Lab Handout 2: Multiprocessing and Unix Tools

The first three problems are questions that could easily appear on a midterm or final exam. The exercises beyond that are designed to help you master the development tools even more so than you already have.

Students are encouraged to share their ideas in #lab2 on Slack and on Piazza. SCPD students are welcome to reach out to me or Hemanth directly if they have questions that can't be properly addressed without being physically present for a discussion section. The lab checkoff sheet for all students–both on-campus and off–can be found right here.

Before starting, go ahead and clone the `lab2` folder, which contains a working solution to Problem 1 and fodder for some of the later problems that have you experiment with some new development tools.

```
poohbear@myth15:~$ hg clone /usr/class/cs110/repos/lab2/shared lab2
poohbear@myth15:~$ cd lab2
poohbear@myth15:~$ make
```

Problem 1: Implementing `exargs`

`exargs` (designed here to emulate the `xargs` builtin–type `man xargs` for the full read) reads tokens from standard input (delimited by spaces and newlines), and executes the command with any initial arguments followed by the tokens read from standard input. `exargs` (or `xargs`) is useful when one program is needed to programmatically generate the argument vector for a second. (Understand that the builtin `xargs` is much more sophisticated than our `exargs`; `xargs` has all kinds of parsing options and flags, but our `exargs` has no such bells and whistles.)

To illustrate the basic idea, consider the `factor` program, which prints out the prime factorizations of all of its numeric arguments, as with:

```
poohbear@myth15:~$ factor 720
720:  2 2 2 2 3 3 5
poohbear@myth15:~$ factor 9 16 2047 870037764750
9:  3 3
16:  2 2 2 2
2047:  23 89
```

```
870037764750:  2 3 3 5 5 5 7 7 7 7 11 11 11 11 11
poohbear@myth15:~$
```

To see how **exargs** works, check this out:

```
poohbear@myth15:~$ printf "720" | ./exargs factor
720:  2 2 2 2 3 3 5
poohbear@myth15:~$ printf "2047 1000\n870037764750" | ./exargs factor 9 16
9:  3 3
16:  2 2 2 2
2047:  23 89
1000:  2 2 2 5 5 5
870037764750:  2 3 3 5 5 5 7 7 7 7 11 11 11 11 11
poohbear@myth15:~$
```

Note that the first process in the pipeline–the **printf**–is a brute force representative of an executable capable of supplying or extending the argument vector of a second executable–in this case, **factor**–through **exargs**. Of course, the two executables needn't be **printf** or **factor**; they can be anything that works. If, for example, I'm interested in exposing just how much code I had to write for my own **assign2** solution (see if you can access the folder :)), I might use **exargs** to do this:

```
poohbear@myth30:~$ \ls /usr/class/cs110/staff/master_repos/assign2/*.c | ./
exargs wc --chars --lines --max-line-length
   78  1792    90 /usr/class/cs110/staff/master_repos/assign2/chksumfile.c
   35  1178   121 /usr/class/cs110/staff/master_repos/assign2/directory.c
  266  8015   111 /usr/class/cs110/staff/master_repos/assign2/diskimageaccess.c
   31   731    86 /usr/class/cs110/staff/master_repos/assign2/diskimg.c
   35  1193   144 /usr/class/cs110/staff/master_repos/assign2/file.c
   72  2751   134 /usr/class/cs110/staff/master_repos/assign2/inode.c
   33   987   152 /usr/class/cs110/staff/master_repos/assign2/pathname.c
   45  1287    91 /usr/class/cs110/staff/master_repos/assign2/unixfilesystem.c
  595 17934   152 total
```

As a whiteboard exercise, construct the entire implementation of the **exargs** program, relying on the following utility function to parse all of standard input around newlines and spaces:

```
static void pullAllTokens(istream& in, vector<string>& tokens) {
  while (true) {
    string line;
    getline(in, line);
    if (in.fail()) break;
    istringstream iss(line);
    while (true) {
      string token;
      getline(iss, token, ' ');
      if (iss.fail()) break;
      tokens.push_back(token);
    }
  }
}
```

You need not perform any error checking on user input, and you can assume that all system calls succeed. Implement the entire program to return 0 if and only if the command executed by **exargs** exits normally with status code 0, and return 1 otherwise.

```
int main(int argc, char *argv[]) {
```

