CMPUT-379 Lab 1

Akemi Izuko, Armaan Katyal, Ellis McDougald, Han Yang, Patrick Zijlstra, Shasta Johnsen-Sollos, Steven Oufan Hai, Zhaoyu Li

Today’s lab

- Get familiar with the programming environment
    - SSH and Git
- Basic steps for Unix system programming
    - Makefiles
    - GDB and Valgrind
    - Man pages
    - Utilities and commands
        - `objdump`, `nm`, `strace`, `ps`, `pstree`, `top`, `watch`, `kill`
    - Debugging:
        - errno & perror
- System calls for process management
    - fork( ), execve( ), getpid( ), wait( ), waitpid( ), exit( )

Getting started with the programming environment

CS Linux Machines

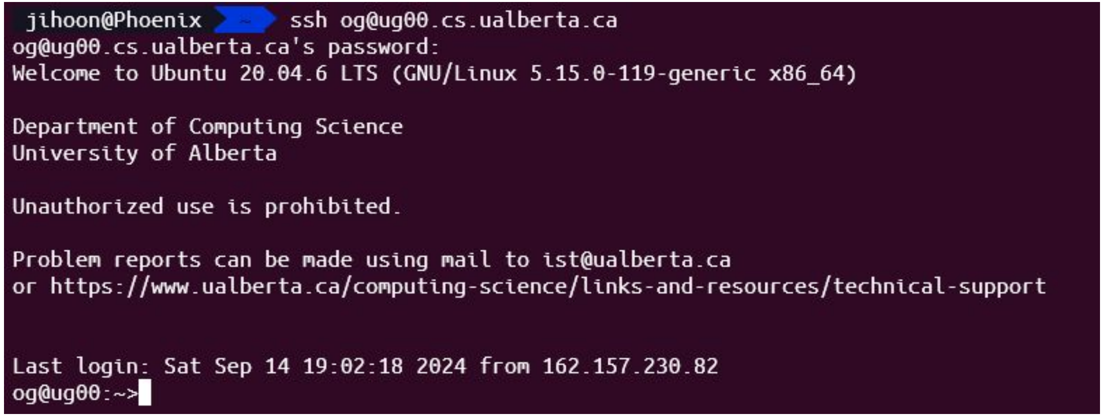
- Make sure your programs can compile and run on these machines
- You must connect to the ualberta vpn to access these, or proxyjump through login.cs.ualberta.ca (advanced)
- **Assignments will be graded on these machines**

Lab machines:

ucomm-2030-wXX.cs.ualberta.ca
(XX must be between 01 and 04 inclusive)
ucomm-2070-wXX.cs.ualberta.ca
(XX must be between 00 and 24 inclusive)
ucomm-2086-wXX.cs.ualberta.ca
(XX must be between 00 and 33 inclusive)
ucomm-2130-wXX.cs.ualberta.ca
(XX must be between 00 and 25 inclusive)
ucomm-2140-wXX.cs.ualberta.ca
(XX must be between 00 and 25 inclusive)
ucomm-3130-wXX.cs.ualberta.ca
(XX must be between 00 and 23 inclusive)
ucomm-3140-wXX.cs.ualberta.ca
(XX must be between 00 and 21 inclusive)

Connect to the CS Linux machines with ssh

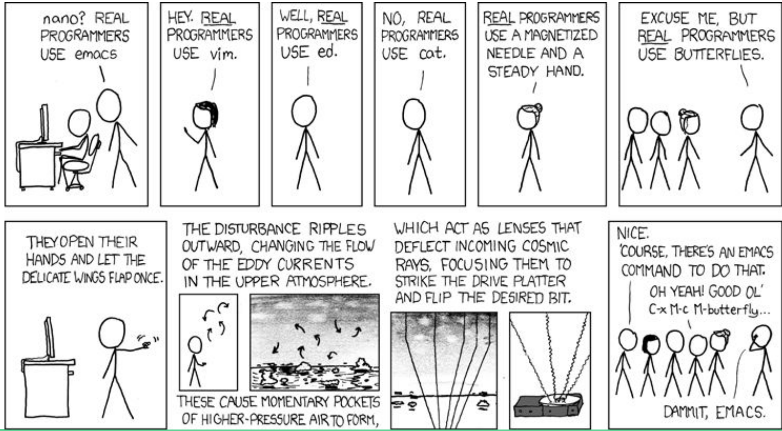
- Open your terminal app
- Run ssh CCID@ucomm-2070-wXX.cs.ualberta.ca
- Enter your password for you CCID
- exit or Ctrl-D to exit
- For Windows you can use PuTTY/MobaXTerm/WSL terminal



```
 jihoon@Phoenix  ~  ssh og@ug00.cs.ualberta.ca
og@ug00.cs.ualberta.ca's password:
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.15.0-119-generic x86_64)

Department of Computing Science
University of Alberta

Unauthorized use is prohibited.

Problem reports can be made using mail to ist@ualberta.ca
or https://www.ualberta.ca/computing-science/links-and-resources/technical-support


Last login: Sat Sep 14 19:02:18 2024 from 162.157.230.82
og@ug00:~>
```

