Next Previous Contents

5. Signals

The common communication channel between user space program and kernel is given by the system calls. But there is a different channel, that of the *signals*, used both between user processes and from kernel to user process.

5.1 Sending signals

A program can signal a different program using the `kill()` system call with prototype

```
int kill(pid_t pid, int sig);
```

This will send the signal with number `sig` to the process with process ID `pid`. Signal numbers are small positive integers. (For the definitions on your machine, try `/usr/include/bits/signum.h`. Note that these definitions depend on OS and architecture.)

A user can send a signal from the command line using the `kill` command. Common uses are `kill -9` *N* to kill the process with pid *N*, or `kill -1` *N* to force process *N* (maybe `init` or `inetd`) to reread its configuration file.

Certain user actions will make the kernel send a signal to a process or group of processes: typing the interrupt character (probably Ctrl-C) causes SIGINT to be sent, typing the quit character (probably Ctrl-\) sends SIGQUIT, hanging up the phone (modem) sends SIGHUP, typing the stop character (probably Ctrl-Z) sends SIGSTOP.

Certain program actions will make the kernel send a signal to that process: for an illegal instruction one gets SIGILL, for accessing nonexisting memory one gets SIGSEGV, for writing to a pipe while nobody is listening anymore on the other side one gets SIGPIPE, for reading from the termnal while in the background one gets SIGTTIN, etc.

More interesting communication from the kernel is also possible. One can ask the kernel to be notified when something happens on a given file descriptor. See fcntl(2). And then there is ptrace(2) - see below.

A whole group of signals is reserved for real-time use.

5.2 Receiving signals

When a process receives a signal, a default action happens, unless the process has arranged to handle the signal. For the list of signals and the corresponding default actions, see signal(7). For example, by default SIGHUP, SIGINT, SIGKILL will kill the process; SIGQUIT will kill

the process and force a core dump; SIGSTOP, SIGTTIN will stop the process; SIGCONT will continue a stopped process; SIGCHLD will be ignored.

Traditionally, one sets up a handler for the signal using the `signal` system call with prototype

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler);
```

This sets up the routine `handler()` as handler for signals with number `sig`. The return value is (the address of) the old handler. The special values SIG_DFL and SIG_IGN denote the default action and ignoring, respectively.

When a signal arrives, the process is interrupted, the current registers are saved, and the signal handler is invoked. When the signal handler returns, the interrupted activity is continued.

It is difficult to do interesting things in a signal handler, because the process can be interrupted in an arbitrary place, data structures can be in arbitrary state, etc. The three most common things to do in a signal handler are (i) set a flag variable and return immediately, and (ii) (messy) throw away all the program was doing, and restart at some convenient point, perhaps the main command loop or so, and (iii) clean up and exit.

Setting up a handler for a signal is called "catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or blocked or ignored.

5.3 Semantics

The traditional semantics was: reset signal behaviour to SIG_DFL upon invocation of the signal handler. Possibly this was done to avoid recursive invocations. The signal handler would do its job and at the end call `signal()` to establish itself again as handler.

This is really unfortunate. When two signals arrive shortly after each other, the second one will be lost if it arrives before the signal handler is called - there is no counter. And if it arrives after the signal handler is called, the default action will happen - this may very well kill the process. Even if the handler calls `signal()` again as the very first thing it does, that may be too late.

Various Unix flavours played a bit with the semantics to improve on this situation. Some block signals as long as the process has not returned from the handler. The BSD solution was to invent a new system call, `sigaction()` where one can precisely specify the desired behaviour. Today `signal()` must be regarded as deprecated - not to be used in serious applications.

5.4 Blocking signals

Each process has a list (bitmask) of currently blocked signals. When a signal is blocked, it is not delivered (that is, no signal handling routine is called), but remains pending.

The `sigprocmask()` system call serves to change the list of blocked signals. See sigprocmask(2).

The `sigpending()` system call reveals what signals are (blocked and) pending.

The `sigsuspend()` system call suspends the calling process until a specified signal is received.

When a signal is blocked, it remains pending, even when otherwise the process would ignore it.

5.5 Voodoo: wait and SIGCHLD

When a process forks off a child to perform some task, it is probably interested in how things went. Upon exit, the child leaves an exit status that should be returned to the parent. So, when the child finishes it becomes a *zombie* - a process that is dead already but does not disappear yet because it has not yet reported its exit status.

Whenever something interesting happens to the child (it exits, crashes, traps, stops, continues), and in particular when it dies, the parent is sent a SIGCHLD signal.

The parent can use the system call `wait()` or `waitpid()` or so, there are a few variations, to learn about the status of its stopped or deceased children. In the case of a deceased child, as soon as a status has been reported, the zombie vanishes.

If the parent is not interested it can say so explicitly (before the fork) using

