Advanced Unix System Administration
Spring 2007
Homework 2 Solutions

1. *Becoming a daemon.* In principle, this is just the job of getting into the background and out of the user's way. In practice, getting completely untangled from the foreground is more difficult than you might expect.

   a. You've probably noticed by now that forking a process puts it in the background. Examine, compile and run `daemon1.c`, which does just that. Follow its execution using `strace`. What does the program do? What happens when you try to log out? Why? (Hint: look at the files `daemon1` has open.)

      `daemon1` prints out its PID, forks a child (which is in the background) and exits, but its child spits out messages like this every five seconds:

      ```
      21463: foobar
      ```

      When you try to log out, the logout hangs, and the printing of the messages continues.

      In terms of going into the background, the interesting portion of the `strace` output is fairly short:

      ```
      21550 clone(child_stack=0,
          flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
          child_tidptr=0x2afd48699760) = 21551
      21550 exit_group(0)                     = ?
      ```

      (The `clone()` system call is used in Linux to create both threads and processes; `fork()` is a wrapper around `clone()`.)

      Looking at the output from `lsof` shows some interesting files open:

      ```
      daemon1 21463 sluo    0u   CHR  136,1                 3 /dev/pts/1
      daemon1 21463 sluo    1u   CHR  136,1                 3 /dev/pts/1
      daemon1 21463 sluo    2u   CHR  136,1                 3 /dev/pts/1
      ```

      The child has our tty open, which is why its messages continue to appear in our terminal. The logout from an `ssh` session hangs because it's waiting for `daemon1` to close the tty.

   b. Compare `daemon2.c` to the previous version. What has changed? Now compile and run it. Follow its execution using `strace` (or `truss`, or `ktrace`, or whatever equivalent your OS has). What does this program do? Does it solve the problems of the previous version? Look at the list of files it has open. Do you think we've successfully solved the problem?

daemon2 closes file descriptors 0, 1, and 2 (which correspond to standard I/O) and opens /dev/null in their place. The relevant part of the strace output now looks like this:

```
21650 close(0)                          = 0
21650 close(1)                          = 0
21650 close(2)                          = 0
21650 clone(child_stack=0,
    flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
    child_tidptr=0x2b1e6e8fc760) = 21651
21650 exit_group(0)                     = ?
21651 open("/dev/null", O_RDONLY)       = 0
21651 open("/dev/null", O_WRONLY)       = 1
21651 open("/dev/null", O_WRONLY)       = 2
```

(Note the effect of the open() calls is to replace file descriptors 0, 1, and 2.) This version solves the most obvious problems of the previous version (spitting out junk to the terminal and hanging the session on logout), but the list of open files shows at least one possible subtle problem:

```
daemon2 21693 sluo  cwd    DIR   22,0    4096 4049554
    /home/sluo/hw2
```

The current working directory is the one we launched the program from; this implies that this directory will be kept around on disk, even if deleted, until daemon2 dies. Hence we're not completely done here.

c. Try killing the parent's process group (kill -TERM -[pgid], where the process group ID will be equal to the process ID of the parent). What happens? Have we successfully solved the problem?

   Recall that the parent nicely gave us its PID on standard output before forking its child; this PID also serves as the PGID for all of its children. Killing the process group with kill -TERM -[pgid] ends up killing the child (despite the fact that the parent process with that particular PGID is no longer around):

```
sluo@adv-login:~/hw2$ ./daemon2
Parent: pid 21700
sluo@adv-login:~/hw2$ pgrep daemon2
21701
sluo@adv-login:~/hw2$ kill -TERM -21700
sluo@adv-login:~/hw2$ pgrep daemon2
sluo@adv-login:~/hw2$
```

   So we have not disentangled ourselves from the parent's process group, which could be a potential problem if we were spawned from something other than a shell (i.e. from some manager process managing daemons). (As it turns out, POSIX requires that PIDs not be reused until there are no processes in the process group corresponding to that ID, so we are also holding up a PID.) In other words, we still have some work to do.

d. Compare `daemon3.c` to the previous version. Now what has changed? Compile and run it; follow its execution using `strace`. What does this program do? Does it solve the problems of the previous version? Look at the list of files it has open. Do you think we've successfully solved the problem?