Problem 2: Incorrect Output File Redirection

The **publish** user program takes an arbitrary number of filenames as arguments and attempts to publish the date and time (via the **date** executable that ships with all versions of Unix and Linux). **publish-to-all** is built from the following source:

```
static void publish(const char *name) {
   printf("Publishing date and time to file named \"%s\".\n", name);
   int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
   dup2(outfile, STDOUT_FILENO);
   close(outfile);
   if (fork() > 0) return;
   char *argv[] = { "date", NULL };
   execvp(argv[0], argv);
}
```

```
int main(int argc, char *argv[]) {
   for (size_t i = 1; i < argc; i++) publish(argv[i]);
   return 0;
}
```

Someone with a fractured understanding of processes, descriptors, and file redirection might expect the program to have printed something like this:

```
myth4> ./publish one two three four
Publishing date and time to file named "one".
Publishing date and time to file named "two".
Publishing date and time to file named "three".
Publishing date and time to file named "four".
```

However, that's not what happens. Questions:

- What text is actually printed?
- What do each of the four files contain?
- How should the program be rewritten so that it works as intended?

Problem 3: Short Answer Questions

Here are a bunch of short answer questions that have appeared on past CS110 midterms and final exams.

- Recall that one vnode table and one file entry table are maintained on behalf of all processes, but that each process maintains its own file descriptor table. What problems would result if just one file descriptor table were maintained on behalf of all processes?
- Your terminal can be configured so that a process **dumps core**–that is, generates a data file named `core`–whenever it crashes (because it seg faults, for instance.) This `core` file can be loaded into and analyzed within `gdb` to help identify where and why the program is crashing. Assuming we can modify the program source code and recompile, how might you **programmatically** generate a core dump at specific point in the program while allowing the process to continue executing? (Your answer might include a **very, very short** code snippet to make its point.)
- The `fork` system call creates a new process with an independent address space, where the contents of the parent's address space are replicated–in a sense, `memcpy`'ed–into the address space of the clone. If, however, a **copy-on-write** implementation strategy is

adopted, then both parent and child share the same address space and only start to piecemeal split into two independent address spaces as one process makes changes that shouldn't be reflected in the other. In general, most operating systems adopt a copy-on-write approach, even though it's more difficult to implement. Defend that approach given what we've learned in class so far.

Problem 4: Experimenting with `strace`

`strace` is an advanced development tool that programmers use to determine what system calls contribute to the execution of a program (generally because the program is malfunctioning, and they're curious if any failing system calls are the root cause of the problem). If, for instance, you want to see how sleep 10 works behind the scenes, you gain a lot by typing this in and following along:

```
// lots of calls omitted in the name of brevity

fstat(3, {st_mode=S_IFREG|0644, st_size=4431376, ...}) = 0
mmap(NULL, 4431376, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f62a18ae000
close(3)                                = 0
nanosleep({10, 0}, NULL)                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
poohbear@myth30:~$poohbear@myth30:~$ strace sleep 10
execve("/bin/sleep", ["sleep", "10"], [/* 27 vars */]) = 0
brk(0)                                  = 0x1636000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f62a22d0000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

// lots of calls omitted in the name of brevity

fstat(3, {st_mode=S_IFREG|0644, st_size=4431376, ...}) = 0
```

```
mmap(NULL, 4431376, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f62a18ae000
close(3)                                = 0
nanosleep({10, 0}, NULL)                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
poohbear@myth30:~$
```

A typical **strace** run starts with an **execve** (which my shell uses instead of **execvp**), and then works through all of these systemsy things to load C++ libraries, build the heap segment, etc., until it reaches the crucial **nanosleep** call, which is the call that halts the process for 10 seconds. You see gestures to system calls that have come up in CS107, CS110, and your first two assignments: **execve**, **access**, **mmap**, **open**, and **close**.

