What could be done in the kernel to make strace happy

Dmitry Levin

Linux Plumbers Conference 2018

1. There is no kernel API to find out whether the tracee is entering or exiting syscall.
2. There is no reliable way to distinguish between x86_64 and x86 syscalls.
3. There is no kernel API to invoke **wait4** syscall with a different signal mask (like **pselect6** and **ppoll**).
4. The time precision provided by **struct rusage** is too low for syscall statistics (**strace -c**).
5. There is no proper kernel API to translate between tracer and tracee views of pids.

---

6. There is no way to obtain network protocol details for descriptors of tracees running in different namespaces (**strace -yy**).
7. There are no consistent detailed declarative syscall descriptions, this forces every user to reinvent its own wheel and catch up with the kernel.
8. strace is slow, perf can lose data

Problem

Both **syscall-enter-stop** and **syscall-exit-stop** look the same for the tracer,
there is no kernel API to distinguish them.

Workaround

strace does its best to keep track of the sequence of ptrace events. When attaching to a tracee inside exec, however, its first syscall stop is ***very likely*** going to be **syscall-exit-stop** instead of **syscall-enter-stop**, the workaround is fragile.

Problem

There is no reliable way to distinguish between x86_64 and x86 syscalls.

Current practice

```
union {
    struct x86_64_user_regs_struct    x86_64_r;
    struct i386_user_regs_struct      i386_r;
} x86_regs_union;
struct iovec x86_io = {
    .iov_base = &x86_regs_union,
    .iov_len = sizeof(x86_regs_union)
};
rc = ptrace(PTRACE_GETREGSET, pid, NT_PRSTATUS, &x86_io);
...
if (x86_io.iov_len == sizeof(x86_regs_union.i386_r)) {
    scno = x86_regs_union.i386_r_regs.orig_eax;
    currpers = 1;
} else {
    scno = x86_regs_union.x86_64_r.orig_rax;
    currpers = 0;
}
```

Problem

In infamous case of int 0x80 on x86_64 the PTRACE_GETREGSET approach does not work.

Example

```
$ cat int_0x80.c
#include <stdio.h>
int main(void) {
    /* 200 is __NR_getgid32 on x86 and __NR_tkill on x86_64. */
    __asm__("movq $246, %rsi; movq $135, %rdi; movq $200, %rax; int $0x80");
    printf("getegid returns %d\n", getegid());
    return 0;
}
$ gcc -Wall -O2 int_0x80.c
$ strace -qq -etrace=tkill,/getegid ./a.out
tkill(135, 246)                         = 500
getegid()                               = 500
getegid returns 500
```

Extend the ptrace API with PTRACE_GET_SYSCALL_INFO request, use it **instead of** PTRACE_GETREGSET et al

```
struct ptrace_syscall_info {
    __u8    op;  /* 0 for entry, 1 for exit */
    __u8    __pad0[7];
    union {
        struct {
            __s32   nr;
            __u32   arch;
            __u64   ip;
            __u64   args[6];
        } entry_info;
        struct {
            __s64   rval;
            __u8    is_error;
            __u8    __pad2[7];
        } exit_info;
    };
};
```

RFC patch and discussion: https://lkml.org/lkml/2018/11/7/313

strace main loop in case of delay injection enabled

```
for (;;) {
    /* What if the timer has expired at this point?  */
    pid = wait4(-1, &status, __WALL, &rusage);
    handle_tracee(pid, status, &rusage);
}
```

Problem

There is no kernel API to invoke **wait4** syscall with a different signal mask, similar to **pselect6** extension over **select** and **ppoll** over **poll**.

Workaround

strace does its best to implement a race-free workaround by doing a lot of non-trivial work inside a signal handler. This is way too complex and very fragile.