   `daemon3`'s child calls `setsid()` to change its session ID and become session leader, ensures that it has no controlling terminal, changes to the root directory, and changes the umask. (There was an error in the code I gave you for the homework, which caused the program to exit after checking for a controlling tty.) The relevant part of the `strace` output now looks like this:

```
21764 close(0)                         = 0
21764 close(1)                         = 0
21764 close(2)                         = 0
21764 clone(child_stack=0,
    flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
    child_tidptr=0x2ba5be880760) = 21765
21764 munmap(0x2ba5be544000, 4096)     = 0
21764 exit_group(0)                    = ?
21765 setsid()                         = 21765
21765 open("/dev/null", O_RDONLY)      = 0
21765 open("/dev/null", O_WRONLY)      = 1
21765 open("/dev/null", O_WRONLY)      = 2
21765 open("/dev/tty", O_RDWR|O_NOCTTY) = -1 ENXIO
    (No such device or address)
21765 chdir("/")                       = 0
21765 umask(022)                       = 022
```

   (Your output would have looked more like this:

```
open("/dev/tty", O_RDWR|O_NOCTTY)      = -1 ENXIO
    (No such device or address)
ioctl(4294967295, TIOCNOTTY)           = -1 EBADF
    (Bad file descriptor)
exit_group(1)                          = ?
```

   This shows the bug quite clearly – I didn't check for the error return from the `open()` call correctly, and the program thus tries to detach from a nonexistent controlling tty.)

   The `lsof` output shows a working directory of `/`, and attempting to kill the parent's process group gives us an error of "No such process", so we appear to have solved the problems of previous versions. Indeed, `daemon3` does do enough to really get itself into the background. (Who knew it could be so complicated?)

e. Set up OpenSSH's `sshd` to run as yourself on a high port. Follow its execution using `strace`. How does `sshd` get itself into the background? Does this have any advantages over what `daemon3` does?

   The easiest way to do this was to generate new hostkeys with `ssh-keygen`, create a copy of `/etc/ssh/sshd_config` and modify it to use the new hostkeys and a different port, and then start `sshd` with the `-f` option to specify the use of the new config file.

   The relevant part of the `strace` output is this:

```
21905 clone(child_stack=0,
    flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
    child_tidptr=0x2b598ef60950) = 21906
21906 setsid( <unfinished ...>
21905 exit_group(0)                      = ?
21906 <... setsid resumed> )             = 21906
21906 chdir("/")                         = 0
21906 open("/dev/null", O_RDWR)          = 3
21906 fstat(3, {st_mode=S_IFCHR|0666,
    st_rdev=makedev(1, 3), ...}) = 0
21906 dup2(3, 0)                         = 0
21906 dup2(3, 1)                         = 1
21906 dup2(3, 2)                         = 2
21906 close(3)                           = 0
21906 open("/dev/tty", O_RDWR|O_NOCTTY) = -1 ENXIO
    (No such device or address)
21906 chdir("/")                         = 0
```

   This is quite similar to what `daemon3` does, except that `dup2()` is used to replace standard I/O with `/dev/null` instead of `close()` and `open()`, and setting the umask is not done. `dup2()` makes the file descriptor that is its second argument a copy of its first argument; this is a more surefire way of ensuring that the standard I/O file descriptors are replaced than the `close()`/`open()` sequence. Setting the umask is considered good practice, since it ensures that files created by the daemon will have sane permissions, but not doing it has no consequences here, since the `sshd` child doesn't create any files.

f. Look up what sessions and session leaders are. Why might you want to fork twice when starting a daemon? Hint: this has something to do with the process's controlling terminal.

   The Single Unix Specification version 3 (SUSv3) defines a "session" as (Base Definitions volume, §3.337)

A collection of process groups established for job control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see `setsid()`. There can be multiple process groups in the same session.

§3.338 defines a session leader as "A process that has created a session".

The documentation for `setsid()` (System Interfaces volume, §3) tells us that "The `setsid()` function shall create a new session, if the calling process is not a process group leader." This explains calling `fork()` before `setsid()`; by forking, we ensure that we are not the process group leader. We want to call `setsid()` because "Upon return the calling process shall be the session leader of this new session, shall be the process group leader of a new process group, and shall have no controlling terminal"; this solves a whole lot of problems we noticed earlier.

The definition of the POSIX terminal interface (Base Definitions volume, §11) tells us why a daemon writer might care about sessions; in particular, §11.1.3 says

> If a session leader has no controlling terminal, and opens a terminal device file that is not already associated with a session without using the `O_NOCTTY` option (see `open()`), it is implementation-defined whether the terminal becomes the controlling terminal of the session leader. If a process which is not a session leader opens a terminal file, or the `O_NOCTTY` option is used on `open()`, then that terminal shall not become the controlling terminal of the calling process.

which explains why we might want to fork twice. Forking after calling `setsid()` ensures that we are not a session leader, which makes it much more difficult to accidentally acquire a controlling tty (which is a problem for a daemon that's supposed to be in the background, not attached to any terminals).

