2 Background

Sometimes the journey of a thousand steps begins by learning to walk

Bhuvy

Systems Architecture

This section is a short review of System Architecture topics that you'll need for System Programming.

Assembly

What is assembly? Assembly is the lowest that you'll get to machine language without writing 1's and 0's. Each computer has an architecture, and that architecture has an associated assembly language. Each assembly command has a 1:1 mapping to a set of 1's and 0's that tell the computer exactly what to do. For example, the following in the widely used x86 Assembly language add one to the memory address 20 [13] – you can also look in [8] Section 2A under the add instruction though it is more verbose.

```
add BYTE PTR [0x20], 1
```

Why do we mention this? Because it is important that although you are going to be doing most of this class in C. That this is what the code is translated into. Serious implications arise for race conditions and atomic operations.

Atomic Operations

An operation is atomic if no other processor should interrupt it. Take for example the above assembly code to add one to a register. In the architecture, it may actually have a few different steps on the circuit. The operation

may start by fetching the value of the memory from the stick of ram, then storing it in the cache or a register, and then finally writing back [12] – under the description for *fetch-and-add* though your micro-architecture may vary. Or depending on performance operations, it may keep that value in cache or in a register which is local to that process – try dumping the `-O2` optimized assembly of incrementing a variable. The problem comes in if two processors try to do it at the same time. The two processors could at the same time copy the value of the memory address, add one, and store the same result back, resulting in the value only being incremented once. That is why we have a special set of instructions on modern systems called atomic operations. If an instruction is atomic, it makes sure that only one processor or thread performs any intermediate step at a time. With x86 this is done by the `lock` prefix [8, p. 1120].

```
lock add BYTE PTR [0x20], 1
```

Why don't we do this for everything? It makes commands slower! If every time a computer does something it has to make sure that the other cores or processors aren't doing anything, it'll be much slower. Most of the time we differentiate these with special consideration. Meaning, we will tell you when we use something like this. Most of the time you can assume the instructions are unlocked.

Caching

Ah yes, Caching. One of computer science's greatest problems. Caching that we are referring is processor caching. If a particular address is already in the cache when reading or writing, the processor will perform the operation on the cache such as adding and update the actual memory later because updating memory is slow [9, Section 3.4]. If it isn't, the processor requests a chunk of memory from the memory chip and stores it in the cache, kicking out the least recently used page – this depends on caching policy, but Intel's does use this. This is done because the l3 processor cache is roughly three times faster to reach than the memory in terms of time [11, p. 22] though exact speeds will vary based on the clock speed and architecture. Naturally, this leads to problems because there are two different copies of the same value, in the cited paper this refers to an unshared line. This isn't a class about caching, know how this could impact your code. A short but non-complete list could be

1. Race Conditions! If a value is stored in two different processor caches, then that value should be accessed by a single thread.
2. Speed. With a cache, your program may look faster mysteriously. Just assume that reads and writes that either happened recently or are next to each other in memory are fast.
3. Side effects. Every read or write effects the cache state. While most of the time this doesn't help or hurt, it is important to know. Check the Intel programmer guide on the lock prefix for more information.

Interrupts

Interrupts are a important part of system programming. An interrupt is internally an electrical signal that is delivered to the processor when something happens – this is a hardware interrupt [3]. Then the hardware decides if this is something that it should handle (i.e. handling keyboard or mouse input for older keyboard and mouses) or it should pass to the operating system. The operating system then decides if this is something that it should handle (i.e. paging a memory table from disk) or something the application should handle (i.e. a

SEGFAULT). If the operating system decides that this is something that the process or program should take care of, it sends a **software fault** and that software fault is then propagated. The application then decides if it is an error (SEGFAULT) or not (SIGPIPE for example) and reports to the user. Applications can also send signals to the kernel and to the hardware as well. This is an oversimplification because there are certain hardware faults that can't be ignored or masked away, but this class isn't about teaching you to build an operating system.

An important application of this is this is how system calls are served! There is a well-established set of registers that the arguments go in according to the kernel as well as a system call "number" again defined by the kernel. Then the operating system triggers an interrupt which the kernel catches and serves the system call [7].

Operating system developers and instruction set developers alike didn't like the overhead of causing an interrupt on a system call. Now, systems use `SYSENTER` and `SYSEXIT` which has a cleaner way of transferring control safely to the kernel and safely back. What safely means is obvious out of the scope for this class, but it persists.

Optional: Hyperthreading

Hyperthreading is a new technology and is in no way shape or form multithreading. Hyperthreading allows one physical core to appear as many virtual cores to the operating system [8, P.51]. The operating system can then schedule processes on these virtual cores and one core will execute them. Each core interleaves processes or threads. While the core is waiting for one memory access to complete, it may perform a few instructions of another process thread. The overall result is more instructions executed in a shorter time. This potentially means that you can divide the number of cores you need to power smaller devices.

There be dragons here though. With hyperthreading, you must be wary of optimizations. A famous hyperthreading bug that caused programs to crash if at least two processes were scheduled on a physical core, using specific registers, in a tight loop. The actual problem is better explained through an architecture lens. But, the actual application was found through systems programmers working on OCaml's mainline [10].