Add **pwait6** syscall

Similar to **pselect6** extension over **select** and **ppoll** over **poll**, add **pwait6** syscall which is **wait4** with additional signal mask arguments:

```
pid_t
wait4(pid_t pid, int *wstatus,
      int options, struct rusage *rusage);
pid_t
pwait6(pid_t pid, int *wstatus,
       int options, struct rusage *rusage,
       const sigset_t *sigmask, size_t sigsetsize);
```

The time precision provided by **struct rusage** is too low for syscall statistics

```
$ strace -c -e%file pwd > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 53.09    0.000043          43         1           execve
 30.86    0.000025          12         2           openat
 16.05    0.000013          13         1         1 access
  0.00    0.000000           0         1           getcwd
------ ----------- ----------- --------- --------- ----------------
100.00    0.000081                     5         1 total

$ strace -c -e%file pwd > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.000009           9         1           getcwd
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2           openat
------ ----------- ----------- --------- --------- ----------------
100.00    0.000009                     5         1 total
```

Use a better structure than **struct rusage** in the new **pwait6** syscall

Replace **struct rusage** argument of the new **pwait6** syscall with **struct rusage_ts64**:

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    ...
}

struct rusage_ts64 {
    struct timespec64 ru_utime; /* user CPU time used */
    struct timespec64 ru_stime; /* system CPU time used */
    ...
}
```

struct **timespec64** is chosen over struct **timespec** to avoid 32-bit time_t overflow.

Problem

PID namespaces have been introduced without a proper kernel API to translate between tracer and tracee views of pids.

strace users are getting confused by PID namespaces: https://bugzilla.redhat.com/1035433

```
# strace -qq -ff -e clone -o s.log unshare --pid -- sh -c 'sh -c "sh -c true & wait" & wait'
# ls s.log.*
s.log.4567  s.log.4568  s.log.4569
# grep ^ s.log.*
s.log.4567:clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, ⟶
            child_tidptr=0x7fa7f9adba10) = 4568
s.log.4567:--- SIGCHLD si_signo=SIGCHLD, si_code=CLD_EXITED, ⟶
            si_pid=4568, si_uid=0, si_status=0, si_utime=0, si_stime=0 ---
s.log.4568:clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, ⟶
            child_tidptr=0x7fe0f586aa10) = 2
s.log.4568:--- SIGCHLD si_signo=SIGCHLD, si_code=CLD_EXITED, ⟶
            si_pid=2, si_uid=0, si_status=0, si_utime=0, si_stime=0 ---
```

Problem

PID namespaces have been introduced without a proper kernel API to translate between tracer and tracee views of pids.

strace users are getting confused by PID namespaces: https://bugzilla.redhat.com/1035433

```
# strace -qq -ff -e clone -o s.log unshare --pid -- sh -c 'sh -c "sh -c true & wait" & wait'
# ls s.log.*
s.log.4567  s.log.4568  s.log.4569
# grep ^ s.log.*
s.log.4567:clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, ⟶
           child_tidptr=0x7fa7f9adba10) = 4568
s.log.4567:--- SIGCHLD si_signo=SIGCHLD, si_code=CLD_EXITED, ⟶
           si_pid=4568, si_uid=0, si_status=0, si_utime=0, si_stime=0 ---
s.log.4568:clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, ⟶
           child_tidptr=0x7fe0f586aa10) = 2<4569>
s.log.4568:--- SIGCHLD si_signo=SIGCHLD, si_code=CLD_EXITED, ⟶
           si_pid=2<4569>, si_uid=0, si_status=0, si_utime=0, si_stime=0 ---
```

Add **translate_pid** syscall proposed by Konstantin Khlebnikov:
`https://lkml.org/lkml/2018/6/1/788`

```
pid_t translate_pid(pid_t pid, int source, int target);
```

pid namespaces are referred by file descriptors opened to proc files `/proc/[pid]/ns/pid` or `/proc/[pid]/ns/pid_for_children`.
Negative argument points to the current pid namespace.
Return value:
pid in the target pid namespace or zero if the task has no pid there.
Error codes:

- EBADF : source or target is not a valid open file descriptor
- EINVAL : file descriptor does not refer to a pid namespace
- ESRCH : task not found in the source namespace

Translation can breach pid namespace isolation and return pids from outer pid namespaces iff process already has file descriptor for these namespaces.

translate_pid examples provided by Konstantin Khlebnikov