Editing files on remote machines

- Terminal text editors like `vim`, `emacs`, `nano`
- VSCode Remote Development
  - Modern GUI application with commonly used keybinds
  - Runs on your own machine
  - Manages and edit your remote files through SSH

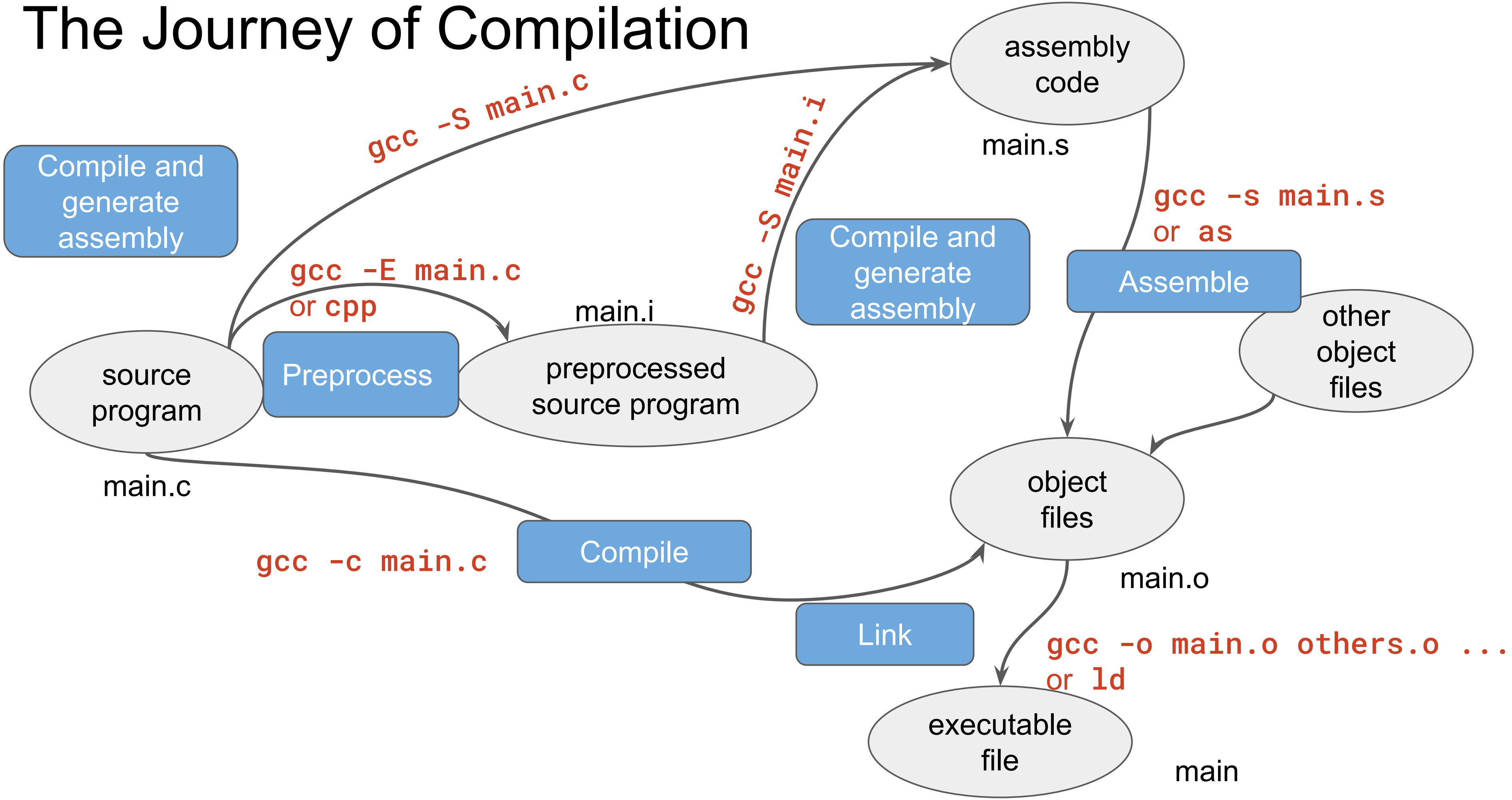

nano? REAL PROGRAMMERS USE emacs
HEY. REAL PROGRAMMERS USE vim.
WELL, REAL PROGRAMMERS USE ed.
NO, REAL PROGRAMMERS USE cat.
REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.
EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.
THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.
THE DISTURBANCE RIPPLES OUTWARD, CHANGING THE FLOW OF THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.
THESE CAUSE MOMENTARY POCKETS OF HIGHER-PRESSURE AIR TO FORM,
WHICH ACT AS LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.
NICE.
'COURSE, THERE'S AN EMACS COMMAND TO DO THAT.
OH YEAH! GOOD OL' C-x M-c M-butterfly...
DAMMIT, EMACS.

