Advanced Unix System Administration
Spring 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:

   ```
   -rwx--s--x 1 sluo sluo 10272 2008-02-24 15:30 wrapper
   ```

   A setgid program will be run as the group owning the binary, not as the group of the user running the binary (we'll talk more about this later in the class). For now, all you need to know is that 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, "V\v\32\2438\34d@"..., 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   22,1    1024   99084
    /home/sluo/hw1-files/random
read    30719 nobody    4r  FIFO   22,1           99083
    /home/sluo/hw1-files/fifo
```

The process has our tty open on file descriptors 0, 1, and 2. File descriptor 3 is `/home/sluo/hw1-files/random` (which I made by taking 1024 bytes out of `/dev/urandom`), while file descriptor 4 is `/home/sluo/hw1-files/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 and man page reading, but you do have enough information to answer this question, assuming you already did the first three parts of the problem.)

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

```
read    30719 nobody  txt    REG   22,1    9736   99081
```

`/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` invokes `/home/sluo/hw1-files/private/read` after the `fork()`.

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`)?

```
sluo@tempest:~/hw1-files$ ./run1
5076: priority 0
5075: priority 0
5076: forked 2461 processes total
5075: forked 2446 processes total
sluo@tempest:~/hw1-files$ ./run2
31876: priority 0
31877: priority 10
31876: forked 3116 processes total
31877: forked 2517 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`? Why?

```
sluo@tempest:~/hw1-files$ ./run3 big-file
24249: priority 0
24248: priority 0
24249: forked 793 processes total
24248: forked 481 processes total
```

```
sluo@tempest:~/hw1-files$ ./run4 big-file
25524: priority 0
25525: priority 10
25524: forked 475 processes total
25525: forked 847 processes total
```

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 because `forkloop-io` spends time blocking for I/O, and processes that are blocked cannot be scheduled.

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.* From `tempest`, SSH into `10.20.0.11` (the RSA host key fingerprint is `96:d8:56:84:b9:14:bc:b3:f8:27:fd:c1:6d:e4:bd:b4`); use your `tempest` login and password. This is a virtual machine configured with 128 MB of RAM and no swap.

On this host, 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-hw1-1:~$ ./malloc3
Succeeded in allocating 20 MB of RAM
Succeeded in allocating 40 MB of RAM
Succeeded in allocating 60 MB of RAM
Succeeded in allocating 80 MB of RAM
Succeeded in allocating 100 MB of RAM
Succeeded in allocating 120 MB of RAM
Succeeded in allocating 140 MB of RAM
Succeeded in allocating 160 MB of RAM
Succeeded in writing to 20 MB of RAM
Succeeded in writing to 40 MB of RAM
Succeeded in writing to 60 MB of RAM
Succeeded in writing to 80 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 2988 (malloc3) score 41594
    and children.
Out of memory: Killed process 2988 (malloc3).
```

The out-of-memory killer, which activates when there's no more physical memory to be had and nothing can be pushed out to swap, 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 128 MB of RAM and no swap)? 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.)

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 `#define BLOCKS 1024`, but the same `BLOCK_SIZE` (in other words, trying to allocate 20 GB of RAM in 20 MB blocks), the memory allocations all succeed. However, choosing `BLOCK_SIZE` larger than about 20-30 (80-120 MB; your results may vary, depending on exactly how much memory is in use on the system at the moment) results in the memory allocation failing.

Why? The VM only looks at the allocation size in comparison to the free (as in unused, not unallocated) memory. Hence 20 MB allocations continue to succeed, because the system has more than 20 MB unused memory available, but 120 MB allocations fail.

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.* If you have a Linux machine where you have root access, 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?

5. *VM behavior.* From `tempest`, SSH into `10.20.0.12` (the RSA host key fingerprint is `18:46:4a:3c:13:01:9a:a7:47:55:ab:10:be:c6:22:c7`); use your `tempest` login and password. This is a virtual machine configured with 64 MB of RAM and 64 MB of swap.

a. Create a large file (say, with `dd`). Find its SHA1 sum twice, timing it each time. Run `malloc3` from the previous problem, then time the SHA1 sum operation again. What results do you get? Can you explain them?

