CSE 29
Lecture 9 Summary

February 3, 2026

📣 Logistical Things

- You took an exam in week 4!
    - You will get a score 0-4
    - You will get a makeup opportunity in finals week
        - Before makeups, you will know if you need to do makeups (grades will be released)
    - DO NOT WORRY TOO MUCH if you got a low score
        - “High standards, multiple tries”
        - We have makeups so you have the room to fail and try again

Review Questions

Answers in speaker notes!

1. What does `strtok` do when given NULL as the first argument? (write a short sentence)
2. Given `./pwcrack 1234abcd` what are the command line arguments?
3. Given `./pwcrack 1234abcd < input.txt > result.txt` what are the command line arguments?

Memory

Let’s look at the handout

Some stuff to observe

- Line 19 Typo - replace “main” with “static_counter_fun” twice to match output
- We printed out the address of global variables
- We printed out the address of functions
    - The addresses are a lot smaller than when we started printing out addresses of variables
    - ex. `&main` is 0x…cf1c2 and `&NUM` is 0x…dd2010

```
1  #include <stdio.h>
2
3  int NUM = 1000;
4  char HI[] = "Hi!";
5
6  int static_counter_fun() {
7    static int ctr = 0; // Only happens on the first call!
8    ctr += 1;
9    printf("ctr:\t%d\n", ctr);
10   printf("&ctr:\t%p\n", &ctr);
11   return ctr;
12 }
13
14 int main(int argc, char** argv) {
15   char hello[] = "hello everyone";
16   char* ptr = hello;// could also write &hello
17   char* hiptr = HI; // could also write &HI
18   int* numptr = &NUM; // could NOT also write NUM
19   printf("&main:\t%p\n", &main);
20   printf("&main:\t%p\n", &main);
21   printf("&NUM:\t%p\n", &NUM);
22   printf("numptr:\t%p\n", numptr);
23   printf("&HI:\t%p\n", &HI);
24   printf("hiptr:\t%p\n", hiptr);
25   printf("stdin:\t%p\n", &stdin);
26   static_counter_fun();
27   static_counter_fun();
28   printf("&argv:\t%p\n", &argv);
29   printf("ptr:\t%p\n", ptr);
30   printf("&ptr:\t%p\n", &ptr);
31   printf("&hiptr:\t%p\n", &hiptr);
32   printf("hello:\t%p\n", hello);
33   printf("argv:\t%p\n", argv);
34   printf("argv[0]:%p\n", argv[0]);
35 }
```

&static_counter_fun

```
$ ./layout
&static_counter_fun:    0x6046a1dcf169
&main:   0x6046a1dcf1c2
&NUM:    0x6046a1dd2010
numptr:  0x6046a1dd2010
&HI:     0x6046a1dd2014
hiptr:   0x6046a1dd2014
stdin:   0x6046a1dd2020
ctr:     1
&ctr:    0x6046a1dd202c
ctr:     2
&ctr:    0x6046a1dd202c
&argv:   0x7ffe89c5a5b0
ptr:     0x7ffe89c5a5d9
&ptr:    0x7ffe89c5a5c0
&hiptr:  0x7ffe89c5a5c8
hello:   0x7ffe89c5a5d9
argv:    0x7ffe89c5a718
argv[0]:        0x7ffe89c5c73c
```

🤔 How many bytes are between the lowest and the highest address? (0x6046a1dcf1c2 and 0x7ffe89c5c73c)

a. 1 thousand
b. 1 million
c. Billions
d. More

```
$ ./layout
&static_counter_fun:    0x6046a1dcf169
&main:   0x6046a1dcf1c2
&NUM:    0x6046a1dd2010
numptr:  0x6046a1dd2010
&HI:     0x6046a1dd2014
hiptr:   0x6046a1dd2014
stdin:   0x6046a1dd2020
ctr:     1
&ctr:    0x6046a1dd202c
ctr:     2
&ctr:    0x6046a1dd202c
&argv:   0x7ffe89c5a5b0
ptr:     0x7ffe89c5a5d9
&ptr:    0x7ffe89c5a5c0
&hiptr:  0x7ffe89c5a5c8
hello:   0x7ffe89c5a5d9
argv:    0x7ffe89c5a718
argv[0]:        0x7ffe89c5c73c
```