Quick git tutorial

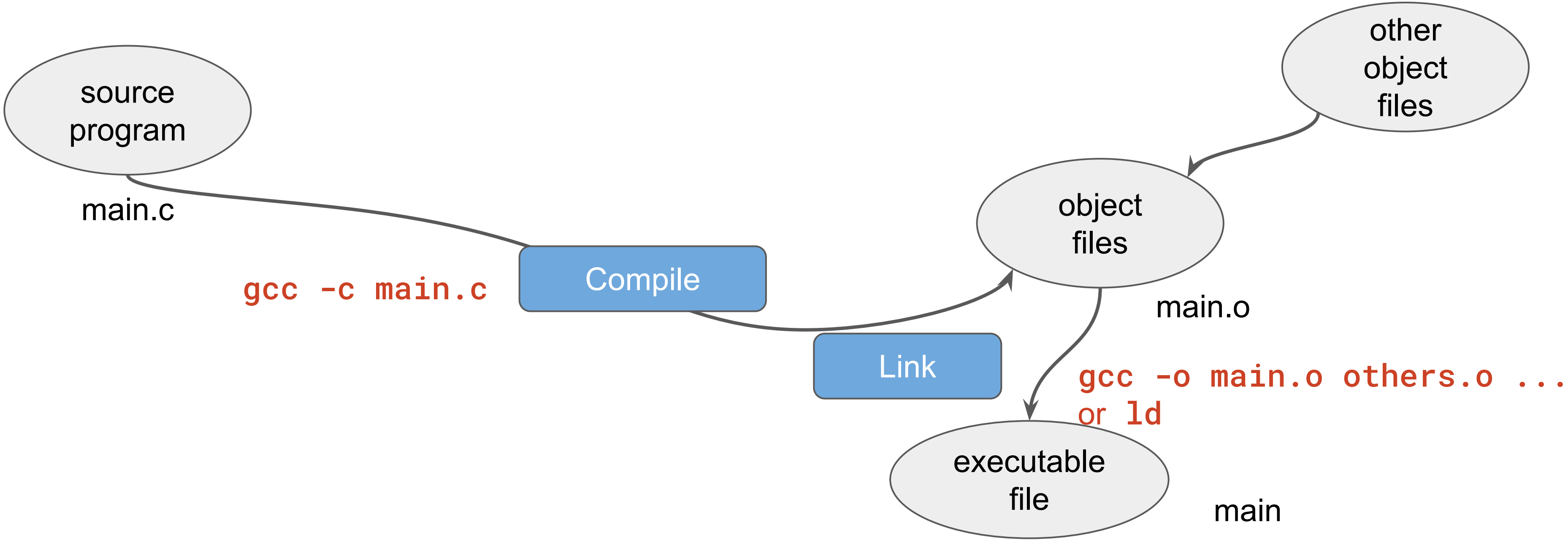
- Cloning a repository onto your local machine
    - `git clone <url>`
- Adding files to the staging area
    - `git add <filename>`
- Check status of files in the local repository
    - `git status`
- Commit files in your staging area to your local git repository
    - `git commit -m <short message>`
- Pushing/Pulling commits to and from GitHub
    - `git pull`
    - `git push`

UNIX Programming 101

The Journey of Compilation



The Simplified Journey of Compilation



source program

main.c

```
gcc -c main.c
```

Compile

object files

main.o

other object files

Link

```
gcc -o main.o others.o ...
```

or `ld`

executable file

main

Compilation options

- Use command line options to control the behaviour of `gcc`
- `-o <output filename>` output file name (create executable if `-c` is not specified)
- `-c [output filename]` create an object file (if no output filename then it will use the source code filename appended with `.o`)
- `-g` keep debugging information
- `-Wall` adds most warnings
    - Your assignment **must** compile cleanly (as in no warnings) for full marks

Compilation - compile and link

- Some files do not need to be recompiled
- Object files can be reused/shared
- Use `make` to help you automate this process - covered next