Debugging and Environments

I'm going to tell you a secret about this course: it is about working smarter *not* harder. The course can be time-consuming but the reason that so many people see it as such (and why so many students don't see it as such) is the relative familiarity of people with their tools. Let's go through some of the common tools that you'll be working on and need to be familiar with.

ssh

`ssh` is short for the Secure Shell [2]. It is a network protocol that allows you to spawn a shell on a remote machine. Most of the times in this class you will need to ssh into your VM like this

```
$ ssh netid@sem-cs241-VM.cs.illinois.edu
```

If you don't want to type your password out every time, you can generate an ssh key that uniquely identifies your machine. If you already have a key pair, you can skip to the copy id stage.

```
> ssh-keygen -t rsa -b 4096
# Do whatever keygen tells you
# Don't feel like you need a passcode if your login password is
    secure
> ssh-copy-id netid@sem-cs241-VM.cs.illinois.edu
# Enter your password for maybe the final time
> ssh  netid@sem-cs241-VM.cs.illinois.edu
```

If you still think that that is too much typing, you can always alias hosts. You may need to restart your VM or reload sshd for this to take effect. The config file is available on Linux and Mac distros. For Windows, you'll have to use the Windows Linux Subsystem or configure any aliases in PuTTY

```
> cat ~/.ssh/config
Host vm
  User         netid
  HostName     sem-cs241-VM.cs.illinois.edu
> ssh vm
```

git

What is 'git'? Git is a version control system. What that means is git stores the entire history of a directory. We refer to the directory as a repository. So what do you need to know is a few things. First, create your repository with the repo creator. If you haven't already signed into enterprise GitHub, make sure to do so otherwise your repository won't be created for you. After that, that means your repository is created on the server. Git is a decentralized version control system, meaning that you'll need to get a repository onto your VM. We can do this with a clone. Whatever you do, **do not go through the README.md tutorial**.

```
$ git clone
    https://github-dev.cs.illinois.edu/cs241-fa18/<netid>.git
```

This will create a local repository. The workflow is you make a change on your local repository, add the changes to a current commit, actually commit, and push the changes to the server.

```
$ # edit the file, maybe using vim
$ git add <file>
$ git commit -m "Committing my file"
$ git push origin master
```

Now to explain git well, you need to understand that git for our purposes will look like a linked list. You will always be at the head of master, and you will do the edit-add-commit-push loop. We have a separate branch on Github that we push feedback to under a specific branch which you can view on Github website. The markdown file will have information on the test cases and results (like standard out).

Every so often git can break. Here is a list of commands you probably won't need to fix your repo

1. git-cherry-pick
2. git-pack
3. git-gc
4. git-clean
5. git-rebase
6. git-stash/git-apply/git-pop
7. git-branch

If you are currently on a branch, and you don't see either

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

or

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
     directory)

        modified:  <FILE>
        ...

no changes added to commit (use "git add" and/or "git commit -a")
```

And something like

```
$ git status
HEAD detached at 4bc4426
nothing to commit, working directory clean
```

Don't panic, but your repository may be in an unworkable state. If you aren't nearing a deadline, come to office hours or ask your question on Piazza, and we'd be happy to help. In an emergency scenario, delete your repository and re-clone (you'll have to add the `release` as above). **This will lose any local uncommitted changes. Make sure to copy any files you were working on outside the directory, remove and copy them back in**

If you want to learn more about git, there are all but an endless number of tutorials and resources online that can help you. Here are some links that can help you out

1. `https://git-scm.com/docs/gittutorial`
2. `https://www.atlassian.com/git/tutorials/what-is-version-control`
3. `https://thenewstack.io/tutorial-git-for-absolutely-everyone/`

Editors

Some people take this as an opportunity to learn a new editor, others not so much. The first part is those of you who want to learn a new editor. In the editor war that spans decades, we have come to the battle of vim vs emacs.

Vim is a text editor and a Unix-like utility. You enter vim by typing `vim [file]`. This takes you into the editor. You start off in normal mode. In this mode, you can move around with many keys with the most common ones being `jklh`. To exit vim from this mode, you need to type `:q` which quits. If you have any unsaved edits, you must either save them `:w`, save and quit `:wq`, or discard changes `:q!`. To make edits you can either type `i` to change you into insert mode or `a` to change to insert mode **a**fter the cursor. This is the basics when it comes to vim

Emacs is more of a way of life, and I don't mean that figuratively. A lot of people say that emacs is a powerful operating system lacking a decent text editor. This means emacs can house a terminal, gdb session, ssh session, code and a whole lot more. It would not be fitting any other way to introduce you to the gnu-emacs any other way than the gnu-docs `https://www.gnu.org/software/emacs/tour/`. Just note that emacs is *insanely* powerful. You can do almost anything with it. There are a fair number of students who like the IDE-aspect of other programming languages. Know that you can set up emacs to be an IDE, but you have to learn a bit of Lisp `http://martinsosic.com/development/emacs/2017/12/09/emacs-cpp-ide.html`.

Then there are those of you who like to use your own editors. That is completely fine. For this, we require sshfs which has ports on many different machines.

1. Windows `https://github.com/billziss-gh/sshfs-win`
2. Mac `https://github.com/osxfuse/osxfuse/wiki/SSHFS`
3. Linux `https://help.ubuntu.com/community/SSHFS`

At that point, the files on your VM are synced with the files on your machine and edits can be made and will be synced.

At the time of writing, Bhuvy (That's me!) likes to use spacemacs `http://spacemacs.org/` which marries both vim and emacs and both of their difficulties. I'll give my soapbox for why I like it, but be warned that if you are starting from absolutely no vim or emacs experience the learning curve along with this course may be too much.

1. Extensible. Spacemacs has a clean design written in lisp. There are 100s of packages ready to be installed by editing your spacemacs config and reloading that do everything from syntax checking, automatic static analyzing, etc.
2. Most of the good parts from vim and emacs. Emacs is good at doing everything by being a fast editor. Vim is good at making fast edits and moving around. Spacemacs is the best of both worlds allowing vim keybindings to all the emacs goodness underneath.

3. Lots of preconfiguration done. As opposed with a fresh emacs install, a lot of the configurations with language and projects are done for you like neotree, helm, various language layers. All that you have to do is navigate neotree to the base of your project and emacs will turn into an IDE for that programming language.

But obviously to each his or her own. Many people will argue that editor gurus spend more time editing their editors and actually editing.

Clean Code

Make your code modular using helper functions. If there is a repeated task (getting the pointers to contiguous blocks in the malloc MP, for example), make them helper functions. And make sure each function does one thing well so that you don't have to debug twice. Let's say that we are doing selection sort by finding the minimum element each iteration like so,