Answer in speaker notes and next slide

A short program for the answer

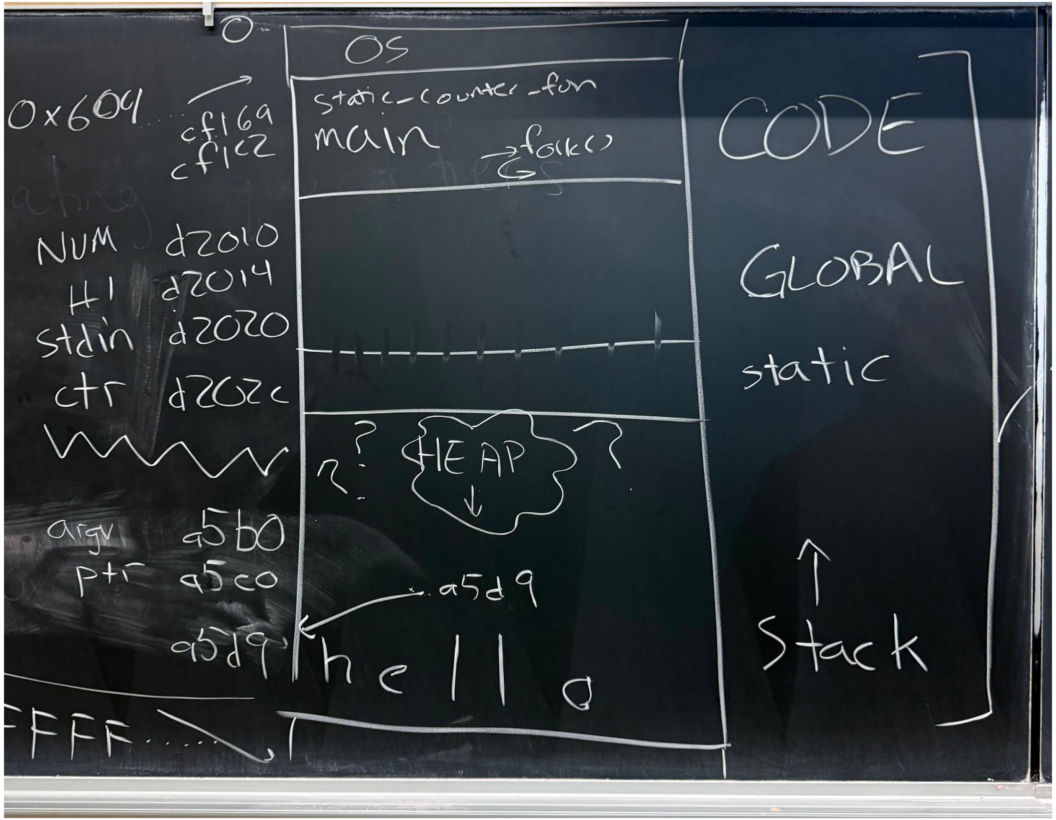
```
from worksheet -> &static_counter_fun: 0x6046A1DCF169, &argv: 0x7FFE89C5A5B0
static_counter_fun            hex = 0x6046A1DCF169, dec = 105856479588713
argv                          hex = 0x7FFE89C5A5B0, dec = 140731209852336
argv-static_counter_fun in    hex = 0x1FB7E7E8B447, dec = 34874730263623
```

That difference is 34,874,730,263,623 which is 34 trillion and change (source code in speaker notes if you're interested)

Does it really take that many bytes to run a program?