Make - Introduction

- **Make**: a tool to automate compilation
- **REQUIRED FOR ASSIGNMENTS**
- When properly setup it should only recompile outdated files
- You will need a `Makefile` in your project folder

make - Makefile basics

- Hello world in Makefile

```
say_hello:
    echo "Hello World"
```

- Run it in shell

```
$ make
echo "Hello World"
Hello World
```

make - Makefile basics

- Syntax to define rules

```
target: prerequisites
<TAB> recipe
```

- Run it in shell

```
$ make <target>
```

make - Makefile basics

- When we run `make <target>` in the shell, `make` will
    - Check the dependencies of the target
        - If any of the dependencies have been modified since the last time the `<target>` was generated it will
            - Run the recipe line by line
- You will need at least 3 targets in your Makefile for your assignment
    - compile
    - link
    - clean

make - Makefile basics

- Dependencies can be another rule’s target!
- Putting them all together

```
code_piece1.o: code_piece1.c
    gcc -c code_piece1.c -o code_piece1.o

code_piece2.o: code_piece2.c
    gcc -c code_piece2.c -o code_piece2.o

awesome_app: code_piece1.o code_piece2.o
    gcc -o awesome_app code_piece1.o code_piece2.o

clean:
    rm *.o awesome_app
```

make - Variables

```
CC      = gcc
CFLAGS  = -Wall
OBJECTS = code_piece1.o code_piece2.o

code_piece1.o: code_piece1.c
    $(CC) $(CFLAGS) -c code_piece1.c -o code_piece1.o

code_piece2.o: code_piece2.c
    $(CC) $(CFLAGS) -c code_piece2.c -o code_piece2.o

awesome_app: $(OBJECTS)
    $(CC) -o awesome_app $(OBJECTS)
```

Define and assign variables

Use variables

```
    gcc -Wall -c code_piece1.c -o code_piece1.o
```

Equivalent to

GDB

- A tool to debug your program
- Use it to find errors that's hard to address
- Need to add the flag `-g` when you compile your program

```
$(GG) $(CFLAGS) prog.c -o output -g
```

- Then, use gdb by:

```
gdb ./output
```

- Running `list` in gdb will display the code
- `break <line#>` will add a breakpoint at the specified line number
- `run` will execute the program from the start to finish or until the first breakpoint
- Important commands
  - **run / continue / next / step / until / print / call / quit / break + line #** / etc...

Valgrind

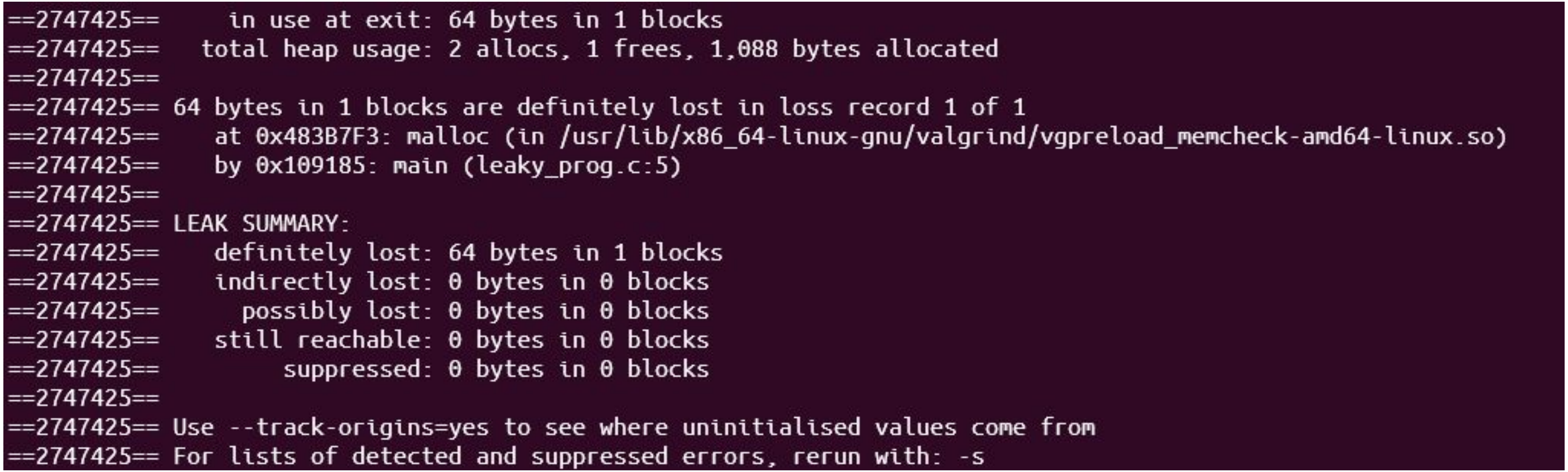
- A tool used to check memory leaks within your program
- Like gdb requires the `-g` flag during compilation to get the line number(s) of where the problem originates

```
valgrind --leak-check=yes ./your_prog arg1 arg2 …
```