```
translate_pid(pid, ns, -1)      - translate pid to our pid namespace
translate_pid(pid, -1, ns)      - translate pid to other pid namespace
translate_pid(pid, -1, ns) > 0  - is pid reachable from ns?
translate_pid(1, ns1, ns2) > 0  - is ns1 inside ns2?
translate_pid(1, ns1, ns2) == 0 - is ns1 outside ns2?
translate_pid(1, ns1, ns2) == 1 - is ns1 equal to ns2?
```

revision history

v1 : https://lkml.org/lkml/2015/9/15/411
v2 : https://lkml.org/lkml/2015/9/24/278
v3 : https://lkml.org/lkml/2015/9/25/290
v4 : https://lkml.org/lkml/2017/10/13/177
v5 : https://lkml.org/lkml/2018/4/4/677
v6 : https://lkml.org/lkml/2018/6/1/788

Problem

strace -yy chromium-browser doesn't show network protocol details because NETLINK_SOCK_DIAG does not report sockets of tracees running in different network namespaces:
https://lists.strace.io/pipermail/strace-devel/2018-September/008374.html

Example

Connected sockets should ve reported this way becase socketpair always generates a pair of connected sockets:

```
socketpair(AF_UNIX, SOCK_STREAM, 0, [27<UNIX:[7162769->7162770]>,
28<UNIX:[7162770->7162769]>]) = 0
```

If the tracee runs in a different network namespaces, the output generated by strace looks as if these sockets are unconnected:

```
socketpair(AF_UNIX, SOCK_STREAM, 0, [223<UNIX:[7162686]>,
224<UNIX:[7162687]>]) = 0
```

Problem

There are no consistent detailed declarative machine readable syscall descriptions, this forces every user to reinvent its own wheel and catch up with the kernel.

Current practice

- strace : A lot of manual work has been done to implement parsers of all syscalls in C, some of these parsers are quite complex, there is a test suite with 85% coverage.
- libc : Every libc has its own wrappers for some subset of syscalls, some of these wrappers are machine generated.
- syzkaller : Detailed declarative machine readable descriptions.
- others : Sanitizers, valgrind.

Proposed solution

Provide detailed declarative machine readable descriptions for all syscalls in the kernel.

hsh-run –mount=/proc –– strace -e trace=sendto,recvmsg ip route list

```
sendto(3, {{len=40, type=RTM_GETROUTE, flags=NLM_F_REQUEST|NLM_F_DUMP, seq=1357924680, pid=0}, {rtm_family=AF_UNSPEC,
rtm_dst_len=0, rtm_src_len=0, rtm_tos=0, rtm_table=RT_TABLE_UNSPEC, rtm_protocol=RTPROT_UNSPEC, rtm_scope=RT_SCOPE_UNIVERSE,
rtm_type=RTN_UNSPEC, rtm_flags=0}, {nla_len=0, nla_type=RTA_UNSPEC}}, 40, 0, NULL, 0) = 40
recvmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0, nl_groups=00000000}, msg_namelen=12, msg_iov=[{iov_base=[ {{len=60,
type=RTM_NEWROUTE, flags=NLM_F_MULTI, seq=1357924680, pid=12345}, {rtm_family=AF_INET, rtm_dst_len=32, rtm_src_len=0,
rtm_tos=0, rtm_table=RT_TABLE_LOCAL, rtm_protocol=RTPROT_KERNEL, rtm_scope=RT_SCOPE_LINK, rtm_type=RTN_BROADCAST,
rtm_flags=0}, [{{nla_len=8, nla_type=RTA_TABLE}, RT_TABLE_LOCAL}, {{nla_len=8, nla_type=RTA_DST}, inet_addr("127.0.0.0")},
{{nla_len=8, nla_type=RTA_PREFSRC}, inet_addr("127.0.0.1")}, {{nla_len=8, nla_type=RTA_OIF}, if_nametoindex("lo")}]},
{{len=60, type=RTM_NEWROUTE, flags=NLM_F_MULTI, seq=1357924680, pid=12345}, {rtm_family=AF_INET, rtm_dst_len=8,
rtm_src_len=0, rtm_tos=0, rtm_table=RT_TABLE_LOCAL, rtm_protocol=RTPROT_KERNEL, rtm_scope=RT_SCOPE_HOST,
rtm_type=RTN_LOCAL, rtm_flags=0}, [{{nla_len=8, nla_type=RTA_TABLE}, RT_TABLE_LOCAL}, {{nla_len=8, nla_type=RTA_DST},
inet_addr("127.0.0.0")}, {{nla_len=8, nla_type=RTA_PREFSRC}, inet_addr("127.0.0.1")}, {{nla_len=8, nla_type=RTA_OIF},
if_nametoindex("lo")}]}, {{len=60, type=RTM_NEWROUTE, flags=NLM_F_MULTI, seq=1357924680, pid=12345}, {rtm_family=AF_INET,
rtm_dst_len=32, rtm_src_len=0, rtm_tos=0, rtm_table=RT_TABLE_LOCAL, rtm_protocol=RTPROT_KERNEL, rtm_scope=RT_SCOPE_HOST,
rtm_type=RTN_LOCAL, rtm_flags=0}, [{{nla_len=8, nla_type=RTA_TABLE}, RT_TABLE_LOCAL}, {{nla_len=8, nla_type=RTA_DST},
inet_addr("127.0.0.1")}, {{nla_len=8, nla_type=RTA_PREFSRC}, inet_addr("127.0.0.1")}, {{nla_len=8, nla_type=RTA_OIF},
if_nametoindex("lo")}]}, {{len=60, type=RTM_NEWROUTE, flags=NLM_F_MULTI, seq=1357924680, pid=12345}, {rtm_family=AF_INET,
rtm_dst_len=32, rtm_src_len=0, rtm_tos=0, rtm_table=RT_TABLE_LOCAL, rtm_protocol=RTPROT_KERNEL, rtm_scope=RT_SCOPE_LINK,
rtm_type=RTN_BROADCAST, rtm_flags=0}, [{{nla_len=8, nla_type=RTA_TABLE}, RT_TABLE_LOCAL}, {{nla_len=8, nla_type=RTA_DST},
inet_addr("127.255.255.255")}, {{nla_len=8, nla_type=RTA_PREFSRC}, inet_addr("127.0.0.1")}, {{nla_len=8, nla_type=RTA_OIF},
if_nametoindex("lo")}]} ], iov_len=32768}], msg_iovlen=1, msg_controllen=0, msg_flags=0}, 0) = 240
...
```

strace/msghdr.c