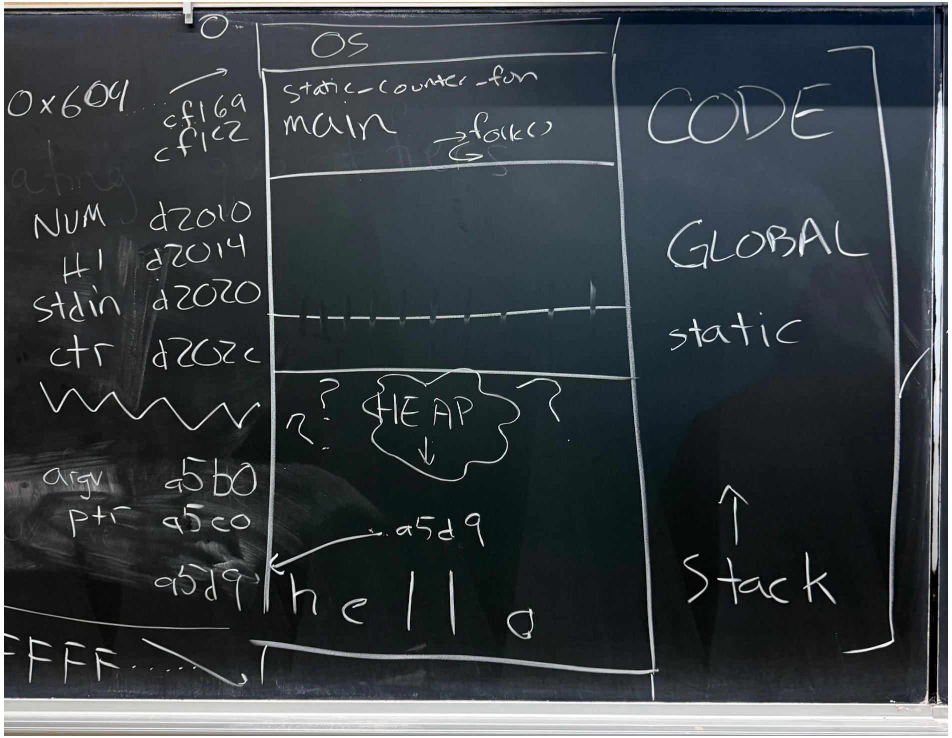
- 34 trillion is a huge number of bytes, do we really need this much to run a simple program?
- Does Joe’s laptop or ieng6 really give each process from each user that much memory?
- This is a good thing to wonder about! We will talk about this more in future lectures!

Let's fill out a memory diagram for this!