Valgrind example - leaky_prog.c

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

int main() {
    int *p = malloc(64);
}
```

Valgrind example output



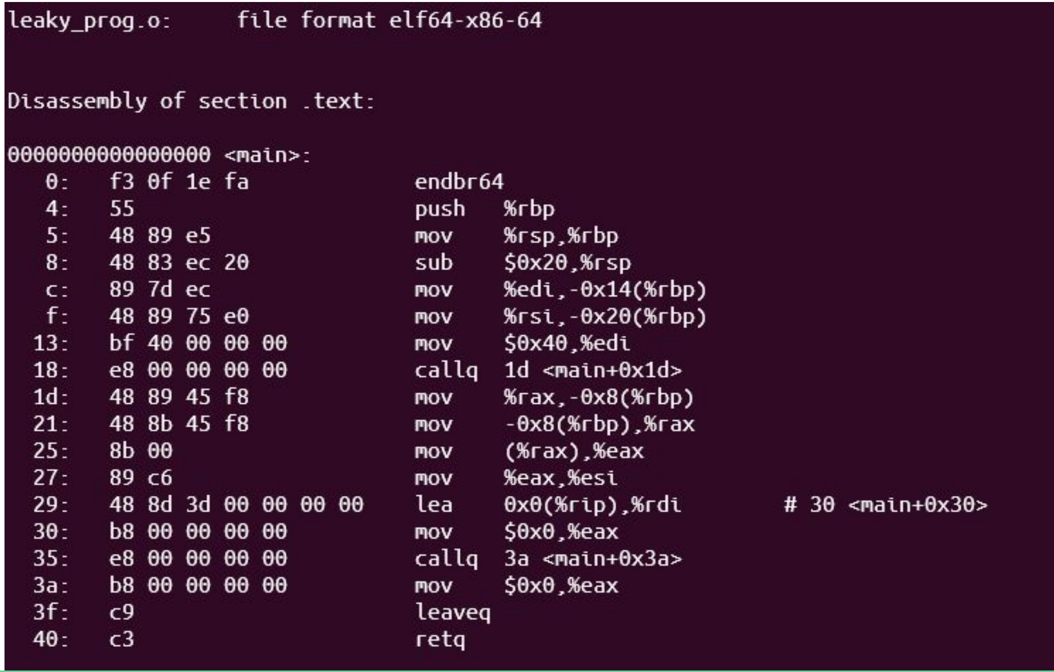
```
==2747425==     in use at exit: 64 bytes in 1 blocks
==2747425==   total heap usage: 2 allocs, 1 frees, 1,088 bytes allocated
==2747425==
==2747425== 64 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2747425==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==2747425==    by 0x109185: main (leaky_prog.c:5)
==2747425==
==2747425== LEAK SUMMARY:
==2747425==    definitely lost: 64 bytes in 1 blocks
==2747425==    indirectly lost: 0 bytes in 0 blocks
==2747425==      possibly lost: 0 bytes in 0 blocks
==2747425==    still reachable: 0 bytes in 0 blocks
==2747425==         suppressed: 0 bytes in 0 blocks
==2747425==
==2747425== Use --track-origins=yes to see where uninitialised values come from
==2747425== For lists of detected and suppressed errors, rerun with: -s
```

Man page

- AKA manual page are a set of software documents for user commands, syscalls, and libraries for Unix and Unix-like OSes
- Usage: `man [section] name`
- Sections
  - 1 - user commands (e.g., `man man` - “What is man and how to use it”)
  - 2 - system calls (e.g., `man 2 fork` - “How to create a child process”)
  - 3 - C standard library (e.g., `man 3 printf` - “How do I print in C”)
  - 7 - miscellaneous (e.g., `man 7 signal` - “What are all the Linux signals”)
- Note, many wrapper functions from the C standard library are named similarly as their system call counterparts. **YOU MUST USE** system calls for your assignment (section 2 ONLY)
  - Read the man page on how to call the right version

objdump

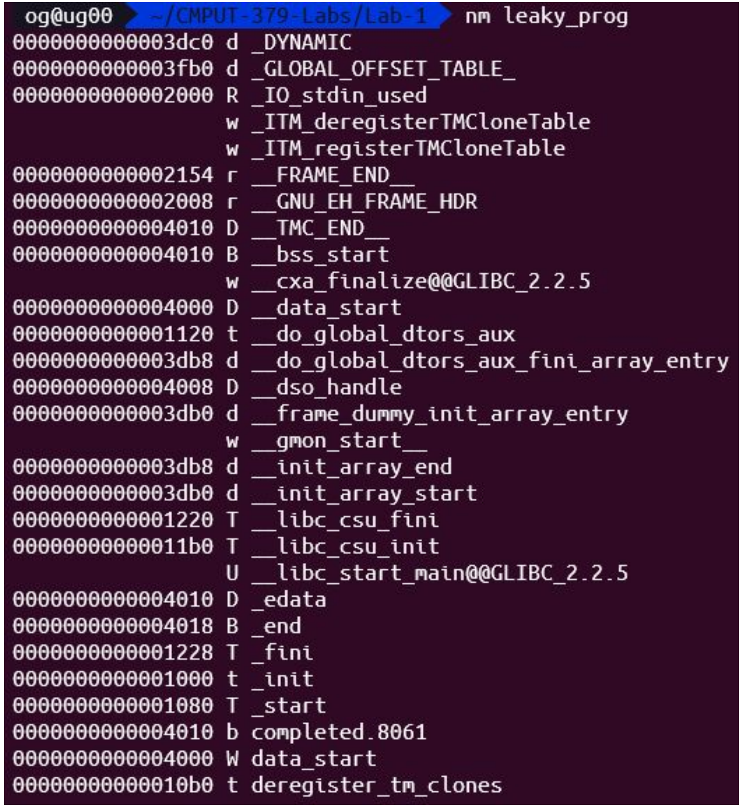
- Displays information about one or more object files
- usage:
  - `objdump -h <exe>`
  - `objdump -D <object_file>`