```
void selection_sort(int *a, long len){
    for(long i = len-1; i > 0; --i){
        long max_index = i;
        for(long j = len-1; j >= 0; --j){
            if(a[max_index] < a[j]){
                max_index = j;
            }
        }
        int temp = a[i];
        a[i] = a[max_index];
        a[max_index] = temp;
    }

}
```

Many can see the bug in the code, but it can help to refactor the above method into

```
long max_index(int *a, long start, long end);
void swap(int *a, long idx1, long idx2);
void selection_sort(int *a, long len);
```

And the error is specifically in one function. In the end, this class is about writing system programs, not a class about refactoring/debugging your code. In fact, most kernel code is so atrocious that you don't want to read it – the defense there is that it needs to be. But for the sake of debugging, it may benefit you in the long run to adopt some of these practices.

Asserts

Use assertions to make sure your code works up to a certain point – and importantly, to make sure you don't break it later. For example, if your data structure is a doubly-linked list, you can do something like `assert(node == node->next->prev)` to assert that the next node has a pointer to the current node. You can also check the pointer is pointing to an expected range of memory address, non-null, ->size is reasonable, etc. The `DEBUG` macro will disable all assertions, so don't forget to set that once you finish debugging [1].

Here is a quick example with an assert. Let's say that we are writing code using memcpy. We would want to put an assert before that checks whethermy two memory regions overlap. If they do overlap, memcpy runs into undefined behavior, so we want to catch that problem than later.

```
assert(!(src < dest+n && dest < src+n)); //Checks overlap
memcpy(dest, src, n);
```

This check can be turned off at compile-time, but will save you **tons** of trouble debugging!

Valgrind

Valgrind is a suite of tools designed to provide debugging and profiling tools to make your programs more correct and detect some runtime issues [4]. The most used of these tools is Memcheck, which can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behavior (for example, unfreed memory buffers). To run Valgrind on your program:

```
valgrind --leak-check=full --show-leak-kinds=all myprogram arg1
    arg2
```

Arguments are optional and the default tool that will run is Memcheck. The output will be presented in the form: the number of allocations, frees, and errors. Suppose we have a simple program like this:

```
#include <stdlib.h>

void dummy_function() {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;      // error 1: Out of bounds write, as you can see
        here we write to an out of bound memory address.
}                   // error 2: Memory Leak, x is allocated at
    function exit.

int main(void) {
    dummy_function();
    return 0;
```

```
}
```

This program compiles and runs with no errors. Let's see what Valgrind will output.

```
==29515== Memcheck, a memory error detector
==29515== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward
    et al.
==29515== Using Valgrind-3.11.0 and LibVEX; rerun with -h for
    copyright info
==29515== Command: ./a
==29515==
==29515== Invalid write of size 4
==29515==   at 0x400544: dummy_function (in
    /home/rafi/projects/exocpp/a)
==29515==   by 0x40055A: main (in /home/rafi/projects/exocpp/a)
==29515== Address 0x5203068 is 0 bytes after a block of size 40
    alloc'd
==29515==   at 0x4C2DB8F: malloc (in
    /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==29515==   by 0x400537: dummy_function (in
    /home/rafi/projects/exocpp/a)
==29515==   by 0x40055A: main (in /home/rafi/projects/exocpp/a)
==29515==
==29515==
==29515== HEAP SUMMARY:
==29515==    in use at exit: 40 bytes in 1 blocks
==29515==  total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==29515==
==29515== LEAK SUMMARY:
==29515==   definitely lost: 40 bytes in 1 blocks
==29515==   indirectly lost: 0 bytes in 0 blocks
==29515==     possibly lost: 0 bytes in 0 blocks
==29515==   still reachable: 0 bytes in 0 blocks
==29515==        suppressed: 0 bytes in 0 blocks
==29515== Rerun with --leak-check=full to see details of leaked
    memory
==29515==
==29515== For counts of detected and suppressed errors, rerun
    with: -v
==29515== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0
    from 0)
```

Invalid write: It detected our heap block overrun, writing outside of an allocated block.
Definitely lost: Memory leak — you probably forgot to free a memory block.

Valgrind is a effective tool to check for errors at runtime. C is special when it comes to such behavior, so after compiling your program you can use Valgrind to fix errors that your compiler may miss and that usually happens when your program is running.

For more information, you can refer to the manual [4]

TSAN

ThreadSanitizer is a tool from Google, built into clang and gcc, to help you detect race conditions in your code [5]. Note, that running with tsan will slow your code down a bit. Consider the following code.

