

Advanced Unix System Administration

Fall 2008

Homework 1

This assignment is due via email to <sluo+decal@ocf.berkeley.edu> by 11:59 PM on **Monday, October 6**. All the files mentioned are in `~sluo/hw1-files` on the login server (`coupdetat.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. We'll be handling account creation on Wednesday; if you won't be there, you need to email me about arranging for access to the login server.

For the purposes of this homework, you will have the use of a virtual machine accessible from `coupdetat` as `10.20.1.x`, where `x` is your login server UID minus 999 (i.e. if `id` returns 1001 on the login server, your VM is `10.20.1.2`). Log in with your login server username and password; the root password will also be the same as your account's password. For reference, the VMs are configured with 64 MB of RAM and no swap. Don't store anything you care about on these, as they're subject to disappear at any time after the assignment's complete.

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, 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!)

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.* It's recommended that you use your VM, or some other controlled environment, for this exercise.
 - 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` (`run4`)? 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.* It's recommended that you use your VM, or some other controlled environment with a reasonably small amount of memory, for this exercise. 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 `#defined` to be). Can you reach a point where the memory allocation fails (so that the program quits before it attempts to write to the allocated memory)? 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.* 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?