If you're interested only in a particular subset of system calls, you can identify those of interest when invoking **strace** using the **-e trace=***<csv-list-of-syscall-name>*, as with this:

```
// lots of calls omitted in the name of brevity

munmap(0x7f2ec1242000, 154308)         = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2ec1267000
munmap(0x7f2ec1267000, 4096)           = 0
mmap(NULL, 4431376, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2ebfdd4000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2ec1267000
ARCHIVE.README   cgi-bin  include  lecture-examples  lib  local   private_data
 README.md  repos   samples  staff   tools  WWW
munmap(0x7f2ec1267000, 4096)           = 0
+++ exited with 0 +++
poohbear@myth30:~$poohbear@myth30:~$ strace -e trace=read ls /usr/class/cs110
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0[\0\0\0\0\0\0"..., 832) = 832
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`\34\0\0\0\0\0\0"..., 832) = 832
```

```
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P \2\0\0\0\0\0"..., 832)
= 832

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\27\0\0\0\0\0\0"...,
832) = 832

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320\16\0\0\0\0\0\0"...,
832) = 832

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300\20\0\0\0\0\0\0"...,
832) = 832

read(3, "nodev\tsysfs\nnodev\trootfs\nnodev\tr"..., 1024) = 344

read(3, "", 1024)                       = 0

cgi-bin  include  lecture-examples  lib  local    private_data  README.md  repos
   samples  staff    tools  WWW

+++ exited with 0 +++

poohbear@myth30:~$ strace -e trace=mmap,munmap ls /usr/class/cs110

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f2ec1268000

mmap(NULL, 154308, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2ec1242000

mmap(NULL, 2238192, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f2ec0e25000

mmap(0x7f2ec1044000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x1f000) = 0x7f2ec1044000

mmap(0x7f2ec1046000, 5872, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x7f2ec1046000

mmap(NULL, 2126336, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f2ec0c1d000

mmap(0x7f2ec0e23000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x6000) = 0x7f2ec0e23000


// lots of calls omitted in the name of brevity


munmap(0x7f2ec1242000, 154308)          = 0

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f2ec1267000

munmap(0x7f2ec1267000, 4096)            = 0

mmap(NULL, 4431376, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2ebfdd4000

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f2ec1267000
```

```
ARCHIVE.README    cgi-bin  include  lecture-examples  lib  local    private_data
 README.md  repos    samples  staff    tools  WWW
munmap(0x7f2ec1267000, 4096)            = 0
+++ exited with 0 +++
poohbear@myth30:~$
```

Take some time to experiment with `strace`, as it'll help you with your Assignment 3 work (because you implement a light version of `strace` as part of it–surprise!). Type in each of these commands from any directory and see what you get:

- `strace date`
- `strace -e trace=write date`
- `strace -e trace=clone,execve date`

Note that calls to `fork` are reframed as calls to `clone`, and calls to `execvp` are reframed as calls to `execve`.

Rather than list out specific system calls via `-e trace=`*<csv-list-of-syscall-name>*, `strace` allows you to specify a family of system calls, as per `strace`'s man page, e.g. `-e trace=file`, or `-e trace=memory`.

Now type these commands from any directory and see what you get:

- `strace -e trace=file ls /usr/class/cs110`
- `strace -e trace=memory ls /usr/class/cs110`
- `strace -e trace=desc ls /usr/class/cs110`

See what happens when you try to launch something that can't be launched, because the specified file isn't an executable:

- `strace /usr/class/cs110/WWW`
- `strace /usr/class/cs110/WWW/index.html`

Just for fun, try `strace strace`, with:

- `strace strace ls`

Finally, descend into your `lab2` folder and trace a few things there:

- `printf "4 6 8" | strace -e trace=clone,execve ./exargs factor`
- `printf "4 6 8" | strace -f -e trace=clone,execve ./exargs factor`

Scan `strace`'s man page for information about the `-f` flag, and be sure you understand why the output of the second command includes so many calls to `execve`.

Problem 5: Using `valgrind` and `gdb` with `fork`

You know by this point that my own solution to Problem 1 resides within that `lab2` folder you cloned. We're going to use this problem as a vehicle for learning how to use `gdb` to step through processes that split into multiple ones.

This exercise is framed in terms of an intentionally buggy version of `exargs.cc`, which I've placed in `buggy-exargs.cc`. I've done bad things in `buggy-exargs.cc` to make sure that `buggy-exargs` fails miserably. Here's a diff between the buggy version and the correct one:

```
poohbear@myth30:$ diff buggy-exargs.cc exargs.cc
47,48c47,48
<     char **exargsv = NULL;
<     memcpy(exargsv, argv + 1, (argc - 1) * sizeof(char *));
---
>     char *exargsv[argc + tokens.size()];
>     memcpy(exargsv, argv + 1, (argc - 1) * sizeof(char *));
poohbear@myth30:~/lab2$
```