```
leaky_prog.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 20             sub    $0x20,%rsp
   c:   89 7d ec                mov    %edi,-0x14(%rbp)
   f:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
  13:   bf 40 00 00 00          mov    $0x40,%edi
  18:   e8 00 00 00 00          callq  1d <main+0x1d>
  1d:   48 89 45 f8             mov    %rax,-0x8(%rbp)
  21:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  25:   8b 00                   mov    (%rax),%eax
  27:   89 c6                   mov    %eax,%esi
  29:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 30 <main+0x30>
  30:   b8 00 00 00 00          mov    $0x0,%eax
  35:   e8 00 00 00 00          callq  3a <main+0x3a>
  3a:   b8 00 00 00 00          mov    $0x0,%eax
  3f:   c9                      leaveq
  40:   c3                      retq
```

nm

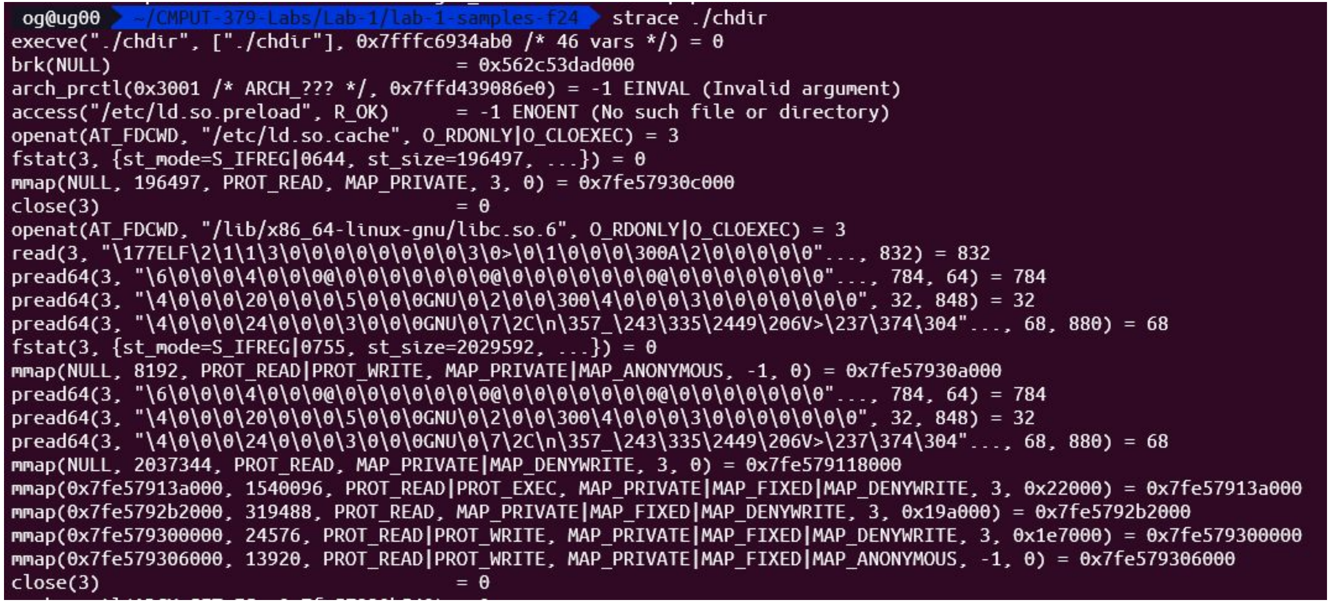
- List symbols from object files
- Usage:
  - `nm <exe>`