```
signal(SIGCHLD, SIG_IGN);
```

or

```
struct sigaction act;
act.sa_handler = something;
act.sa_flags = SA_NOCLDWAIT;
sigaction (SIGCHLD, &act, NULL);
```

and as a result it will not hear about deceased children, and children will not be transformed into zombies. Note that the default action for SIGCHLD is to ignore this signal; nevertheless `signal(SIGCHLD, SIG_IGN)` has effect, namely that of preventing the transformation of children into zombies. In this situation, if the parent does a `wait()`, this call will return only when all children have exited, and then returns -1 with *errno* set to ECHILD.

It depends on the Unix flavor whether SIGCHLD is sent when SA_NOCLDWAIT was set. After `act.sa_flags = SA_NOCLDSTOP` no SIGCHLD is sent when children stop or stopped children continue.

If the parent exits before the child, then the child is reparented to `init`, process 1, and this process will reap its status.

5.6 Returning from a signal handler

When the program was interrupted by a signal, its status (including all integer and floating point registers) was saved, to be restored just before execution continues at the point of interruption.

This means that the return from the signal handler is more complicated than an arbitrary procedure return - the saved state must be restored.

To this end, the kernel arranges that the return from the signal handler causes a jump to a short code sequence (sometimes called *trampoline*) that executes a `sigreturn()` system call. This system call takes care of everything.

In the old days the trampoline lived on the stack, but nowadays (since 2.5.69) we have a trampoline in the vsyscall page, so that this trampoline no longer is an obstacle in case one wants a non-executable stack.

5.7 ptrace

For debugging purposes, the `ptrace()` system call was introduced. A process can trace a different process, examine or change its memory, see the system calls done or change them, etc. The way this is implemented is that the tracing process is notified each time the traced process does something interesting. Always interesting is the reception of signals. When the tracing process specifies this, also excution of system calls, or execution of any instruction is interesting.

Thus, one has interactive debuggers like `gdb`, and tracers like `strace`. This call is also very useful for hacking purposes. One can make one's own program attach to some utility and subtly change its workings, while the binary of the utility is unchanged, still has the correct date stamps and md5sum.

Exercise *Write a program that attaches itself to a process with specified pid, and watches its read() calls; whenever a specified string occurs, change it into something else. Be otherwise invisible.*

Exercise *Write a program that attaches itself to a shell and watches the commands given. Whenever a specified binary must be executed, execute a different specified binary instead. Be otherwise invisible.*

Below a baby example of the use of ptrace. This program will list the system calls done by some existing process (call `./ptrace -p` *N*) or some subprocess (call `./ptrace some_command`, e.g., `./ptrace /bin/ls -l`). The version below will work only on i386, and only for relatively recent kernels.

```
/*
 * ptrace a child - baby demo example - i386 only
 *
 * call: "ptrace command args" to trace command
 *       "ptrace -p N" to trace process with pid N
 */
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/ptrace.h>
#include <linux/user.h>      /* user_regs_struct */
#include <linux/unistd.h>    /* __NR_exit */
#include <linux/ptrace.h>    /* EAX */

#define SIZE(a) (sizeof(a)/sizeof((a)[0]))