```
#include <pthread.h>
#include <stdio.h>

int global;

void *Thread1(void *x) {
    global++;
    return NULL;
}

int main() {
    pthread_t t[2];
    pthread_create(&t[0], NULL, Thread1, NULL);
    global = 100;
    pthread_join(t[0], NULL);
}
// compile with gcc -fsanitize=thread -pie -fPIC -ltsan -g
    simple_race.c
```

We can see that there is a race condition on the variable `global`. Both the main thread and created thread will try to change the value at the same time. But, does ThreadSantizer catch it?

```
$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=28888)
  Read of size 4 at 0x7f73ed91c078 by thread T1:
    #0 Thread1 /home/zmick2/simple_race.c:7 (exe+0x000000000a50)
    #1  :0 (libtsan.so.0+0x00000001b459)

  Previous write of size 4 at 0x7f73ed91c078 by main thread:
    #0 main /home/zmick2/simple_race.c:14 (exe+0x000000000ac8)

  Thread T1 (tid=28889, running) created by main thread at:
    #0  :0 (libtsan.so.0+0x00000001f6ab)
```

```
    #1 main /home/zmick2/simple_race.c:13 (exe+0x000000000ab8)

SUMMARY: ThreadSanitizer: data race /home/zmick2/simple_race.c:7
    Thread1
==================
ThreadSanitizer: reported 1 warnings
```

If we compiled with the debug flag, then it would give us the variable name as well.

GDB

GDB is short for the GNU Debugger. GDB is a program that helps you track down errors by interactively debugging them [6]. It can start and stop your program, look around, and put in ad hoc constraints and checks. Here are a few examples.

Setting breakpoints programmatically A breakpoint is a line of code where you want the execution to stop and give control back to the debugger. A useful trick when debugging complex C programs with GDB is setting breakpoints in the source code.

```c
int main() {
    int val = 1;
    val = 42;
    asm("int $3"); // set a breakpoint here
    val = 7;
}
```

```
$ gcc main.c -g -o main
$ gdb --args ./main
(gdb) r
[...]
Program received signal SIGTRAP, Trace/breakpoint trap.
main () at main.c:6
6     val = 7;
(gdb) p val
$1 = 42
```

You can also set breakpoints programmatically. Assume that we have no optimization and the line numbers are as follows

```
1. int main() {
```

```
2.     int val = 1;
3.     val = 42;
4.     val = 7;
5. }
```

We can now set the breakpoint before the program starts.

```
$ gcc main.c -g -o main
$ gdb --args ./main
(gdb) break main.c:4
[...]
(gdb) p val
$1 = 42
```

Checking memory content We can also use gdb to check the content of different pieces of memory. For example,

```
int main() {
    char bad_string[3] = {'C', 'a', 't'};
    printf("%s", bad_string);
}
```

Compiled we get

```
$ gcc main.c -g -o main && ./main
$ Cat ZVQ- $
```

We can now use gdb to look at specific bytes of the string and reason about when the program should've stopped running

```
(gdb) l
1 #include <stdio.h>
2 int main() {
3     char bad_string[3] = {'C', 'a', 't'};
4     printf("%s", bad_string);
5 }
(gdb) b 4
Breakpoint 1 at 0x100000f57: file main.c, line 4.
(gdb) r
[...]
```

```
Breakpoint 1, main () at main.c:4
4     printf("%s", bad_string);
(gdb) x/16xb bad_string
0x7fff5fbff9cd: 0x63 0x61 0x74 0xe0 0xf9  0xbf  0x5f  0xff
0x7fff5fbff9d5: 0x7f 0x00 0x00 0xfd 0xb5  0x23  0x89  0xff
(gdb)
```

Here, by using the `x` command with parameters `16xb`, we can see that starting at memory address `0x7fff5fbff9c` (value of `bad_string`), `printf` would actually see the following sequence of bytes as a string because we provided a malformed string without a null terminator.

Involved gdb example

Here is how one of your TAs would go through and debug a simple program that is going wrong. First, the source code of the program. If you can see the error immediately, please bear with us.

```c
#include <stdio.h>

double convert_to_radians(int deg);

int main(){
    for (int deg = 0; deg > 360; ++deg){
        double radians = convert_to_radians(deg);
        printf("%d. %f\n", deg, radians);
    }
    return 0;
}

double convert_to_radians(int deg){
    return ( 31415 / 1000 ) * deg / 180;
}
```

How can we use gdb to debug? First we ought to load GDB.