```
og@ug00  ~/CMPUT-379-Labs/Lab-1  nm leaky_prog
0000000000003dc0 d _DYNAMIC
0000000000003fb0 d _GLOBAL_OFFSET_TABLE_
0000000000002000 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000002154 r __FRAME_END__
0000000000002008 r __GNU_EH_FRAME_HDR
0000000000004010 D __TMC_END__
0000000000004010 B __bss_start
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000004000 D __data_start
0000000000001120 t __do_global_dtors_aux
0000000000003db8 d __do_global_dtors_aux_fini_array_entry
0000000000004008 D __dso_handle
0000000000003db0 d __frame_dummy_init_array_entry
                 w __gmon_start__
0000000000003db8 d __init_array_end
0000000000003db0 d __init_array_start
0000000000001220 T __libc_csu_fini
00000000000011b0 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000004010 D _edata
0000000000004018 B _end
0000000000001228 T _fini
0000000000001000 t _init
0000000000001080 T _start
0000000000004010 b completed.8061
0000000000004000 W data_start
00000000000010b0 t deregister_tm_clones
```

strace

- Trace system calls and signals
- Usage:
  - `strace [options] command [args]`



```
 og@ug00  ~/CMPUT-379-Labs/Lab-1/lab-1-samples-f24  strace ./chdir
execve("./chdir", ["./chdir"], 0x7fffc6934ab0 /* 46 vars */) = 0
brk(NULL)                               = 0x562c53dad000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd439086e0) = -1 EINVAL (Invalid argument)
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=196497, ...}) = 0
mmap(NULL, 196497, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fe57930c000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300A\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\7\2C\n\357_\243\335\2449\206V>\237\374\304"..., 68, 880) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2029592, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fe57930a000
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\7\2C\n\357_\243\335\2449\206V>\237\374\304"..., 68, 880) = 68
mmap(NULL, 2037344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fe579118000
mmap(0x7fe57913a000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x7fe57913a000
mmap(0x7fe5792b2000, 319488, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19a000) = 0x7fe5792b2000
mmap(0x7fe579300000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7fe579300000
mmap(0x7fe579306000, 13920, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fe579306000
close(3)                                = 0
```

More Linux Tools

- **top** - a tool to show running processes
- **objdump** - display information from object files
- **pstree** - show running processes as a tree
- **ps** - get the list of running processes (-el, -aux)
  - `ps -U CCID -u CCID l` - get all processes from user `CCID`
  - `ps aux` get all processes on computer!
  - `ps aux | grep <pattern>` returns relevant processes
- **tmux** - terminal multiplexer
  - `Ctrl-b d` to detach, `Ctrl-b %` to split, `Ctrl-b o` next pane
- **[p]kill** - kill a process
  - `pkill -u CCID -U CCID` - kill all processes from user `CCID`
- **watch** - execute a program periodically
  - `watch -n <interval> command`

Using system calls - error handling

- For most system calls, a return value < 0 indicates an error
- See **man 3 errno** to see all possible errors
- Check the variable **errno** to see what the error is
  - Include the header `#include<errno.h>`
- Use **perror()** to print error detail
  - Include the header `#include<stdio.h>`

Using system calls

- For this assignment you have to use system calls to implement your shell, here are some listed
    - `chdir()`
    - `getpid()`
    - `fork()`
    - `execve()`
    - `_exit()`
    - `wait(), waitpid()`

- Discuss during the next lab
    - `open()`
    - `close()`
    - `dup2()`
    - `pipe()`
    - `kill()`
    - `sigaction()`

```
$ man 2 fork

$ man 2 execve

$ man 2 _exit
```

System call `chdir()`

- Change the current working directory
- Example code - chdir.c

```
#include <unistd.h>

int chdir(const char *path);
```

- Parameter:
  - path - which the user want to make the current working directory
- Return Value:
  - 0 - success
  - -1 - an error occurs and **errno** is set appropriately

System call `fork()`

- Create a new process
- Example code - fork.c

```
#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);
```

- Return Value:
  - -1 - creation of a child process was unsuccessful
  - 0 - Returned to the newly created child process
  - >0 - Returned to parent or caller. The value contains process ID of newly created child process

System call `getpid()`, `getppid()`

- Get the process ID
- Example code - getpid.c

```
#include <sys/types.h>

#include <unistd.h>

pid_t getpid(void);
```

- `Return Value:`
    - The process ID of the calling process

```
pid_t getppid(void);
```

- `Return Value:`
    - The process ID of the parent process

System call `execve()`

- Execute a program and replace the current process image
- Example code - execve.c

```
#include <unistd.h>

int execve(const char *path, char *const argv[], char *const envp[]);
```

- Parameters:
  - path - the path of the file being executed
  - argv - null terminated array of the arguments for the program being executed
  - envp - array of strings, conventionally of the form **key=value**
- Return Value:
  - No return - success
  - -1 - an error occurs and **errno** is set appropriately

System call `_exit()`

- Terminate process and return status to the parent
- Not the same as `exit()` **DO NOT USE** `exit()` on the assignment as it is a library function
- Example codes - exit1.c and exit2.c

```
#include <unistd.h>
```

```
int _exit(int status);
```

- `Parameter:`
    - `Status - value returned to the parent process`

System call `wait()`

- Wait until one of its children terminates
- Example code - wait.c and waitpid.c