/*
 * syscall names for i386 under 2.5.51, taken from <asm/unistd.h>
 */
char *(syscall_names[256]) = {
"exit", "fork", "read", "write", "open", "close", "waitpid", "creat",
"link", "unlink", "execve", "chdir", "time", "mknod", "chmod",
"lchown", "break", "oldstat", "lseek", "getpid", "mount", "umount",
"setuid", "getuid", "stime", "ptrace", "alarm", "oldfstat", "pause",
"utime", "stty", "gtty", "access", "nice", "ftime", "sync", "kill",
"rename", "mkdir", "rmdir", "dup", "pipe", "times", "prof", "brk",
"setgid", "getgid", "signal", "geteuid", "getegid", "acct", "umount2",
"lock", "ioctl", "fcntl", "mpx", "setpgid", "ulimit", "oldolduname",
"umask", "chroot", "ustat", "dup2", "getppid", "getpgrp", "setsid",
"sigaction", "sgetmask", "ssetmask", "setreuid", "setregid",
"sigsuspend", "sigpending", "sethostname", "setrlimit", "getrlimit",
"getrusage", "gettimeofday", "settimeofday", "getgroups", "setgroups",
"select", "symlink", "oldlstat", "readlink", "uselib", "swapon",
"reboot", "readdir", "mmap", "munmap", "truncate", "ftruncate",
"fchmod", "fchown", "getpriority", "setpriority", "profil", "statfs",
"fstatfs", "ioperm", "socketcall", "syslog", "setitimer", "getitimer",
"stat", "lstat", "fstat", "olduname", "iopl", "vhangup", "idle",
"vm86old", "wait4", "swapoff", "sysinfo", "ipc", "fsync", "sigreturn",
"clone", "setdomainname", "uname", "modify_ldt", "adjtimex",
"mprotect", "sigprocmask", "create_module", "init_module",
"delete_module", "get_kernel_syms", "quotactl", "getpgid", "fchdir",
"bdflush", "sysfs", "personality", "afs_syscall", "setfsuid",
"setfsgid", "_llseek", "getdents", "_newselect", "flock", "msync",
"readv", "writev", "getsid", "fdatasync", "_sysctl", "mlock",
"munlock", "mlockall", "munlockall", "sched_setparam",
"sched_getparam", "sched_setscheduler", "sched_getscheduler",
"sched_yield", "sched_get_priority_max", "sched_get_priority_min",
"sched_rr_get_interval", "nanosleep", "mremap", "setresuid",
"getresuid", "vm86", "query_module", "poll", "nfsservctl",
```

```
"setresgid", "getresgid", "prctl","rt_sigreturn","rt_sigaction",
"rt_sigprocmask", "rt_sigpending", "rt_sigtimedwait",
"rt_sigqueueinfo", "rt_sigsuspend", "pread", "pwrite", "chown",
"getcwd", "capget", "capset", "sigaltstack", "sendfile", "getpmsg",
"putpmsg", "vfork", "ugetrlimit", "mmap2", "truncate64",
"ftruncate64", "stat64", "lstat64", "fstat64", "lchown32", "getuid32",
"getgid32", "geteuid32", "getegid32", "setreuid32", "setregid32",
"getgroups32", "setgroups32", "fchown32", "setresuid32",
"getresuid32", "setresgid32", "getresgid32", "chown32", "setuid32",
"setgid32", "setfsuid32", "setfsgid32", "pivot_root", "mincore",
"madvise", "getdents64", "fcntl64", 0, "security", "gettid",
"readahead", "setxattr", "lsetxattr", "fsetxattr", "getxattr",
"lgetxattr", "fgetxattr", "listxattr", "llistxattr", "flistxattr",
"removexattr", "lremovexattr", "fremovexattr", "tkill", "sendfile64",
"futex", "sched_setaffinity", "sched_getaffinity",
};

void my_ptrace_void(int request, pid_t pid, void *addr, void *data) {
        int i = ptrace(request, pid, addr, data);
        if (i) {
                perror("ptrace");
                exit(1);
        }
}

/*
 * Since -1 may be valid data, we have to check errno.
 */
int my_ptrace_read(int request, pid_t pid, void *addr, void *data) {
        int i;

        errno = 0;
        i = ptrace(request, pid, addr, data);
        if (i == -1 && errno) {
                perror("ptrace");
                exit(1);
        }
        return i;
}

pid_t pid;      /* the traced program */

/* detach from traced program when interrupted */
void interrupt(int dummy) {
        ptrace(PTRACE_DETACH, pid, 0, 0);
        exit(-1);
}

int got_sig = 0;

void sigusr1(int dummy) {
        got_sig = 1;
}
```

```
void my_kill(pid_t pid, int sig) {
        int i = kill(pid, sig);
        if (i) {
                perror("kill");
                exit(1);
        }
}

/*
 * A child stopped at a syscall has status as if it received SIGTRAP.
 * In order to distinguish between SIGTRAP and syscall, some kernel
 * versions have the PTRACE_O_TRACESYSGOOD option, that sets an extra
 * bit 0x80 in the syscall case.
 */
#define SIGSYSTRAP      (SIGTRAP | sysgood_bit)

int sysgood_bit = 0;

void set_sysgood(pid_t p) {
#ifdef PTRACE_O_TRACESYSGOOD
        int i = ptrace(PTRACE_SETOPTIONS, p, 0, (void*) PTRACE_O_TRACESYSGOOD);
        if (i == 0)
                sysgood_bit = 0x80;
        else
                perror("PTRACE_O_TRACESYSGOOD");
#endif
}

