Advanced Unix System Administration
Spring 2007
Homework 1 Solutions

1. *Memory overcommit and out-of-memory behavior.* 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-login:~/hw1$ ./malloc3
Succeeded in allocating 400 MB of RAM
Succeeded in allocating 800 MB of RAM
Succeeded in allocating 1200 MB of RAM
Succeeded in allocating 1600 MB of RAM
Succeeded in allocating 2000 MB of RAM
Succeeded in allocating 2400 MB of RAM
Succeeded in allocating 2800 MB of RAM
Succeeded in allocating 3200 MB of RAM
Succeeded in writing to 400 MB of RAM
Succeeded in writing to 800 MB of RAM
Succeeded in writing to 1200 MB of RAM
Succeeded in writing to 1600 MB of RAM
Killed
```

   The program is killed when the system runs out of memory. Examination of the kernel messages (which, unfortunately, you weren't able to see if you were on the login server) shows:

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

Call Trace:

[way too much debugging output and memory info removed]

Out of Memory: Kill process 31446 (malloc3) score 819834
    and children.
Out of memory: Killed process 31446 (malloc3).
```

   The out-of-memory killer, which activates when there's no more physical memory to be had and nothing can be pushed out, 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 (this is a system with 2 GB 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. 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?

```
sluo@adv-login:~/hw1$ dd if=/dev/zero of=big-file bs=128M count=1
1+0 records in
1+0 records out
134217728 bytes (134 MB) copied, 0.474338 seconds, 283 MB/s
sluo@adv-login:~/hw1$ time sha1sum big-file
ba713b819c1202dcb0d178df9d2b3222ba1bba44  big-file

real    0m0.954s
user    0m0.812s
sys     0m0.140s
sluo@adv-login:~/hw1$ time sha1sum big-file
ba713b819c1202dcb0d178df9d2b3222ba1bba44  big-file

real    0m0.953s
user    0m0.800s
sys     0m0.148s
sluo@adv-login:~/hw1$ ./malloc3
[output removed]
sluo@adv-login:~/hw1$ time sha1sum big-file
ba713b819c1202dcb0d178df9d2b3222ba1bba44  big-file

real    0m2.209s
user    0m0.340s
sys     0m0.104s
```

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.

Note that if you had used a pre-existing big file, or trashed the cache between creating it and calculating the hash, the first and last SHA1 operations would be slow, while the second would still be fast.

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?

   This question was unclear; by "this behavior" I meant the behavior of the out-of-memory killer. I'm ignoring it for purposes of grading the homework.

   Anyhow, 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.)

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 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 some memory; do you notice any effect on your system's performance and behavior? If so, were they effects you predicted?

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?

```
sluo@adv-login:~/hw1$ time ./wcl war-and-peace.txt
67337 war-and-peace.txt

real    0m0.154s
user    0m0.000s
```

```
sys     0m0.008s
sluo@adv-login:~/hw1$ ./malloc3
sluo@adv-login:~/hw1$ time ./wcl-mmap war-and-peace.txt
67337 war-and-peace.txt

real    0m0.143s
user    0m0.000s
sys     0m0.000s
```

The memory-mapped version is a bit faster, as the kernel is allowed to handle the I/O itself. The call to `malloc3` in between the invocations of the versions of `wcl` was done to trash the cache; if you didn't do this, you might have gotten inconsistent results or not noticed much of a difference. Practically, though, we don't expect a very large difference – both programs do essentially the same thing, one big read from disk into memory.

b. 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?

   `wcl` complains that it's "unable to allocate buffer", while `wcl-mmap` works fine. This is because `wcl` calls `malloc()` to allocate all of the buffer at once; this causes the kernel to look for a large memory allocation, which is refused. On the other hand, `wcl-mmap` maps the file into memory; the kernel is therefore free to have as much or as little of the file as it wants actually in physical memory at any one time. This allows `wcl-mmap` to work with the file as if it were all in memory (like both `wcl` and `wcl-mmap` expect), even though not all of it fits.

c. *Optional.* When would one *not* want to use `mmap()` for file access?

   The answer I was looking for is that `mmap()` works very poorly, if at all, over NFS; this is because memory-mapped I/O is integrated intimately with the kernel's VM system, and having network access tied into the work of the kernel's paging system leads to many nasty issues. Consider, for instance, a case in which the kernel is under memory pressure and looking to push out a memory-mapped page backed by NFS to disk; this requires sending network traffic to the NFS server, which might require allocating more memory! There are also nasty caching issues with the design of NFS that make guaranteeing data integrity in such a setup very difficult.

   That was, however, by no means the only correct answer. It was also pointed out that it's impossible to `mmap()` all of a file – or any amount of data, in general – larger than the size of the virtual address space. VM subsystems also tend to be extremely complicated beasts, and it usually works out to be easier, safer, and more portable to use standard I/O mechanisms, in conjunction with good buffering, than to use memory-mapped I/O.

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.

   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@adv-login:~/hw1$ ./run1
31671: priority 0
31670: priority 0
31671: forked 4163 processes total
31670: forked 4180 processes total
sluo@adv-login:~/hw1$ ./run2
7570: priority 0
7571: priority 10
7570: forked 3432 processes total
7571: forked 2746 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@adv-login:~/hw1$ ./run3 big-file
16111: priority 0
16110: priority 0
16110: forked 1670 processes total
16111: forked 2001 processes total
sluo@adv-login:~/hw1$ ./run4 big-file
```

```
19785: priority 0
19786: priority 10
19785: forked 1991 processes total
19786: forked 1473 processes total
```

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.

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?

   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. *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).

   There are 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.