Advanced Unix System Administration
Fall 2008
Homework 1 Solutions

1. *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.

   When tracing the program, though, we get the following output:

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

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

   ```
   -rwxr-sr-x 1 sluo sluo 10269 2008-09-30 21:25 wrapper
   ```

   Recall that a setgid program will be run as the group owning the binary, not as the group of the user running the binary. 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:

   ```
   sh-3.1$ strace -p 30719
   [snip]
   lseek(3, 256, SEEK_SET)                 = 256
   read(3, "\10\f?\37\214\3["..., 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, 256, 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
[snip]
```

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.

Notice how `strace` is nice enough to show you what pointers passed as arguments to syscalls point to, and to print out error numbers (and their descriptions!) when errors occur. This is unfortunately more than can be said for some of the other tracing tools you'll come across.

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

Using `lsof`, we see the following at the bottom of the output:

```
sh-3.1$ lsof -p 30719
[snip]
read    30719 nobody    0u   CHR  136,0              2
    /dev/pts/0
read    30719 nobody    1u   CHR  136,0              2
    /dev/pts/0
read    30719 nobody    2u   CHR  136,0              2
    /dev/pts/0
read    30719 nobody    3r   REG 202,17    1024  118075
    /home/sluo/entropy
read    30719 nobody    4r  FIFO 202,17          118076
    /home/sluo/fifo
```

The process has our tty open on file descriptors 0, 1, and 2. File descriptor 3 is `/home/sluo/entropy` (which I made by taking 1024 bytes out of

`/dev/urandom`), while file descriptor 4 is `/home/sluo/fifo`, a named pipe (as expected).

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

Above the files in the `lsof` output, we see the following:

```
read    30719 nobody  txt    REG 202,17    9344  118073
    /home/sluo/hw1-files/private/read
```

From the `lsof` man page, we see that the `txt` under the FD field means "program text (code and data)"; in other words, this is the file containing the code of the running program. We therefore conclude that `wrapper` runs `/home/sluo/hw1-files/private/read` after the `fork()`.

What this part was asking for didn't seem to be particularly clear to most of you – I got several answers along the lines of "changes its effective GID". While this does have to happen at some point (otherwise you wouldn't be able to trace the process), `wrapper` actually does this before forking its child process.

2. *Examining the process scheduler.* Here's an exercise looking at scheduling processes with different nice levels. *Do not run these examples on multi-user machines.*

a. Examine, compile, and run `forkloop.c` and `forkloop-io.c`, which fork child processes until they are interrupted by a signal or reach a cap on the total number of processes created. (Don't worry if you don't understand the C; the comments say everything you need to know about the operation of these two programs.) What distinguishes these two programs?

`forkloop`'s children simply (attempt to) compute the factorial of their process ID, while `forkloop-io`'s children read from a file on disk as well.

b. Run two copies of `forkloop` simultaneously with the same priority (see the script `run1`). Do they share the CPU roughly equally? What about with one copy running at lower priority (see script `run2`)?

We run each of these, and let it run for about 10 seconds before interrupting with Ctrl-C (`SIGINT`):

```
sluo@login:~/hw1-files$ ./run1
28563: forkloop priority 0
```

```
28562: forkloop priority 0
28562: forked 701 processes total
28563: forked 647 processes total
sluo@login:~/hw1-files$ ./run2
29912: forkloop priority 0
29913: forkloop priority 10
29912: forked 903 processes total
29913: forked 589 processes total
```

Unsurprisingly, two `forkloops` running with the same priority share the CPU roughly equally, while the copy with lower priority suffers when the two are run with different priorities.

c. Run `forkloop` and `forkloop-io` simultaneously with the same priority (see script `run3`). Do they share the CPU equally? What happens when you lower the priority of `forkloop` (`run4`)? Why?

```
sluo@login:~/hw1-files$ ./run3 ../zeros
31408: forkloop priority 0
31407: forkloop-io priority 0
31407: forked 312 processes total
31408: forked 816 processes total
sluo@login:~/hw1-files$ ./run4 ../zeros
27807: forkloop-io priority 0
27808: forkloop priority 10
27807: forked 267 processes total
27808: forked 825 processes total
```

`zeros` is a 1 GB file of null bytes (generated by reading from `/dev/zero` using `dd`).