```
$ gdb --args ./main
(gdb) layout src; # If you want a GUI type
(gdb) run
(gdb)
```

Want to take a look at the source?

```
(gdb) l
1   #include <stdio.h>
2
3   double convert_to_radians(int deg);
4
5   int main(){
6       for (int deg = 0; deg > 360; ++deg){
7           double radians = convert_to_radians(deg);
8           printf("%d. %f\n", deg, radians);
9       }
10      return 0;
(gdb) break 7 # break <file>:line or break <file>:function
(gdb) run
(gdb)
```

From running the code, the breakpoint didn't even trigger, meaning the code never got to that point. Thats because of the comparison! Okay, flip the sign it should work now right?

```
(gdb) run
350. 60.000000
351. 60.000000
352. 60.000000
353. 60.000000
354. 60.000000
355. 61.000000
356. 61.000000
357. 61.000000
358. 61.000000
359. 61.000000
```

```
(gdb) break 14 if deg == 359 # Let's check the last iteration only
(gdb) run
...
(gdb) print/x deg # print the hex value of degree
$1 = 0x167
(gdb) print (31415/1000)
$2 = 0x31
(gdb) print (31415/1000.0)
$3 = 201.749
(gdb) print (31415.0/10000.0)
$4 = 3.1414999999999999
```

That was only the bare minimum, though most of you will get by with that. There are a whole load more resources on the web, here are a few specific ones that can help you get started.

1. Introduction to gdb
2. Memory Content
3. CppCon 2015: Greg Law Give me 15 minutes I'll change your view of GDB

Shell

What do you actually use to run your program? A shell! A shell is a programming language that is running inside your terminal. A terminal is merely a window to input commands. Now, on POSIX we usually have one shell called `sh` that is linked to a POSIX compliant shell called `dash`. Most of the time, you use a shell called `bash` that is somewhat POSIX compliant but has some nifty built-in features. If you want to be even more advanced, `zsh` has some more powerful features like tab complete on programs and fuzzy patterns.

Undefined Behavior Sanitizer

The undefined behavior sanitizer is a wonderful tool provided by the llvm project. It allows you to compile code with a runtime checker to make sure that you don't do undefined behavior for various categories. We will try to include it into our projects, but requires support form all the external libraries that we use so we may not get around to all of them. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

Undefined behavior - why we can't solve it in general

Also please please read Chris Lattner's 3 Part blog post on undefined behavior. It can shed light on debug builds and the mystery of compiler optimization.

`http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html`

Clang Static Build Tools

Clang provides a great drop-in replacement tools for compiling programs. If you want to see if there is an error that may cause a race condition, casting error, etc, all you need to do is the following.

```
$ scan-build make
```

And in addition to the make output, you will get static build warnings.

strace and ltrace

strace and ltrace are two programs that trace the system calls and library calls respectively of a running program or command. These may be missing on your system so to install feel free to run the following.

```
$ sudo apt install strace ltrace
```

Debugging with ltrace can be as simple as figuring out what was the return call of the last library call that failed.

```
int main() {
    FILE *fp = fopen("I don't exist", "r");
    fprintf(fp, "a");
    fclose(fp);
    return 0;
}
```

```
> ltrace ./a.out
 __libc_start_main(0x8048454, 1, 0xbfc19db4, 0x80484c0, 0x8048530
     <unfinished ...>
 fopen("I don't exist", "r")                    = 0x0
 fwrite("Invalid Write\n", 1, 14, 0x0 <unfinished ...>
 --- SIGSEGV (Segmentation fault) ---
 ++++ killed by SIGSEGV ++++
```

ltrace output can clue you in to weird things your program is doing live. Unfortunately, ltrace cant be used to inject faults, meaning that ltrace can tell you what is happening, but it can't tamper with what is already happening.

strace on the other hand could modify your program. Debugging with strace is amazing. The basic usage is running strace with a program, and itll get you a complete list of system call parameters.

```
$ strace head README.md
execve("/usr/bin/head", ["head", "README.md"], 0x7ffff28c8fa8 /*
    60 vars */) = 0
brk(NULL)                               = 0x7ffff5719000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
    directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
    directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=32804, ...}) = 0
...
```

If the output is too verbose, you can use trace= with a command delimited list of syscalls to filter all but those calls.

```
$ strace -e trace=read,write head README.md
read(3,
    "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"...,
    832) = 832
read(3, "# Locale name alias data base.\n#"..., 4096) = 2995
read(3, "", 4096)                       = 0
read(3, "# C Datastructures\n\n[![Build Sta"..., 8192) = 1250
write(1, "# C Datastructures\n", 19# C Datastructures
```

You can also trace files or targets.

