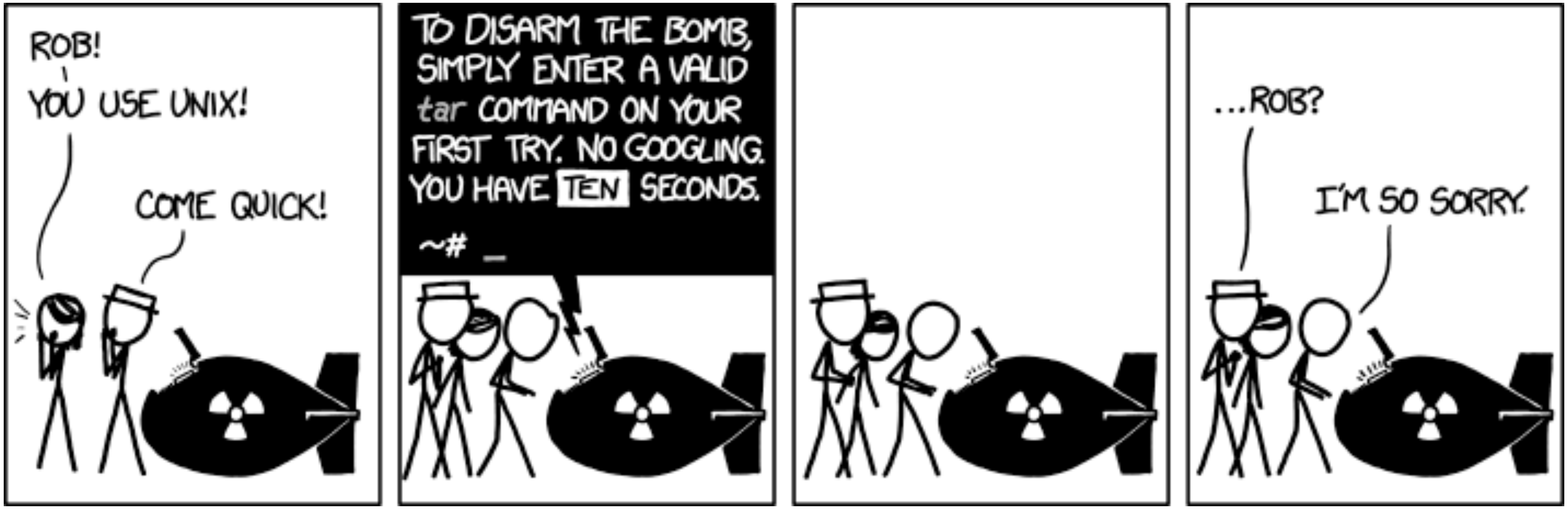
CIS 330:

Unix and C/C++

Lab: ssh, scp, gdb, valgrind



xkcd 1168

Unix command: ssh

- Problem: you're using a computer, but you want to be using a different computer …
  - the other computer is far away
  - the other computer is inaccessible
  - the other computer doesn't have a display (server)
  - etc.

- ssh lets you log onto another machine

Unix command: ssh

Basic Use:

- ssh username@machine

(equivalent version using the -l flag)

- ssh -l username machine
- -Y flag: Enables X11 forwarding ➔ Remote use of GUI applications

DEMO

Unix command: ssh

From demo: don't need to type username / machine name / password every time!

- Instructions for accomplishing this could be confusing since there are potentially different steps for different systems … ask after class or come to office hours if interested.

Unix command: scp

- Problem: you have files on one computer, but you want those files on a different computer ...
- scp lets you send files from one machine to another machine

Unix command: scp

Basic Use:

- scp source destination
- either source or destination might be a remote machine … examples:
  - scp my_file username@ix-dev.cs.uoregon.edu:~

    (this copies my_file in the current working directory to HOME directory on ix-dev)
  - scp username@ix-dev.cs.uoregon.edu:/absolute/path/my_other_file .

    (this copies my_other_file in /absolute/path on ix-dev to the current working directory)
  - nothing special about these examples ... DEMO

Debugging

- Problem: you wrote a computer program and it doesn't work ...