```
sluo@adv-hw1-2:~$ dd if=/dev/zero of=big-file bs=24M count=1
1+0 records in
1+0 records out
25165824 bytes (25 MB) copied, 0.086758 seconds, 290 MB/s
sluo@adv-hw1-2:~$ time sha1sum big-file
f13b36769904213001eb97475213960fae3120bb  big-file

real    0m0.181s
user    0m0.160s
sys     0m0.020s
sluo@adv-hw1-2:~$ time sha1sum big-file
f13b36769904213001eb97475213960fae3120bb  big-file

real    0m0.180s
user    0m0.164s
sys     0m0.016s
```

```
sluo@adv-hw1-2:~$ ./malloc3
[output removed]
sluo@adv-hw1-2:~$ time sha1sum big-file
f13b36769904213001eb97475213960fae3120bb  big-file

real    0m0.632s
user    0m0.092s
sys     0m0.008s
```

The first two SHA1 hash operations are fast, but the last one is considerably slower. This is because of the kernel's caching of data; when you create the file or read it, the kernel caches the result, in anticipation of future use of that data. Thus the first two SHA1 hashes operate entirely in memory. The cached data is pushed out when `malloc3` creates memory pressure, so the last SHA1 operation has to read all of the file back in from disk.

b. Compile `malloc4.c`, which acquires and uses lots of memory, then goes to sleep until interrupted by SIGHUP, at which point it rereads all the memory it used. Run the program alone, send SIGHUP as soon as possible, and observe how long it says the memory reread takes. (You may need to run it a few times to get repeatable results.) Run it again, and this time run `malloc3` before sending SIGHUP. How long does it take this time? Why the difference?

   We see the following:

```
[window 1]
sluo@adv-hw1-2:~ [1]$ ./malloc4
Succeeded in allocating 4 MB of RAM
Succeeded in allocating 8 MB of RAM
Succeeded in allocating 12 MB of RAM
Succeeded in allocating 16 MB of RAM
Succeeded in allocating 20 MB of RAM
Succeeded in allocating 24 MB of RAM
Succeeded in writing to 4 MB of RAM
Succeeded in writing to 8 MB of RAM
Succeeded in writing to 12 MB of RAM
Succeeded in writing to 16 MB of RAM
Succeeded in writing to 20 MB of RAM
Succeeded in writing to 24 MB of RAM
Sleeping until SIGHUP is received

[window 2]
sluo@adv-hw1-2:~ [2]$ pkill -HUP malloc4

[window 1]
```

```
Continuing.
Done -- reread took 20058 us.
Sleeping until SIGHUP is received

[window 2]
sluo@adv-hw1-2:~ [2]$ ./malloc3
[output removed]
sluo@adv-hw1-2:~ [2]$ pkill -HUP malloc4

[window 1]
Continuing.
Done -- reread took 889610 us.
```

The first reread is extremely fast, whereas the second is considerably slower. Running `malloc3` before triggering the second reread pushes `malloc4`'s memory to swap, so the second reread needs to pull this memory back in from disk.

c. Start by compiling `malloc5.c` (which behaves like `malloc3`, except it uses less memory), then creating a file about 24 MB or so in size. Run `malloc3`, then `malloc4`. While `malloc4` is sleeping, find the SHA1 sum of the 24 MB file a few times, timing it each time. Run `malloc5`. Time a few more SHA1 sums of the 24 MB file, then send SIGHUP to `malloc4` and note how long it takes to reread its memory.

Repeat the above exercise, except this time, after running `malloc5`, send SIGHUP to `malloc4` before timing the SHA1 sums.

Explain what's happening at each step of these two scenarios, and the differences between the two. (Hint: `top` may be useful to you here.)