```
$ strace -e trace=read,write -P README.md head README.md
strace: Requested path 'README.md' resolved into
    '/mnt/c/Users/bhuvy/personal/libds/README.md'
read(3, "# C Datastructures\n\n[![Build Sta"..., 8192) = 1250
```

Newer versions of strace can actually inject faults into your program. This is useful when you want to occasionally make reads and writes fail for example in a networking application, which your program should handle. The problem is as of early 2019, that version is missing from Ubuntu repositories. Meaning that you'll have to install it from the source.

printfs

When all else fails, print! Each of your functions should have an idea of what it is going to do. You want to test that each of your functions is doing what it set out to do and see exactly where your code breaks. In the case with race conditions, tsan may be able to help, but having each thread print out data at certain times could help you identify the race condition.

To make printfs useful, try to have a macro that fills in the context by which the printf was called – a log statement if you will. A simple useful but untested log statement could be as follows. Try to make a test and figure out something that is going wrong, then log the state of your variables.

```c
#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <unistd.h>

// bt is print backtrace
const int num_stack = 10;
```

```
int __log(int line, const char *file, int bt, const char *fmt,
    ...) {
  if (bt) {
    void *raw_trace[num_stack];
    size_t size = backtrace(raw_trace, sizeof(raw_trace) /
        sizeof(raw_trace[0]));
    char **syms = backtrace_symbols(raw_trace, size);

    for(ssize_t i = 0; i < size; i++) {
      fprintf(stderr, "|%s:%d| %s\n", file, line, syms[i]);
    }
    free(syms);
  }
  int ret = fprintf(stderr, "|%s:%d| ", file, line);
  va_list args;
  va_start(args, fmt);
  ret += vfprintf(stderr, fmt, args);
  va_end(args);
  ret += fprintf(stderr, "\n");
  return ret;
}

#ifdef DEBUG
#define log(...) __log(__LINE__, __FILE__, 0, __VA_ARGS__)
#define bt(...) __log(__LINE__, __FILE__, 1, __VA_ARGS__)
#else
#define log(...)
#define bt(...)
#endif

//Use as log(args like printf) or bt(args like printf) to either
    log or get backtrace

int main() {
  log("Hello Log");
  bt("Hello Backtrace");
}
```

And then use as appropriately.

Extra: Compiling and Linking

This is a high-level overview from the time you compile your program to the time you run your program. We often know that compiling your program is easy. You run the program through an IDE or a terminal, and it just works.

```
$ cat main.c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
$ gcc main.c -o main
$ ./main
Hello World!
$
```

Here are the rough stages of compiling for gcc.

1. Preprocessing: The preprocessor expands all preprocesor directives.
2. Parsing: The compiler parses the text file for function declarations, variable declarations, etc.
3. Assembly Generation: The compiler then generates assembly code for all the functions after some optimizations if enabled.
4. Assembling: The assembler turns the assembly into 0s and 1s and creates an object file. This object file maps names to pieces of code.
5. Static Linking: The linker then takes a series of objects and static libraries and resolves references of variables and functions from one object file to another. The linker then finds the main method and makes that the entry point for the function. The linker also notices when a function is meant to be dynamically linked. The compiler also creates a section in the executable that tells the operating system that these functions need addresses right before running.
6. Dynamic Linking: As the program is getting ready to be executed, the operating system looks at what libraries that the program needs and links those functions to the dynamic library.
7. The program is run.

Further classes will teach you about parsing and assembly – preprocessing is an extension of parsing. Most classes won't teach you about the two different types of linking though. Static linking a library is similar to combining object files. To create a static library, a compiler combines different object files to create one executable. A static library is literally is an archive of object files. These libraries are useful when you want your executable to be secure, you know all the code that is being included into your executable, and portable, all the code is bundled with your executable meaning no additional installs.

The other type is a dynamic library. Typically, dynamic libraries are installed user-wide or system-wide and are accessible by most programs. Dynamic libraries' functions are filled in right before they are run. There are a number of benefits to this.

- Lower code footprint for common libraries like the C standard library
- Late binding means more generalized code and less reliance on specific behavior.
- Differentiation means that the shared library can be updated while keeping the executable the same.

There are a number of drawbacks as well.

- All the code is no longer bundled into your program. This means that users have to install something else.
- There could be security flaws in the other code leading to security exploits in your program.
- Standard Linux allows you to "replace" dynamic libraries, leading to possible social engineering attacks.
- This adds additional complexity to your application. Two identical binaries with different shared libraries could lead to different results.

Homework 0

```
// First, can you guess which lyrics have been transformed into
    this C-like system code?
char q[] = "Do you wanna build a C99 program?";
#define or "go debugging with gdb?"
static unsigned int i = sizeof(or) != strlen(or);
char* ptr = "lathe";
size_t come = fprintf(stdout,"%s door", ptr+2);
int away = ! (int) * "";

int* shared = mmap(NULL, sizeof(int*), PROT_READ | PROT_WRITE,
    MAP_SHARED | MAP_ANONYMOUS, -1, 0);
munmap(shared,sizeof(int*));

if(!fork()) {
    execlp("man","man","-3","ftell", (char*)0); perror("failed");
}

if(!fork()) {
    execlp("make","make", "snowman", (char*)0);
        execlp("make","make", (char*)0));
}

exit(0);
```

So you want to master System Programming? And get a better grade than B?

m