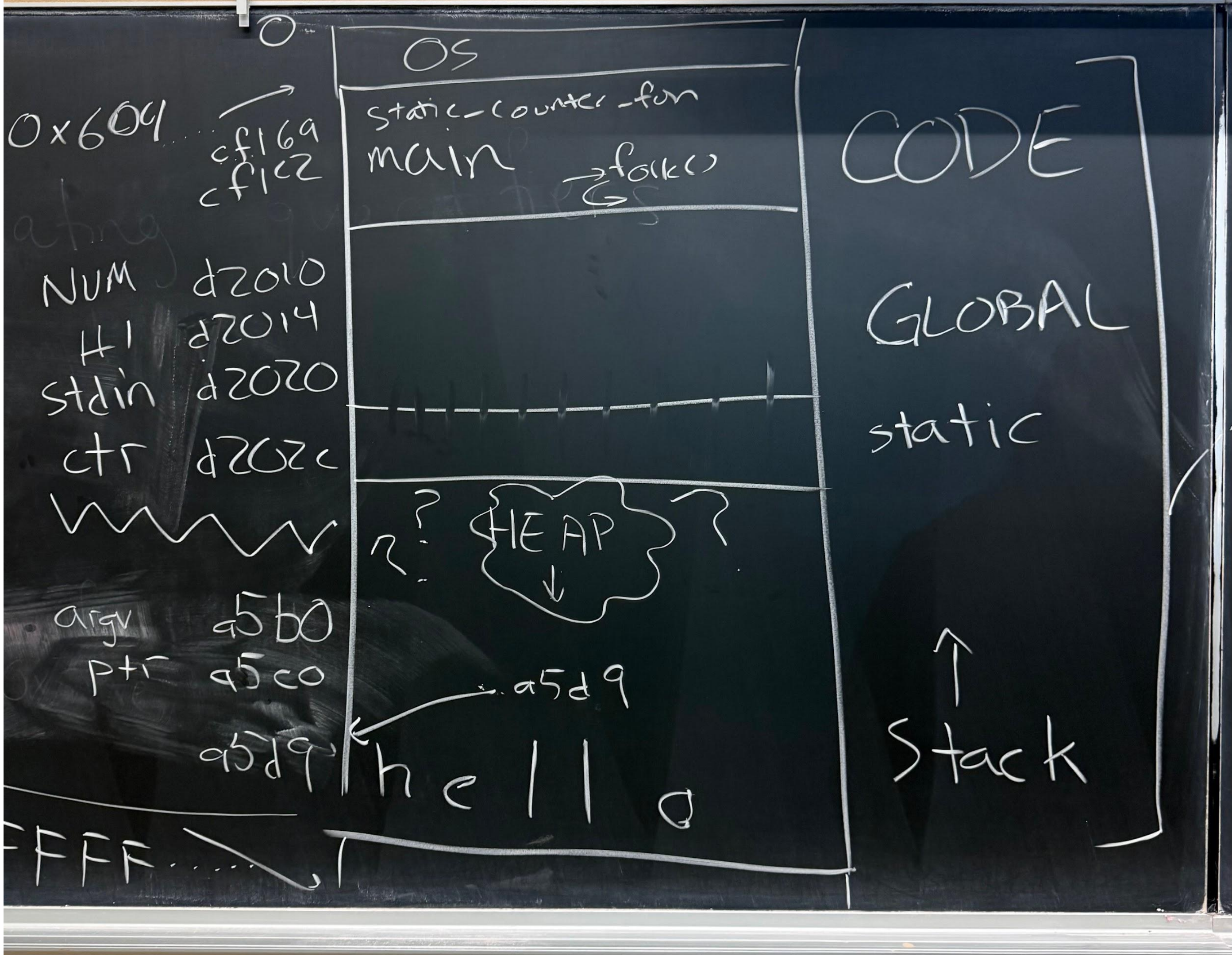
Where functions are in memory

- Function definitions are stored at the top/lower addresses of memory
- We can see in this picture that `static_counter_fun` and `main` are both in the Code part of our memory diagram



Where variables are in memory

- OS
- Global variables
  - NUM, HI, stdin
- Static variables
  - ctr
- Heap 🤔
- Stack
  - Argv, ptr …



Questions

- When you import a file, where is that stored?
  - Whatever is in an import/header file (ex. stdio.h), it gets concatenated to your code and gets compiled and ran normally
  - It would appear in the “Code” section of your memory diagram

fork(), execvp(...) and wait(...)

Back to the shell

- How can we run commands for the shell we created?
- Here’s what we currently have →

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




int parse_args(char* input, char** args) {
  char* current = strtok(input, " ");
  int current_index = 0;
  while(current != NULL) {
    args[current_index] = current;
    current = strtok(NULL, " ");
    current_index += 1;
  }
  return current_index;
}

int main() {
  while(1) {
    printf("  ");
    char input[2048];
    fgets(input, 2048, stdin);
    printf("Now computer run this: %s\n", input);

