Advanced Unix System Administration
Spring 2008
Homework 1

This assignment is due via email to <sluo+decal@ocf.berkeley.edu> by 11:59 PM on **Thursday, February 28**. All the files mentioned are in `~sluo/hw1-files` on the login server (`tempest.ocf.berkeley.edu`); the ones which can be used on machines other than the DeCal servers are also in a tarball `hw1-files.tar.gz` available from the website. If you do not already have access to the login server, you need to email me about setting up an account. If you have an OCF account, you can ask me to steal the password hash from there; otherwise, you'll need to arrange to meet me to have the account created.

Note that you'll have to do some documentation-reading to answer some of these questions. If you're stuck, don't hesitate to ask for help, but do try to look for the answers on your own first – learning where to look is one of the more important sysadmin skills.

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?

    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.

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

    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.)

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?

    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`)?

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?

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?

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

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

   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).

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?

   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?

   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.

   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?

   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?

   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?

   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.)