#define EXPECT_EXITED   1
#define EXPECT_SIGNALED 2
#define EXPECT_STOPPED  4

void my_wait(pid_t p, int report, int stopsig) {
        int status;
        pid_t pw = wait(&status);

        if (pw == (pid_t) -1) {
                perror("wait");
                exit(1);
        }

        /*
         * Report only unexpected things.
         *
         * The conditions WIFEXITED, WIFSIGNALED, WIFSTOPPED
         * are mutually exclusive:
         * WIFEXITED:  (status & 0x7f) == 0, WEXITSTATUS: top 8 bits
         * and now WCOREDUMP:  (status & 0x80) != 0
         * WIFSTOPPED: (status & 0xff) == 0x7f, WSTOPSIG: top 8 bits
         * WIFSIGNALED: all other cases, (status & 0x7f) is signal.
         */
        if (WIFEXITED(status) && !(report & EXPECT_EXITED))
                fprintf(stderr, "child exited%s with status %d\n",
                        WCOREDUMP(status) ? " and dumped core" : "",
```

```
                        WEXITSTATUS(status));
        if (WIFSTOPPED(status) && !(report & EXPECT_STOPPED))
                fprintf(stderr, "child stopped by signal %d\n",
                        WSTOPSIG(status));
        if (WIFSIGNALED(status) && !(report & EXPECT_SIGNALED))
                fprintf(stderr, "child signalled by signal %d\n",
                        WTERMSIG(status));

        if (WIFSTOPPED(status) && WSTOPSIG(status) != stopsig) {
                /* a different signal - send it on and wait */
                fprintf(stderr, "Waited for signal %d, got %d\n",
                        stopsig, WSTOPSIG(status));
                if ((WSTOPSIG(status) & 0x7f) == (stopsig & 0x7f))
                        return;
                my_ptrace_void(PTRACE_SYSCALL, p, 0, (void*) WSTOPSIG(status));
                return my_wait(p, report, stopsig);
        }

        if ((report & EXPECT_STOPPED) && !WIFSTOPPED(status)) {
                fprintf(stderr, "Not stopped?\n");
                exit(1);
        }
}

/*
 * print value when changed
 */
void outlonghex(unsigned long old, unsigned long new) {
        if (old == new)
                fprintf(stderr, "         ");
        else
                fprintf(stderr, " %08lx", new);
}

int
main(int argc, char **argv, char **envp){
        pid_t p0, p;

        if (argc <= 1) {
                fprintf(stderr, "Usage: %s command args -or- %s -p pid\n",
                        argv[0], argv[0]);
                exit(1);
        }

        if (argc >= 3 && !strcmp(argv[1], "-p")) {
                pid = p = atoi(argv[2]);

                signal(SIGINT, interrupt);

                /*
                 * attach to specified process
                 */
                my_ptrace_void(PTRACE_ATTACH, p, 0, 0);
                my_wait(p, EXPECT_STOPPED, SIGSTOP);
```

```
                set_sysgood(p);

                /*
                 * we stopped the program in the middle of what it was doing
                 * continue it, and make it stop at the next syscall
                 */
                my_ptrace_void(PTRACE_SYSCALL, p, 0, 0);
        } else {
                void (*oldsig)(int);

                /*
                 * fork off a child that executes the specified command
                 */

                /*
                 * The parent process will send a signal to the child
                 * and do a wait() to wait until the child stops.
                 * If the signal arrives before the child has said
                 * PTRACE_TRACEME, then maybe the child is killed, or
                 * maybe the signal is ignored and we wait forever, or
                 * maybe the child is stopped but we are not tracing.
                 * So, let us arrange for the child to signal the parent
                 * when it has done the PTRACE_TRACEME.
                 */

                /* prepare both parent and child for signal */
                oldsig = signal(SIGUSR1, sigusr1);
                if (oldsig == SIG_ERR) {
                        perror("signal");
                        exit(1);
                }

                /* child needs parent pid */
                p0 = getpid();

                p = fork();
                if (p == (pid_t) -1) {
                        perror("fork");
                        exit(1);
                }

                if (p == 0) {   /* child */
                        my_ptrace_void(PTRACE_TRACEME, 0, 0, 0);

