Advanced Unix System Administration
Spring 2007
Homework 1

This assignment is due via email to <sluo+decal@ocf.berkeley.edu> by 11:59 PM on **Friday, February 16**. All the files mentioned are in the tarball `hw1-files.tar.gz`, available from the website, or from `~sluo` on the login server (`plague.ocf.berkeley.edu`).

If you do not already have access to the login server, you may want to email me about setting up an account, as you may not want to run many of these exercises on your own machines, and you definitely don't want to do it on places like `inst.eecs` or the OCF. 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. *Memory overcommit and out-of-memory behavior. Note: Do not do 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?

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

   c. 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?

   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 some memory; do you notice any effect on your system's performance and behavior? If so, were they effects you predicted? [I'm still trying to get the infrastructure set up to give everyone their own virtual containers with full root access and the ability to play with things like kernel parameters, so this isn't really possible on the DeCal machine this week, unfortunately.]

2. *Use of `mmap()`.* Compile the two programs `wcl.c` and `wcl-mmap.c`. These two programs count the number of newlines in a file, reading the whole file into memory

first (similar to what `wc -l` does, but in a dumber way). As the name suggests, `wcl-mmap` uses `mmap()`, while `wcl` uses `read()`. Run the programs on a variety of text files large and small (a copy of Project Gutenberg's edition of *War and Peace* is included in the tarball).

   a. Which one is faster? Why?

   b. *Do not do this on production machines.* Compile and run `malloc4.c`, which is a modification of `malloc2.c` that uses smaller block sizes and continues to run until interrupted by a signal. With `malloc4` running (and hogging memory), try `wcl` and `wcl-mmap` again on reasonably large files (greater than the amount of free memory available on the system; `cat`ing *War and Peace* together a few times should do it). Which one works? Why?

   c. *Optional; might take a bit of investigation.* When would one *not* want to use `mmap()` for file access?

3. *Examining the process scheduler.* The examples I hacked up for the demonstration in class didn't behave as expected because I didn't think them through enough; here's a chance to play with examples that actually work. *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?

4. *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. *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 afterwards).