Looking at `run3` and `run4`, we see that `run3` runs `forkloop` first before doing an `exec()` of `forkloop-io`, while `run4` runs `forkloop` at the lower priority. Using this information to distinguish the two programs in the output, we see that the `forkloop-io` process runs less frequently than the `forkloop` process does, and this remains true when the priority of the `forkloop` process is lowered. This happens largely because `forkloop-io` spends time blocking for I/O, and processes that are blocked cannot be scheduled.

This example didn't work as well for most of you as I intended, for two reasons: (1) I didn't bother to say that the file read should be a large file, without which

the reads will hit end of file and return nothing at all very quickly; and (2) for best results, you'd want to clear out the file cache (with something like a variant of `malloc2`, for example) between runs of `forkloop-io`, to make sure that each read blocks while waiting for the disk.

3. *The load average.* If you watched the output of `top` and/or `uptime` while running the `forkloop` examples, you probably noticed that the load average numbers spiked while they were running.

   a. How is this load average computed?

   The load average is an exponentially-damped moving average of the length of the run queue over the past m minutes (where load is usually computed for $m = 1$, $m = 5$, and $m = 15$). On Linux systems, the exact formula for the one-minute load average is

   $$L_i = L_{i-1}e^{-5/60} + n\left(1 - e^{-5/60}\right)$$

   where L_i is the current load average, L_{i-1} was the load average at the previous sampling point, and n is the length of the run queue. See `http://www.teamquest.com/resources/gunther/display/5/index.htm` (the source for this formula) for a bit of analysis on these figures.

   b. What is the "full utilization" load average for an n-processor machine? Why?

   The full utilization load for an n-processor machine (by "processor", we mean any physical hardware capable of having a thread scheduled on it, be it a processor in a socket, one core among many in a processor, or a processor with HyperThreading) is simply n. The run queue gives us the number of processes that are trying to run at any moment; on a system that's being fully utilized (but not overloaded), this should be equal to the number of processes that can actually be run at once.

   c. Suppose you have a system with only one process running. What are the minimum and maximum load averages possible, and why?

   The maximum load average in this situation is 1, since on a system with only one process running, the longest the run queue can get is one process. (In practice, the maximum load average would be achieved if this process were always ready to run – i.e. never slept or blocked for I/O.) The minimum load average is of course 0 (indicating the process has been sleeping or blocked all the time recently).

d. *Optional.* Suggest a small change or two to `forkloop.c` that would maximize the load average spike produced by running it. You do not need to test your change(s) (and you don't want to, unless you have a system that you are willing to hard reset afterward).

There are (at least) two possible changes here. You could let each child fork its own children, instead of only letting the parent create child processes. This lets the number of processes grow exponentially, instead of linearly, with time. You could also remove the `sleep(1)` that the children perform, which would make the child processes compete much more actively to run, making the run queue longer.

4. *Memory overcommit and out-of-memory behavior. Note: Do not try this exercise on production machines.*

Compile and run `malloc3.c`; this is similar to the `malloc2.c` demonstrated in class, but allocates all of the memory it desires before attempting to write to any of it.

a. What happens when you run this program?

```
sluo@adv-hw1-sluo:~$ ./malloc3
Succeeded in allocating 8 MB of RAM
Succeeded in allocating 16 MB of RAM
Succeeded in allocating 24 MB of RAM
Succeeded in allocating 32 MB of RAM
Succeeded in allocating 40 MB of RAM
Succeeded in allocating 48 MB of RAM
Succeeded in allocating 56 MB of RAM
Succeeded in allocating 64 MB of RAM
Succeeded in writing to 8 MB of RAM
Succeeded in writing to 16 MB of RAM
Succeeded in writing to 24 MB of RAM
Succeeded in writing to 32 MB of RAM
Succeeded in writing to 40 MB of RAM
Killed
```