```
int main(int argc, char** argv) {
    puts("Great! We have plenty of useful resources for you, but
        it's up to you to");
    puts(" be an active learner and learn how to solve problems and
        debug code.");
    puts("Bring your near-completed answers the problems below");
    puts(" to the first lab to show that you've been working on
        this.");
    printf("A few \"don't knows\" or \"unsure\" is fine for lab
        1.\n");
```

```
    puts("Warning: you and your peers will work hard in this
        class.");
    puts("This is not CS225; you will be pushed much harder to");
    puts(" work things out on your own.");
    fprintf(stdout,"This homework is a stepping stone to all future
        assignments.\n");
    char p[] = "So, you will want to clear up any confusions or
        misconceptions.\n";
    write(1, p, strlen(p) );
    char buffer[1024];
    sprintf(buffer,"For grading purposes, this homework 0 will be
        graded as part of your lab %d work.\n", 1);
    write(1, buffer, strlen(buffer));
    printf("Press Return to continue\n");
    read(0, buffer, sizeof(buffer));
    return 0;
}
```

Watch the videos and write up your answers to the following questions

Important!

The virtual machine-in-your-browser and the videos you need for HW0 are here:
`http://cs-education.github.io/sys/`

Questions? Comments? Use the current semester's CS241 Piazza: `https://piazza.com/`

The in-browser virtual machine runs entirely in JavaScript and is fastest in Chrome. Note the VM and any code you write is reset when you reload the page, **so copy your code to a separate document.** The post-video challenges are not part of homework 0, but you learn the most by doing rather than passively watching. You have some fun with each end-of-video challenge.

HW0 questions are below. Copy your answers into a text document because you'll need to submit them later in the course.

Chapter 1

In which our intrepid hero battles standard out, standard error, file descriptors and writing to files

1. **Hello, World! (system call style)** Write a program that uses `write()` to print out "Hi! My name is <Your Name>".

2. **Hello, Standard Error Stream!** Write a function to print out a triangle of height `n` to standard error. Your function should have the signature `void write_triangle(int n)` and should use `write()`. The triangle should look like this, for n = 3:

```
*
**
***
```

3. **Writing to files** Take your program from "Hello, World!" modify it write to a file called `hello_world.txt`. Make sure to use correct flags and a correct mode for `open()` (`man 2 open` is your friend).
4. **Not everything is a system call** Take your program from "Writing to files" and replace `write()` with `printf()`. *Make sure to print to the file instead of standard out!*
5. What are some differences between `write()` and `printf()`?

Chapter 2

Sizing up C types and their limits, `int` and `char` arrays, and incrementing pointers

1. How many bits are there in a byte?
2. How many bytes are there in a `char`?
3. How many bytes the following are on your machine? `int`, `double`, `float`, `long`, and `long long`
4. On a machine with 8 byte integers, the declaration for the variable `data` is `int data[8]`. If the address of data is `0x7fbd9d40`, then what is the address of `data+2`?
5. What is `data[3]` equivalent to in C? Hint: what does C convert `data[3]` to before dereferencing the address? Remember, the type of a string constant `"abc"` is an array.
6. Why does this SEGFAULT?

```
char *ptr = "hello";
*ptr = 'J';
```

7. What is the value of the variable `str_size`?

```
ssize_t str_size = sizeof("Hello\0World")
```

8. What is the value of the variable `str_len`

```
ssize_t str_len = strlen("Hello\0World")
```

9. Give an example of X such that `sizeof(X)` is 3.
10. Give an example of Y such that `sizeof(Y)` might be 4 or 8 depending on the machine.

Chapter 3

Program arguments, environment variables, and working with character arrays (strings)

1. What are at least two ways to find the length of `argv`?
2. What does `argv[0]` represent?
3. Where are the pointers to environment variables stored (on the stack, the heap, somewhere else)?
4. On a machine where pointers are 8 bytes, and with the following code:

```
char *ptr = "Hello";
char array[] = "Hello";
```

   What are the values of `sizeof(ptr)` and `sizeof(array)`? Why?

5. What data structure manages the lifetime of automatic variables?

Chapter 4

Heap and stack memory, and working with structs

1. If I want to use data after the lifetime of the function it was created in ends, where should I put it? How do I put it there?
2. What are the differences between heap and stack memory?
3. Are there other kinds of memory in a process?
4. Fill in the blank: "In a good C program, for every malloc, there is a ___".
5. What is one reason `malloc` can fail?
6. What are some differences between `time()` and `ctime()`?
7. What is wrong with this code snippet?

```
free(ptr);
free(ptr);
```

8. What is wrong with this code snippet?