                        /* tell parent that we are ready */
                        my_kill(p0, SIGUSR1);

                        /* wait for parent to start tracing us */
                        while (!got_sig) ;

                        /*
                         * the first thing the parent will see is
                         *  119: sigreturn - the return from the signal handler
                         */
```

```
                /* exec the given process */
                argv[argc] = 0;
                execve(argv[1], argv+1, envp);
                _exit(1);
        }

        /* wait for child to get ready */
        while (!got_sig) ;

        /*
         * tell child that we got the signal
         * this kill() will stop the child
         */
        my_kill(p, SIGUSR1);
        my_wait(p, EXPECT_STOPPED, SIGUSR1);
        set_sysgood(p);

        my_ptrace_void(PTRACE_SYSCALL, p, 0, (void *) SIGUSR1);
}

/*
 * trace the victim's syscalls
 */
while (1) {
        int syscall;
        struct user_regs_struct u_in, u_out;

        my_wait(p, EXPECT_STOPPED, SIGSYSTRAP);

        my_ptrace_void(PTRACE_GETREGS, p, 0, &u_in);
        syscall = u_in.orig_eax;

        fprintf(stderr, "SYSCALL %3d:", syscall);
        outlonghex(-38, u_in.eax);      /* seems constant */
        fprintf(stderr, " %08lx %08lx %08lx",
                u_in.ebx, u_in.ecx, u_in.edx);
        if (syscall-1 >= 0 && syscall-1 < SIZE(syscall_names) &&
            syscall_names[syscall-1])
                fprintf(stderr, " /%s", syscall_names[syscall-1]);
        fprintf(stderr, "\n");

        if (syscall == __NR_execve) {
                long *regs = 0; /* relative address 0 in user area */
                long eax;

                my_ptrace_void(PTRACE_SYSCALL, p, 0, 0);
                my_wait(p, EXPECT_STOPPED, SIGSYSTRAP);

                /*
                 * For a successful execve we get one more trap
                 * But was this call successful?
                 */
                eax = my_ptrace_read(PTRACE_PEEKUSER, p, &regs[EAX],0);
```

```
                        if (eax == 0) {
                                fprintf(stderr, "SYSCALL execve, once more\n");

                                /* the syscall return - no "good" bit */
                                my_ptrace_void(PTRACE_SYSCALL, p, 0, 0);
                                my_wait(p, EXPECT_STOPPED, SIGTRAP);
                        }
                } else {
                        /* wait for syscall return */
                        my_ptrace_void(PTRACE_SYSCALL, p, 0, 0);
                        if (syscall == __NR_exit ||
                            syscall == __NR_exit_group) {
                                my_wait(p, EXPECT_EXITED, 0);
                                exit(0);
                        }
                        my_wait(p, EXPECT_STOPPED, SIGSYSTRAP);
                }

                my_ptrace_void(PTRACE_GETREGS, p, 0, &u_out);
                fprintf(stderr, " RETURN %3d:", syscall);
                outlonghex(u_in.eax, u_out.eax);
                outlonghex(u_in.ebx, u_out.ebx);
                outlonghex(u_in.ecx, u_out.ecx);
                outlonghex(u_in.edx, u_out.edx);
                fprintf(stderr, "\n");

                my_ptrace_void(PTRACE_SYSCALL, p, 0, 0);
        }
        return 0;
}
```

A tracee gets sent a SIGTRAP signal (and stops) when it successfully does an `execve` system call. This means that we see `execve` three times (on exit, on return, on SIGTRAP). This signal can be used to synchronize on, instead of the above complicated SIGUSR1 construction.

5.8 The Linux "parent death" signal

For each process there is a variable `pdeath_signal`, that is initialized to 0 after fork() or clone(). It gives the signal that the process should get when its parent dies.

This variable can be set using

```
prctl(PR_SET_PDEATHSIG, sig);
```

and read out using

```
prctl(PR_GET_PDEATHSIG, &sig);
```

This construct can be used in thread libraries: when the manager thread dies, all threads managed by it should clean up and exit. E.g.:

```
        prctl(PR_SET_PDEATHSIG, SIGHUP);        /* set pdeath sig */
        if (getppid() == 1)                     /* parent died already? */
                kill(getpid(), SIGHUP);
```

Next Previous Contents