The program is killed when the system runs out of memory. Examination of the kernel messages using `dmesg` shows

```
oom-killer: gfp_mask=0x280d2, order=0
```

```
Call Trace:
[way too much debugging output and memory info removed]

Out of Memory: Kill process 1455 (malloc3) score 17018
    and children.
Out of memory: Killed process 1455 (malloc3).
```

The out-of-memory killer, which activates when there's no more physical memory to be had and nothing can be pushed out (memory written to swap, or `mmap()`ed pages dropped, or cache dropped), assigns each process a score based on factors such as memory usage, and kills the one with the highest score.

Why did the system allocate so much memory in the first place (remember, we only have 64 MB of RAM and no swap, and undoubtedly there are other things in physical memory besides `malloc3`)? The Linux VM system overcommits memory – that is, it's willing to allocate memory beyond the amount of physical RAM and swap available to a certain point, in the expectation that not all allocated memory ends up being used. This reduces the need to swap to disk and/or deny memory allocations, improving performance.

b. Can you imagine scenarios where this behavior might affect a process other than the one writing to memory at the time the out-of-memory condition occurs?

The primary risk from the out-of-memory killer comes when no one process is hogging memory and piling up a big score. In such cases, it's possible that the victim could be an innocent bystander, perhaps an important system daemon whose killing would leave the system effectively inoperable. (No, `init` won't be killed under any circumstances, but on a system with no console access, the death of `sshd` effectively leaves the system inaccessible, for instance – several of you seem to have come across this problem while doing this homework.)

c. Try increasing the number of blocks the program tries to allocate (change the value `BLOCKS` is `#define`d to be). Can you reach a point where the memory allocation fails? Try increasing the block size (`BLOCK_SIZE`). Can you reach a point where the memory allocation fails? Explain your results.

Even with absurdly large values of `BLOCKS`, allocations succeed. In my testing, it takes more than 5000 blocks (about 40 GB) to produce an allocation failure. However, choosing `BLOCK_SIZE` larger than about 45 causes the allocation to fail. Your results may vary depending on how much memory was in use on your system at the time of the allocations.

Why? The VM only looks at the allocation size in comparison to the free (as in unused, not unallocated) memory. Hence 8 MB allocations continue to succeed, because the system has more than 8 MB unused memory available, but 48 MB allocations fail.

Why do allocations of more than a certain monstrously monstrous number of blocks fail, then? Each memory allocation incurs a certain memory overhead, both in-kernel and in userspace (`malloc()` does bookkeeping in userspace); at some point this overhead grows beyond our free memory, even if none of the allocated memory actually gets used.

d. In situations where the consequences of overcommitting memory are unacceptable, how would you go about disabling this on your Linux system? What would be some of the other effects of this change?

   The `malloc()` man page tells us that we can disable memory overcommit in the kernel by writing the value `2` to `/proc/sys/kernel/vm/overcommit_memory`. `vm/overcommit-accounting` in the kernel documentation (see any Linux kernel source tree, or `/usr/share/doc/linux-doc-2.6.18` on the login server) tells us that this means that "The total address space commit for the system is not permitted to exceed swap + a configurable percentage (default is 50) of physical RAM".

   Disabling memory overcommit will increase swap usage when the system is busy, thus decreasing system performance. Applications may also run slower if they could have made use of large sparse memory allocations.

e. *Optional.* Try `malloc3` with the configuration change from part (d). Use the machine for other tasks, and try to use up lots of memory; do you notice any effect on your system's performance and behavior? If so, were they effects you predicted?