Run the following commands to reaffirm that the correct version runs clean under `valgrind` (aside from the suppressed errors I spoke of in the Assignment 1 handout):

```
poohbear@myth30:~/lab2$ printf "1 2 3 4" | ./exargs factor
1:
2:  2
3:  3
4:  2 2
poohbear@myth30:~/lab2$ printf "1 2 3 4" | valgrind ./exargs factor
==12801== Memcheck, a memory error detector
==12801== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==12801== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==12801== Command:  ./exargs factor
==12801==
1:
2:  2
```

```
3: 3
4: 2 2
==12801==
==12801== HEAP SUMMARY:
==12801==     in use at exit: 72,704 bytes in 1 blocks
==12801==   total heap usage: 13 allocs, 12 frees, 73,011 bytes allocated
==12801==
==12801== LEAK SUMMARY:
==12801==    definitely lost: 0 bytes in 0 blocks
==12801==    indirectly lost: 0 bytes in 0 blocks
==12801==      possibly lost: 0 bytes in 0 blocks
==12801==    still reachable: 0 bytes in 0 blocks
==12801==         suppressed: 72,704 bytes in 1 blocks
==12801==
==12801== For counts of detected and suppressed errors, rerun with: -v
==12801== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
poohbear@myth30:~/lab2$
```

Not surprisingly, things don't go so well with the buggy version:

```
// reports of still reachable memory resulting from premature exit

poohbear@myth30:~/lab2$ poohbear@myth30:~/lab2$ printf "1 2 3 4" | ./
buggy-exargs factor
poohbear@myth30:~/lab2$ printf "1 2 3 4" | valgrind ./buggy-exargs factor
==14058== Memcheck, a memory error detector
==14058== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==14058== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==14058== Command: ./buggy-exargs factor
==14058==
==14059== Invalid write of size 8
==14059==    at 0x4C2F793: memcpy@@GLIBC_2.14 (in /usr/lib/valgrind/
vgpreload_memcheck-amd64-linux.so)
```

```
==14059==    by 0x4013A7: main (buggy-exargs.cc:48)
==14059==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==14059==
==14059==
==14059== Process terminating with default action of signal 11 (SIGSEGV)
==14059==  Access not within mapped region at address 0x0
==14059==    at 0x4C2F793: memcpy@@GLIBC_2.14 (in /usr/lib/valgrind/
vgpreload_memcheck-amd64-linux.so)
==14059==    by 0x4013A7: main (buggy-exargs.cc:48)

// reports of still reachable memory resulting from premature exit

poohbear@myth30:~/lab2$
```

`valgrind` tells us that `memcpy` is trying to write memory to an invalid address (`0x0` is as `NULL` an address as they come). Notice that the second process of the second pipeline is set up so that `valgrind` runs `./buggy-exargs` runs `factor`. Neat!

We'll play dumb and pretend we don't know why it's segfaulting, even if `valgrind` does tell us that line 48 of `buggy-exargs.cc` seems to be the culprit. If we want to use `gdb` to set a breakpoint in `buggy-exargs.cc`, line 48, it's totally possible. You just do so like this:

```
poohbear@myth30:~/lab2$ gdb --args ./buggy-exargs factor
// startup preamble omitted for brevity
Reading symbols from ./buggy-exargs...done.
(gdb) list 48
43      vector<string> tokens;
44      pullAllTokens(cin, tokens);
45      pid_t pid = fork();
46      if (pid == 0) {
47        char **exargsv = NULL;
48        memcpy(exargsv, argv + 1, (argc - 1) * sizeof(char *));
49        transform(tokens.cbegin(), tokens.cend(), exargsv + argc - 1,
50                  [](const string& str) { return const_cast<char
*>(str.c_str()); });
51        exargsv[argc + tokens.size() - 1] = NULL;
52        execvp(exargsv[0], exargsv);
```

```
(gdb) b 48
Breakpoint 1 at 0x401381: file buggy-exargs.cc, line 48.
(gdb)
```

However, according to this, by default gdb doesn't track execution of additional processes spawned off by the primary. If we were to type run right now, `gdb` would proceed and fully circumvent the block (or rather, the child process continuation) and miss our breakpoint entirely. Sadness.

Let's confirm by having you execute the following:

- type `run`, advancing the `gdb` trace of the parent to block on some `getline` call within `pullAllTokens`
- manually type something like `"1 2 3 4"` (without the double quote) to feed the process's standard input, and then hit ctrl-D to close standard input down
- note that the primary process ends

Here are some questions for you:

- Why is the primary process permitted to run to completion? What is its return value (i.e. its returned status code)?
- Why do we need to manually type in `"1 2 3 4"`? Why can't we just do what we've done prior and go with something like `printf "1 2 3 4" | gdb --args ./buggy-exargs factor` instead? Why were we able to do this with `valgrind`?
- What gdb command can you type in before you type `run` to trace execution of the child process instead of the parent after the `fork`?

Relaunch `gdb` as you did before, but this time configure it to follow the child process instead of the parent. Confirm that you break on line 48 as intended, and further confirm that `exargsv` is `NULL` as if you've never noticed it before and you've found your bug.