```
[window 1]
sluo@adv-hw1-2:~ [1]$ ./malloc3
[output removed]
sluo@adv-hw1-2:~ [1]$ ./malloc4
[snip]
Sleeping until SIGHUP is received

[window 2]
sluo@adv-hw1-2:~ [2]$ time sha1sum big-file
f13b36769904213001eb97475213960fae3120bb  big-file

real    0m0.632s
user    0m0.208s
sys     0m0.040s
sluo@adv-hw1-2:~ [2]$ time sha1sum big-file
```

```
f13b36769904213001eb97475213960fae3120bb  big-file

real    0m0.605s
user    0m0.084s
sys     0m0.032s
sluo@adv-hw1-2:~ [2]$ ./malloc5
Succeeded in allocating 4 MB of RAM
Succeeded in allocating 8 MB of RAM
Succeeded in allocating 12 MB of RAM
Succeeded in allocating 16 MB of RAM
Succeeded in allocating 20 MB of RAM
Succeeded in allocating 24 MB of RAM
Succeeded in writing to 4 MB of RAM
Succeeded in writing to 8 MB of RAM
Succeeded in writing to 12 MB of RAM
Succeeded in writing to 16 MB of RAM
Succeeded in writing to 20 MB of RAM
Succeeded in writing to 24 MB of RAM
sluo@adv-hw1-2:~ [2]$ time sha1sum big-file
f13b36769904213001eb97475213960fae3120bb  big-file

real    0m0.950s
user    0m0.080s
sys     0m0.024s
sluo@adv-hw1-2:~ [2]$ time sha1sum big-file
f13b36769904213001eb97475213960fae3120bb  big-file

real    0m0.181s
user    0m0.152s
sys     0m0.028s
sluo@adv-hw1-2:~ [2]$ pkill -HUP malloc4

[window 1]
Continuing.
Done -- reread took 1058257 us.
```

The initial `malloc3` pushes everything not currently in use out of physical memory, thereby freeing up a big chunk for what follows. (We'll be using it for this sort of thing again in the future.) The `malloc4` then hogs some chunk of that memory, enough that `big-file` can't be resident in cache; hence the SHA1 sums run slowly. `malloc5` uses 24 MB of RAM before quitting, thereby pushing out part of `malloc4`'s memory. The first SHA1 sum needs to load in the code for `sha1sum` again, as well as all of the contents of the file, so it

runs even more slowly than before; however, with a 24 MB chunk of memory free, the entire contents of the file can be cached, so the second SHA1 sum runs quickly. `malloc4`'s reread of memory is slow because it has to pull in its memory from swap.

If we trigger the reread of memory from `malloc4` before trying the SHA1 sum:

```
[window 2]
sluo@adv-hw1-2:~ [2]$ ./malloc5
[output removed]
sluo@adv-hw1-2:~ [2]$ pkill -HUP malloc4

[window 1]
Continuing.
Done -- reread took 572148 us.
Sleeping until SIGHUP is received

[window 2]
sluo@adv-hw1-2:~ [2]$ time sha1sum big-file
f13b36769904213001eb97475213960fae3120bb  big-file

real    0m1.179s
user    0m0.060s
sys     0m0.016s
sluo@adv-hw1-2:~ [2]$ time sha1sum big-file
f13b36769904213001eb97475213960fae3120bb  big-file

real    0m0.519s
user    0m0.088s
sys     0m0.028s
```

`malloc4`'s reread is slow, but not as slow as last time, which suggests that not all of its memory got pushed out to swap. Indeed, if you had `top` open, and were sorting the output by memory usage (press 'O' to bring up the sort options, and then press 'n' to sort by memory use), you'd have seen `malloc4`'s memory usage drop – but not all the way down – confirming this suspicion. Meanwhile, the second SHA1 sum is slow, but again, not as slow as before – suggesting that at least a part of the file is getting into the cache this time.

Particularly on this last part of the exercise, your results may (and probably will) vary, since the VM heuristics depend in part on how recently a page was used. What I'm looking for is a good explanation of what's happening on the system, not this particular set of results.