    char* args[1000];
    int argc = parse_args(input, args);
    for(int i = 0; i < argc; i += 1) {
      printf("args[%d]: %s\n", i, args[i]);
    }
```

How can we start running a new program in our shell?

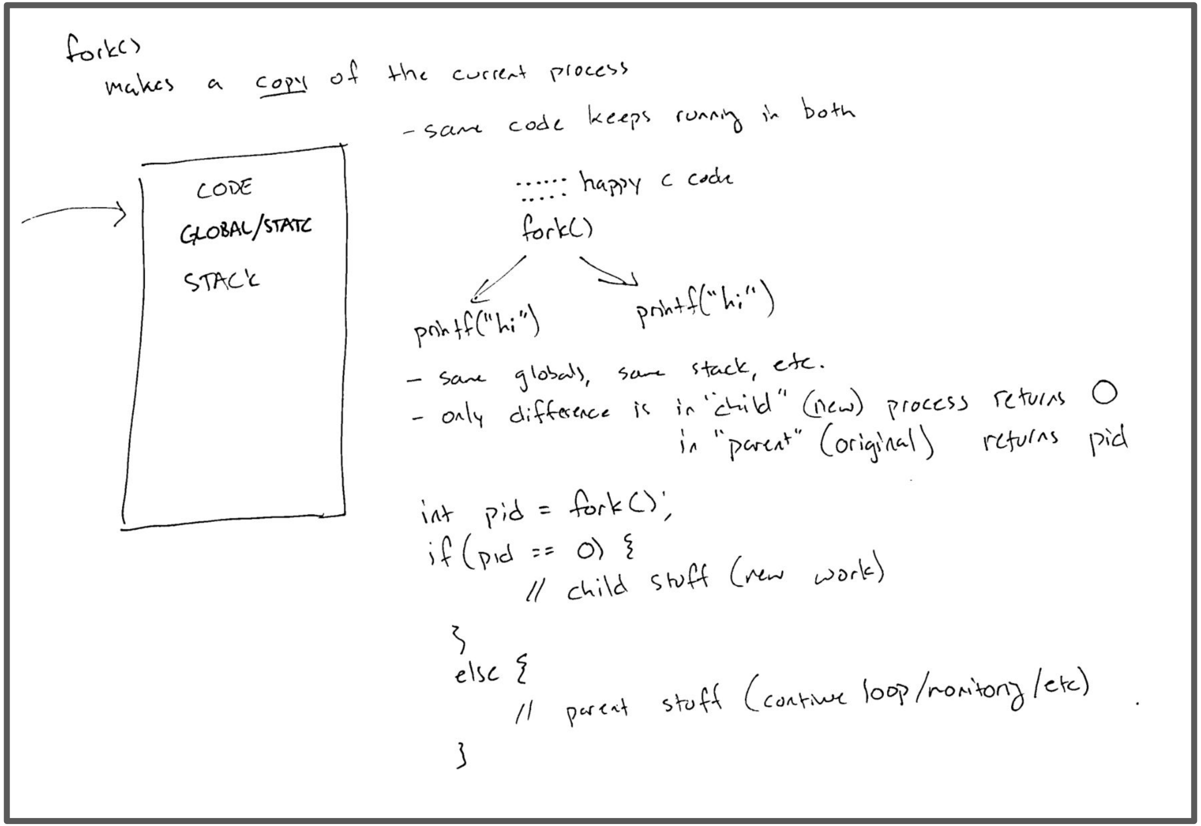
- Answer: `fork()`, `exec()`, and `wait()`
- These are all **system calls**
  - System calls - a special kind of function that interacts with the operating system
- Across different OSs, there is something similar but not exactly the same
  - For example, in Windows, `fork()` and `exec()` are replaced with `CreateProcess()`. `wait()` is replaced with `WaitForSingleObject()`
  - Operating systems usually go about creating a new process the same way

fork()

- fork() - makes a copy of the current process
  - Takes a process' address space and make a copy of that, making a new process with the same code, global & static area, stack, everything
  - It makes a **child process** from the **parent process**
  - The child process starts running where fork() was called in the parent process
  - The only difference is that 0 is returned from fork() in the child and a non-zero integer is returned from fork in the parent
    - What is returned is the PID (Process ID)
    - Each process (Chrome, Spotify, ./my_program, on your computer has an ID when it's running)
- Common pattern when using fork()→

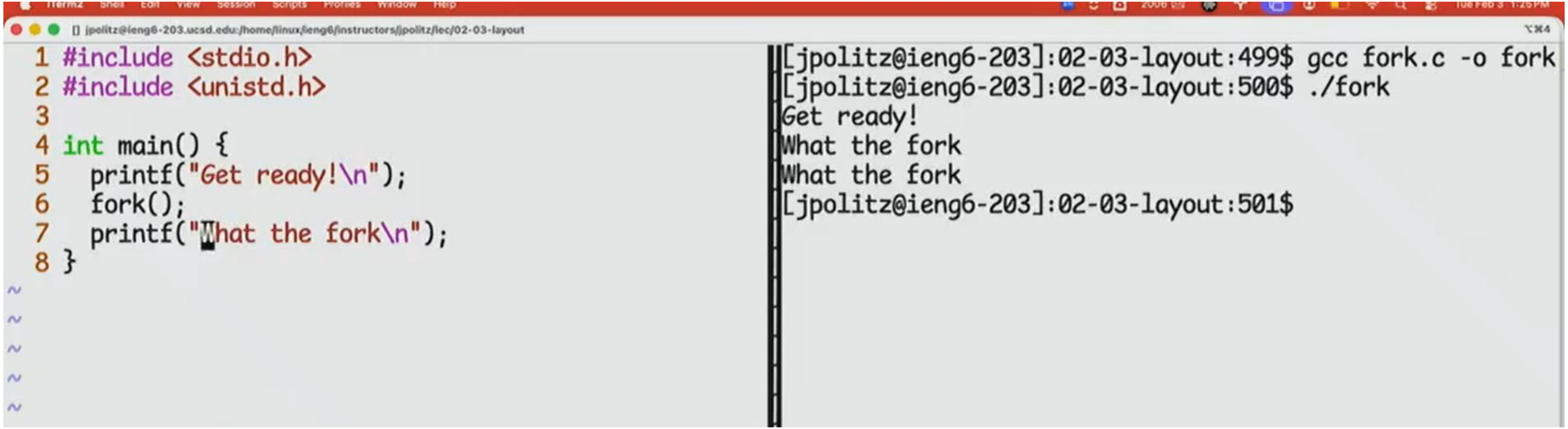
```
int pid = fork()
if pid == 0 { child process stuff
}
else { parent process stuff }
```

fork() Notes



fork()
makes a copy of the current process
- same code keeps running in both
CODE
GLOBAL/STATIC
STACK
...... happy c code
fork()
printf("hi")
printf("hi")
- same globals, same stack, etc.
- only difference is in "child" (new) process returns 0
in "parent" (original) returns pid
int pid = fork();
if (pid == 0) {
// child stuff (new work)
}
else {
// parent stuff (continue loop/monitoring/etc)
}

`fork()` demo



Parent process → “Get Ready” & “What the fork”

Child process → “What the fork”

`execvp(char* cmd, char** args)`

- Part of the `exec()` function family
- Takes in the name of the program (`cmd`) and also a list of arguments (`args`)
    - `cmd` could be “ls”, “pwd”, “./myprogram”, and anything you can run in the command line
- Replaces the current process by running `cmd` with the `args` as a new program
    - Erases everything in the process it was called in and starts running `cmd` from the beginning
- The cmd is argv[0] in our shell
- execvp needs a null terminator at the end of args

`wait()`

- `wait()` is called in the parent process to wait until the child process is finished
- We can call wait with NULL or with a child PID
  - `wait(NULL)` - waits for SOME child process to finish
  - `wait(pid)` - waits to the child process that has the PID to finish
- Not covered at length yet – more to come later
- Example of using wait