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

int main(void){
        int my_variable;
        printf("%d\n", 10 / my_variable);
}
~
~
                                                  1,1           All
```

Debugging

- Worse problem: someone else wrote a computer program and it doesn't work ...

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

int main(void){
        int my_variable = 2;
        int A[8] = {0,0,0,0,0,0,0,0};

        /* a billion lines of code */

        my_variable <<= 85 / 27 + 1;

        /* another billion lines of code */

        int x = A[my_variable];
}
~
                                              1,1           All
```

Debugging: lots of printf

- Method #1: just print everything and figure it out … this works pretty good most of the time!

```c
#include <stdio.h>
#include <stdlib.h>

int main(void){
        int my_variable = 2;
        int A[8] = {0,0,0,0,0,0,0,0};

        /* a billion lines of code */

        my_variable <<= 85 / 27 + 1;

        /* another billion lines of code */

        printf("%d\n", my_variable);
        int x = A[my_variable];
}
~
                                            1,1           All
```

Debugging: lots of printf

- Method #1: sometimes you are in a tough spot!

```c
#include <stdio.h>
#include <stdlib.h>

int my_func(int *p){
        *(p+7) += 100;
        *(p+7) %= 200;
        return 7;
}

int main(void){
        int my_variable = 2;
        int A[8] = {0,0,0,0,0,0,0,0};

        /* a billion lines of code */

        my_variable <<= 85 / 27 + 1;

        /* another billion lines of code */

        printf("%d\n", my_variable);
        int x = A[my_variable];

        int y = A[A[my_func(A)]];
        printf("%d\n", y);
}
~
~
                                                  1,1           All
```

when I run this, I get the value 1661289645 for y

Debugging: lots of printf

- Method #1: sometimes you are in a tough spot!

```c
#include <stdio.h>
#include <stdlib.h>

int my_func(int *p){
        *(p+7) += 100;
        *(p+7) %= 200;
        return 7;
}

int main(void){
        int my_variable = 2;
        int A[8] = {0,0,0,0,0,0,0,0};

        /* a billion lines of code */

        my_variable <<= 85 / 27 + 1;

        /* another billion lines of code */

        printf("%d\n", my_variable);
        int x = A[my_variable];

        printf("%d\n", my_func(A));
        int y = A[A[my_func(A)]];
        printf("%d\n", y);
}
~
                                        1,1           All
```

when I run this, I get 7 for the return value of my_func … but now y is 0???

Debugging: lots of printf

- Method #1: sometimes you are in a tough spot!

```c
#include <stdio.h>
#include <stdlib.h>

int my_func(int *p){
        *(p+7) += 100;
        *(p+7) %= 200;
        return 7;
}

int main(void){
        int my_variable = 2;
        int A[8] = {0,0,0,0,0,0,0,0};

        /* a billion lines of code */

        my_variable <<= 85 / 27 + 1;

        /* another billion lines of code */

        printf("%d\n", my_variable);
        int x = A[my_variable];

        printf("%d\n", my_func(A));
        int y = A[A[my_func(A)]];
        printf("%d\n", y);
}
~
                                                  1,1           All
```

This example is kind of contrived … a more typical situation (for me, at least) is that I'm reading some code and it's completely mind boggling, and putting in print statements would just take a really long time.

Debugging: gdb

More options would be great!

- What are all the local variables defined at some point in the program?
- What are the values of each variable?
- What happens if we change the value of a variable?

Method #2: gdb can do all of this. And much more!

Debugging: gdb

Method #2: gdb

- Can inspect and modify code as it runs without recompiling!
- Similar program called lldb on macOS
- Run from the command line, but need to compile with debug info (-g flag). Example:
  - Compile: gcc -g -o bad incorrect_program.c
  - Run: gdb ./bad

Debugging: gdb

DEMO: I'll be switching over to Ubuntu for this…

- Newer macOSX versions stopped supporting gdb
  - Encourage the use of comparable lldb
  - "brew install gdb" still there…but errors may occur while using gdb
- Recommend: use lldb on Mac; gdb on Linux

Debugging: gdb

```
(gdb) break 23
Breakpoint 1 at 0x400651: file test.c, line 23.
(gdb) run
Starting program: /home/awh/Dropbox/uo_courses/330/test/goofin
/bad
32

Breakpoint 1, main () at test.c:23
23              printf("%d\n", my_func(A));
(gdb) print *A@8
$1 = {0, 0, 0, 0, 0, 0, 0, 0}
(gdb) next
7
24              int y = A[A[my_func(A)]];
(gdb) print *A@8
$2 = {0, 0, 0, 0, 0, 0, 0, 100}
(gdb) next
25              printf("%d\n", y);
(gdb) print *A@8
$3 = {0, 0, 0, 0, 0, 0, 0, 0}
(gdb) 
```

Example gdb session working with the previous example program.