```
#include <sys/types.h>

#include <sys/wait.h>

pid_t wait(int *wstatus);
```

- Parameter:
  - Status - value returned to the parent process

- Return Value:
  - Terminated process ID - success
  - 0 or -1   - error

System call waitpid()

- Wait for a specific process ID to change state
- Example code - waitpid.c

```
#include <sys/types.h>

#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- Parameter:
  - pid - The process id to wait on
  - wstatus - Value returned to the parent process (often NULL)
  - options - ORed constants to change the behaviour of waitpid (often 0)

- Return Value:
  - Terminated process ID - success
  - 0 or -1

System call `times()`

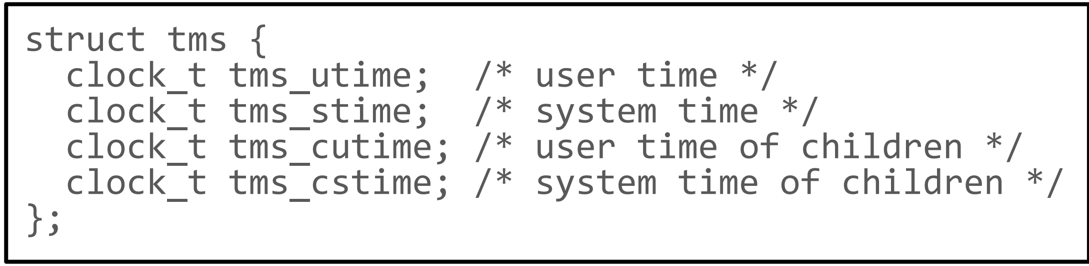
- Get the process times
- Example code - times.c

```
#include <sys/times.h>
```

```
clock_t times(struct tms *buf);
```

- Return Value:
  - The number of clock ticks that have elapsed since the past (> 0)
  - -1 - error has occurred and errno is set

`$ man 2 times`



```
struct tms {
  clock_t tms_utime;  /* user time */
  clock_t tms_stime;  /* system time */
  clock_t tms_cutime; /* user time of children */
  clock_t tms_cstime; /* system time of children */
};
```

System call open( )

- Open a file

```
#include <sys/stat.h>

#include <fcntl.h>

int open(const char *path, int oflags, mode_t mode);
```

- Parameter:
  - oflags - O_RDONLY, O_WRONLY,O_RDWR, O_APPEND, O_CREAT, etc.
  - mode   - S_IRUSR, S_IWUSR, S_IXUSR, etc.
- Return Value:
  - -1   - error
  - Others - file descriptor

System call `close()`

- Close a file

```
#include <unistd.h>

int close(int fildes);
```

- Parameter:
  - fildes - The file descriptor to be closed
- Return Value:
  - -1 - error
  - 0 - success

System call write()

- Write to a file descriptor
- Example code - write.c

```
#include <unistd.h>

ssize_t write(int fd, const void* buf, size_t count);
```

- Parameter:
  - fd - The file descriptor to be written to
  - buf - The buffer to write to the fd
  - count - The number of bytes to write to the fd
- Return Value:
  - -1 - error
  - 0 - success

System call `read()`

- Read from a file descriptor
- Example code - read.c

```
#include <unistd.h>

ssize_t read(int fd, const void* buf, size_t count);
```

- Parameter:
  - fd - The file descriptor to be read from
  - buf - The buffer to read the contents in fd into
  - count - The number of bytes to read from the fd
- Return Value:
  - -1 - error
  - 0 - success

System call dup2()

- Duplicates one file descriptor, making them aliases, and then deleting the old file descriptor
- Example code - dup2.c

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
```

```
●  Parameter:
   ○  fildes  - source file descriptor
   ○  fildes2 - target file descriptor
●  Return Value:
   ○  <0     - error
   ○  Others - second file descriptor
```

System call `pipe()`

- Creates a unidirectional data channel for interprocess communication (IPC)
- Example code - pipe.c

```
int pipe(int pipefd[2]);
```

- `Parameter:`
  - `pipefd - two file descriptors, read/write ends of the pipe`
- `Return Value:`
  - `-1 - error`
  - `0  - success`

System call `kill()`

- Send signal to a process
- Example code - kill.c

```
#include <sys/types.h>

#include <signal.h>

int kill(pid_t pid, int sig);
```

- Parameter:
  - pid - target process
  - sig - signal want to send
- Return Value:
  - 0 - success
  - -1 - error

Question from class

What if the parent of a process is killed?

systemd(1) - A - B - C

A gets killed... B is orphan to systemd(1)

systemd(1) - B - C

B gets killed... C is orphan to systemd(1), B is a zombie

systemd(1) - C

systemd(1) - A - B(Z)

```
pstree -s -p <pid>
```

```
ps -f <pid>
```