The first parts of §11, and the documentation for `setsid()` (whether the SUSv3 definition or a man page from elsewhere) make for interesting reading in the context of this homework assignment, in that they explain much of the dance we do to get rid of our controlling terminal.

2. *Tracing a running process. This exercise must be done on the login server.* Among the files for this week's assignment is `wrapper`, which forks off a child process to perform some tasks.

   a. Run this program and observe its behavior. Now try tracing it with `strace`. Does the program do the same thing? If not, why not?

      We get the following output:

```
sh-3.1$ ./wrapper
Forked background process, pid 22187
```

This doesn't tell us much. Tracing the program doesn't change the behavior, thanks to my braindamaged attempt at using Unix file permissions; this in fact allows you to do the later parts of the problem much more easily than I intended.

A fixed version (`wrapper2`) behaves the same way when not traced, but gives us the following when traced:

```
sh-3.1$ strace -f -o /tmp/strace.out ./wrapper2
Incorrect gid -- go away
```

Why? A look at the binary tells us that the program is setgid:

```
-rwx--s--x 1 sluo sluo 9812 2007-02-26 03:37
    /home/sluo/hw2-files/wrapper2
```

You cannot trace a setuid or setgid program for security reasons; the kernel ignores the setuid or setgid bit when executing the program, to prevent you from seeing information you shouldn't.

b. Attach to the child process using `strace`. Describe in detail what the program is doing or trying to do, and the errors it is getting.

The process seems to be doing this over and over again:

```
lseek(3, 256, SEEK_SET)                  = 256
read(3, "\275\321\rM\23\26)"..., 127)   = 127
open("/etc/shadow", O_RDONLY)            = -1 EACCES
    (Permission denied)
open("/home/sluo/foobarbaz", O_RDONLY)  = -1 ENOENT
    (No such file or directory)
lseek(4, 64, SEEK_SET)                   = -1 ESPIPE
    (Illegal seek)
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({5, 0}, {5, 0})                = 0
```

It's seeking to byte 256 on file descriptor 3 (whatever that is), and reading 127 bytes from that file. It then attempts to open `/etc/shadow` for reading, which fails with "Permission denied", since `/etc/shadow` isn't readable by ordinary users. It then tries to open `/home/sluo/foobarbaz` for reading, which fails because the file doesn't exist. It then tries to seek to byte 64 on file descriptor 4, which fails because file descriptor 4 is a named pipe and seeking on pipes isn't permitted. After that, it sleeps for 5 seconds, and repeats.

c. What files does the process have open? Identify the file descriptors that were referred to in the `strace` output.

```
read    25147 nobody    0u   CHR  136,1              3
    /dev/pts/1
read    25147 nobody    1u   CHR  136,1              3
    /dev/pts/1
read    25147 nobody    2u   CHR  136,1              3
    /dev/pts/1
read    25147 nobody    3r   REG   22,0    1024 3915954
    /home/sluo/entropy
read    25147 nobody    4r  FIFO   22,0         3915953
    /home/sluo/fifo
```

The process has our tty open on file descriptors 0, 1, and 2. File descriptor 3 is `/home/sluo/entropy`, which is 1024 bytes of `/dev/urandom`, while file descriptor 4 is `/home/sluo/fifo`, a named pipe (as expected).

You could also have obtained this information directly via `strace` of the parent process, because of the permissions screwup.

d. What did `wrapper`'s child process do after the call to `fork()`? (This might take a bit of thought, but you do have enough information to answer this question, assuming you already did the first three parts of the problem.)

```
read    25147 nobody  txt    REG   22,0    9324 4030772
    /home/sluo/hw2-binaries/read
```

"txt" here refers to the code of the program being run. Hence we conclude that `wrapper`'s child process executed `/home/sluo/hw2-binaries/read`.

Because of the permissions screwup, you could also have gotten this information directly from `strace`.

e. *Optional; might take a bit of thought.* What do you think `wrapper` does before the call to `fork()`?

Notice that the child process runs with your own GID (try `ps -g yourgroup`). We know the parent is setgid; hence at some point it must have changed its GID. (The call in this case is to `setegid()`, which changes the effective GID; we'll talk much more about this when we get to security.)

Again, because of the permissions screwup, you could have gotten this information directly from `strace`.