DEMO

These gdb commands, and more, explained on next slide.

Debugging: gdb

Useful commands in gdb:

- break N: set breakpoint at line N
- break my_func: break whenever my_func is called
- watch my_var: break whenever my_var is changed
- run: start the program
- continue: go until the next breakpoint
- next: do the next line of code
- step: do the next line of code, descending into function calls
- info locals: display local variable information
- backtrace: show frames leading to crash
- print x: print the value of variable x
- print *A@N: print the first N values of array A
- set var x=v: set the value of variable x to v

lldb/gdb comparison commands:
https://developer.apple.com/library/content/documentation/IDEs/Conceptual/gdb_to_lldb_transition_guide/document/lldb-command-examples.html#//apple_ref/doc/uid/TP40012917-CH3-SW3

Debugging: valgrind

Method #3: valgrind

- Need to compile with debug info (-g flag). Example:
  - Compile: gcc -g -o bad incorrect_program.c
  - Run: valgrind ./bad
- Might not be installed by default on macOS.
  - Install with homebrew (brew install valgrind)
  - Run on ix-dev (already installed)
  - If using Ubuntu: sudo apt-get install valgrind

Debugging: valgrind

```c
#include <stdlib.h>
#include <stdio.h>

int main(void){
    int X[3] = {1,2,3};
    int w = X[4];
    X[5] = 6;
    int q = X[-100];

    int *Y = malloc(sizeof(int) * 5);
    Y[0] = 1;
    Y[1] = 2;
    Y[2] = 3;
    int r = Y[4];
    Y[5] = 6;

}
~
                                                  1,1           All
```

Valgrind finds only a certain type of error: memory errors. This is great, though! These errors can be really tough. Let's try finding the memory errors in this program using valgrind.

Debugging: valgrind

```
==7410== Memcheck, a memory error detector
==7410== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==7410== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==7410== Command: ./bad
==7410==
==7410== Invalid read of size 4
==7410==    at 0x4005CF: main (mem_err.c:8)
==7410==  Address 0xffefffc50 is on thread 1's stack
==7410==  368 bytes below stack pointer
==7410==
==7410== Invalid write of size 4
==7410==    at 0x40061E: main (mem_err.c:15)
==7410==  Address 0x5203054 is 0 bytes after a block of size 20 alloc'd
==7410==    at 0x4C2DB8F: malloc (vg_replace_malloc.c:299)
==7410==    by 0x4005E1: main (mem_err.c:10)
==7410==
```

DEMO: valgrind ./bad

Debugging: valgrind

```c
#include <stdlib.h>
#include <stdio.h>

int main(void){
    int X[3] = {1,2,3};
    int w = X[4];       /* valgrind can struggle with memory errors */
    X[5] = 6;           /* on the stack a little bit ...            */
    int q = X[-100];    /* had to work to find this one!            */

    int *Y = malloc(sizeof(int) * 5);
    Y[0] = 1;
    Y[1] = 2;
    Y[2] = 3;
    int r = Y[4];   /* misses the uninitialized memory read */
    Y[5] = 6;       /* but easily catches the invalid write */

}
~
                                                          1,1           All
```

Debugging: valgrind

What about the other output? Valgrind tells us that there is a "memory leak" … memory allocated on the heap that was never freed. A memory leak isn't great because the program is unable to re-use that memory, perhaps leading it to exhaust the available memory. You need to make your projects leak free!

```
==7410==    by 0x4005E1: main (mem_err.c:10)
==7410==
==7410==
==7410== HEAP SUMMARY:
==7410==     in use at exit: 20 bytes in 1 blocks
==7410==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==7410==
==7410== LEAK SUMMARY:
==7410==    definitely lost: 20 bytes in 1 blocks
==7410==    indirectly lost: 0 bytes in 0 blocks
==7410==      possibly lost: 0 bytes in 0 blocks
==7410==    still reachable: 0 bytes in 0 blocks
==7410==         suppressed: 0 bytes in 0 blocks
==7410== Rerun with --leak-check=full to see details of leaked memory
==7410==
==7410== For counts of detected and suppressed errors, rerun with: -v
```