```
SYS_FUNC(recvmsg) {
    int msg_namelen;
    if (entering(tcp)) {
        printfd(tcp, tcp->u_arg[0]);
        tprints(", ");
        if (fetch_msghdr_namelen(tcp, tcp->u_arg[1], &msg_namelen)) {
            set_tcb_priv_ulong(tcp, msg_namelen);
            return 0;
        }
        printaddr(tcp->u_arg[1]);
    } else {
        msg_namelen = get_tcb_priv_ulong(tcp);
        if (syserror(tcp))
            tprintf("msg_namelen=%d", msg_namelen);
        else
            decode_msghdr(tcp, &msg_namelen, tcp->u_arg[1], tcp->u_rval);
    }
    tprints(", ");
    printflags(msg_flags, tcp->u_arg[2], "MSG_???");
    return RVAL_DECODED;
}
```

syzkaller/sys/linux/socket.txt

```
recvmsg(fd sock, msg ptr[in, recv_msghdr], f flags[recv_flags])
...
recv_flags = MSG_CMSG_CLOEXEC, MSG_DONTWAIT, MSG_ERRQUEUE, MSG_OOB, MSG_PEEK,
MSG_TRUNC, MSG_WAITALL, MSG_WAITFORONE
...
recv_msghdr {
    msg_name ptr[out, sockaddr_storage, opt]
    msg_namelen len[msg_name, int32]
    msg_iov ptr[in, array[iovec_out]]
    msg_iovlen len[msg_iov, intptr]
    msg_control ptr[out, array[int8], opt]
    msg_controllen bytesize[msg_control, intptr]
    msg_flags int32
}
```

net/socket.c

```
SYSCALL_DEFINE3(recvmsg, int, fd, struct user_msghdr __user *, msg, unsigned int, flags)
{
    return __sys_recvmsg(fd, msg, flags, true);
}
```

ptrace API is slow

There are two syscall stops per syscall: **syscall-enter-stop** and **syscall-exit-stop**.
There are two context switches per syscall stop: from tracee to tracer and back.
strace invokes at least three syscalls per syscall stop:
wait4, PTRACE_GETREGSET, and PTRACE_SYSCALL.

kernel tracing can lose data

The data is written to a ring buffer and could be lost if the reader is not fast enough.

Ideas

- Add a flag to struct perf_event_attr that new perf events should block on overflow
- Implement a perf backend for strace
- Compile strace decoders into eBPF

Questions?

homepage

https://strace.io

strace.git

https://github.com/strace/strace.git
https://gitlab.com/strace/strace.git

mailing list

strace-devel@lists.strace.io

IRC channel

#strace@freenode