```
free(ptr);
printf("%s\n", ptr);
```

9. How can one avoid the previous two mistakes?
10. Create a `struct` that represents a `Person`. Then make a `typedef`, so that `struct Person` can be replaced with a single word. A person should contain the following information: their name (a string), their age (an integer), and a list of their friends (stored as a pointer to an array of pointers to `Person`s).
11. Now, make two persons on the heap, "Agent Smith" and "Sonny Moore", who are 128 and 256 years old respectively and are friends with each other. Create functions to create and destroy a Person (Person's and their names should live on the heap).
12. `create()` should take a name and age. The name should be copied onto the heap. Use malloc to reserve sufficient memory for everyone having up to ten friends. Be sure initialize all fields (why?).
13. `destroy()` should free up both the memory of the person struct and all of its attributes that are stored on the heap. Destroying one person keeps other people in tact any other.

Chapter 5

Text input and output and parsing using `getchar`, `gets`, and `getline`.

1. What functions can be used for getting characters from `stdin` and writing them to `stdout`?
2. Name one issue with `gets()`.
3. Write code that parses the string "Hello 5 World" and initializes 3 variables to "Hello", 5, and "World".
4. What does one need to define before including `getline()`?
5. Write a C program to print out the content of a file line-by-line using `getline()`.

C Development

These are general tips for compiling and developing using a compiler and git. Some web searches will be useful here

1. What compiler flag is used to generate a debug build?
2. You fix a problem in the Makefile and type `make` again. Explain why this may be insufficient to generate a new build.
3. Are tabs or spaces used to indent the commands after the rule in a Makefile?
4. What does `git commit` do? What's a `sha` in the context of git?
5. What does `git log` show you?
6. What does `git status` tell you and how would the contents of `.gitignore` change its output?
7. What does `git push` do? Why is it insufficient to commit with `git commit -m 'fixed all bugs'` ?
8. What does a non-fast-forward error `git push` reject mean? What is the most common way of dealing with this?

Optional: Just for fun

- Convert a song lyrics into System Programming and C code covered in this wiki book and share on Piazza.
- Find, in your opinion, the best and worst C code on the web and post the link to Piazza.
- Write a short C program with a deliberate subtle C bug and post it on Piazza to see if others can spot your bug.
- Do you have any cool/disastrous system programming bugs you've heard about? Feel free to share with your peers and the course staff on Piazza.

UIUC Specific Guidelines

Piazza

TAs and student assistants get a ton of questions. Some are well-researched, and some are not. This is a handy guide that'll help you move away from the latter and towards the former. Oh, and did I mention that this is an easy way to score points with your internship managers? Ask yourself...

1. Am I running on my Virtual Machine?
2. **Did I check the man pages?**
3. Have I searched for similar questions/followups on Piazza?
4. Have I read the MP/Lab specification completely?
5. Have I watched all of the videos?
6. Did I Google the error message and a few permutations thereof if necessary? How about StackOverflow.
7. Did I try commenting out, printing, and/or stepping through parts of the code bit by bit to find out precisely where the error occurs?
8. **Did I commit my code to git in case the TAs need more context?**
9. Did I include the console/GDB/Valgrind output **AND** code surrounding the bug in my Piazza post?
10. Have I fixed other segmentation faults unrelated to the issue I'm having?
11. Am I following good programming practice? (i.e. encapsulation, functions to limit repetition, etc)

The biggest tip that we can give you when asking a question on piazza if you want a swift answer is to **ask your question like you were trying to answer it**. Like before you ask a question, try to answer it yourself. If you are thinking about posting

> Hi, My code got a 50

Sounds good and courteous, but course staff would much much prefer a post resembling the following

> Hi, I recently failed test X, Y, Z which is about half the tests on this current assignment. I noticed that they all have something to do with networking and epoll, but couldn't figure out what was linking them together, or I may be completely off track. So to test my idea, I tried spawning 1000 clients with various get and put requests and verifying the files matched their originals. I couldn't get it to fail while running normally, the debug build, or valgrind or tsan. I have no warnings and none of the pre-syntax checks showed me anything. Could you tell me if my understanding of the failure is correct and what I could do to modify my tests to better reflect X, Y, Z? netid: bvenkat2

You don't need to be as courteous, though we'd appreciate it, this will get a faster response time hand over foot. If you were trying to answer this question, you'd have everything you need in the question body.

Bibliography

[1] assert. URL `http://www.cplusplus.com/reference/cassert/assert/`.

[2] ssh(1). URL `https://man.openbsd.org/ssh.1`.

[3] Chapter 3. hardware interrupts. URL `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/1.3/html/Realtime_Reference_Guide/chap-Realtime_Reference_Guide-Hardware_interrupts.html`.

[4] 4. memcheck: a memory error detector. URL `http://valgrind.org/docs/manual/mc-manual.html`.

[5] Threadsanitizercppmanual, Dec 2018. URL `https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual`.

[6] Gdb: The gnu project debugger, Feb 2019. URL `https://www.gnu.org/software/gdb/`.

[7] Manu Garg. Sysenter based system call mechanism in linux 2.6, 2006. URL `http://articles.manugarg.com/systemcallinlinux2_6.html`.

[8] Part Guide. Intel® 64 and ia-32 architectures software developers manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[9] CAT Intel. Improving real-time performance by utilizing cache allocation technology. *Intel Corporation, April*, 2015.

[10] Xavier Leroy. How i found a bug in intel skylake processors, Jul 2017. URL `http://gallium.inria.fr/blog/intel-skylake-bug/`.

[11] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18, 2009.

[12] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456. IEEE, 2015.

[13] Wikibooks. X86 assembly — wikibooks, the free textbook project, 2018. URL `https://en.wikibooks.org/w/index.php?title=X86_Assembly&oldid=3477563`. [Online; accessed 19-March-2019].