process creates child processes, they are not included to the tested part of the system. This may be a crucial problem in multiprocessor systems.

Some experiments were made using the fault injector (see for example [1] and [2]).

7. References

[1] Joachim Hönig, Volkmar Sieh; "Software-Based Concurrent Control Flow Checking"; submitted to Proc. 24th Int. Symposium on Fault Tolerant Computing (FTCS-24), 1993

[2] A. Pataricza, I. Majzik, M. Dal Cin, W. Hohl, J. Hönig, V. Sieh; "Fast Watchdog Processors for Multiprocessor Systems"; submitted to Proc. 24th Int. Symposium on Fault Tolerant Computing (FTCS-24), 1993

[3] Benyó Balázs; "Fault injection based dependability analysis"; diploma thesis at the Technical University of Budapest, Dept. Measurement and Instrument Engineering, 1993

[4] Thomas R. Dilenno, David A. Yaskin, James H. Barton; "Fault Tolerance

Testing in the Advanced Automation System”; Proc. 21st Int. Symposium on Fault Tolerant Computing (FTCS-21), page 18-26, 1991

[5] Ghani A. Kanawati, Nasser A. Kanawati, Jacob A. Abraham; “FERRARI: A Tool for The Validation of System Dependability Properties”; Proc. 22nd Int. Symposium on Fault Tolerant Computing (FTCS-22), page 336-344, 1992

[6] Tomislav Lovric, Klaus Echtle; “ProFI: Processor Fault Injection for Dependability Validation”;

[7] W. C. Carter, R. K. Iyer; “Panel Sessions on Measurement and Experimentation Dependability and Performance Evaluation”; Proc. 14th Int. Symposium on Fault Tolerant Computing (FTCS-14), page 122-125, 1984

[8] A. Pataricza, I. Majzik, W. Hohl, J. Hönig; “Watchdog processors in Parallel Systems”; EUROMICRO ’93, 19th Symposium on Microprocessing and Microcomputing Microprocessors and Microsystems (in press), 1993

8. Appendix

8.1 Currently implemented fault/error injecting functions

List of currently implemented fault/error injecting functions[1] of the fault injector (see "`injector.c`"):

Table 1:

name of fault	available on	description
1_bit_pc, 2_bit_pc, word_pc	sun3, sun4, sequent386_ptx, linux	1, 2 or up to 32 bits are modified in the program counter register
1_bit_xpc, 2_bit_xpc, word_xpc	mot88	1, 2 or up to 32 bits are modified in the execution program counter register
1_bit_npc, 2_bit_npc, word_npc	sun4, mot88	1, 2 or up to 32 bits are modified in the next program counter register
1_bit_fpc, 2_bit_fpc, word_fpc	mot88	1, 2 or up to 32 bits are modified in the fetch program counter register
1_bit_sp, 2_bit_sp, word_sp	sun3, sun4, mot88, sequent386_ptx, linux	1, 2 or up to 32 bits are modified in the stack pointer
1_bit_sr, 2_bit_sr, word_sr	sun3, sun4, mot88, sequent386_ptx, linux	1, 2 or up to 32 bits are modified in the status and condition code register
1_bit_reg, 2_bit_reg, word_reg	sun3, sun4, mot88, sequent386_ptx, linux	1, 2 or up to 32 bits are modified in one of the other registers (data or address register)
1_bit_text, 2_bit_text, 1_word_text, 2_word_text, 1_page_text, 2_page_text	sun3, sun4, mot88, sequent386_ptx, linux	1 or 2 bits, 1 or 2 words, 1 or 2 pages are modified in the text segment

1. bit numbers, register numbers, word addresses and page addresses are calculated using the `rand`(3) function

Table 1:

name of fault	available on	description
1_bit_data, 2_bit_data, 1_word_data, 2_word_data, 1_page_data, 2_page_data	sun3, sun4, mot88, sequent386_ptx, linux	1 or 2 bits, 1 or 2 words, 1 or 2 pages are modified in one of the data segments (data, bss or stack segment)

8.2 Constants for `ptrace`

The following constants are defined in `<sys/ptrace.h>` and can be used as request parameter for the `ptrace` function (see section “ptrace(2) Interface”):

Table 2:

SunOS	Motorola/Memsos	Linux	Dynix
PTRACE_TRACEME	PT_TRACE_ME	PTRACE_TRACEME	PT_CHILD
PTRACE_CONT	PT_CONTINUE	PTRACE_CONT	PT_CONTSIG
PTRACE_SINGLESTEP	PT_STEP	PTRACE_SINGLESTEP	PT_SSTEP
PTRACE_SYSCALL	-	-	-
PTRACE_KILL	PT_KILL	PTRACE_KILL	PT_KILL
PTRACE_DUMPCORE	-	-	-
PTRACE_PEEKTEXT	PT_READ_I	PTRACE_PEEKTEXT	PT_RTEXT
PTRACE_READTEXT	-	-	-
PTRACE_PEEKDATA	PT_READ_D	PTRACE_PEEKDATA	PT_RDATA
PTRACE_READDATA	-	-	-
PTRACE_PEEKUSER	PT_READ_U	PTRACE_PEEKUSR	PT_RUSER
PTRACE_POKETEXT	PT_WRITE_I	PTRACE_POKETEXT	PT_WTEXT
PTRACE_WRITETEXT	-	-	-
PTRACE_POKEDATA	PT_WRITE_D	PTRACE_POKEDATA	PT_WDATA
PTRACE_WRITEDATA	-	-	-
PTRACE_POKEUSER	PT_WRITE_U	PTRACE_POKEUSR	PT_WUSER
PTRACE_GETREGS	-	-	-
PTRACE_SETREGS	-	-	-
PTRACE_GETFPREGS	-	-	-
PTRACE_SETFPREGS	-	-	-