```
int pid = fork()
if pid == 0 { printf(“I’m a child!”) }
else { wait(NULL) }
```

Questions

- If we are `wait()`-ing, why are we `fork()`-ing anyways?
  - In different programs, the parent process needs to wait until the child process finishes their task before continuing
  - It will not always be the case (and usually not the case) that the parent process and the child process are doing the same exact thing

Lets try implementing this in our shell

```
int main() {
  while(1) {
    printf("⇒ ");
    char input[2048];
    fgets(input, 2048, stdin);
    if(input[strlen(input) - 1] == '\n') {
      input[strlen(input) - 1] = 0;
    }
    printf("Now computer run this: %s\n", input);

    char* args[1000];
    int argc = parse_args(input, args);
    args[argc] = NULL; // execvp expects a NULL
                       // at the end of the arg list
    for(int i = 0; i < argc; i += 1) {
      printf("args[%d]: \"%s\"\n", i, args[i]);
    }

    int pid = fork();
    if(pid == 0) {
      execvp(args[0], args);
    }
    else {
      // wait for child, then continue the loop
    }
  }
```

```
[jpolitz@ieng6-203]:02-03-layout:511$ ./shell
⇒ ls ..
Now computer run this: ls ..
args[0]: "ls"
args[1]: ".."
⇒ 01-13-utf8         01-20-utf8-analyzer   exam.txt
01-15-utf8-analyzer  02-03-layout

Now computer run this:
⇒ ls ..
Now computer run this: ls ..
args[0]: "ls"
args[1]: ".."
⇒ 01-13-utf8         01-20-utf8-analyzer   exam.txt
01-15-utf8-analyzer  02-03-layout
```

"ls .." was successfully ran!

Adding in fork() and execvp()

Wait, why is our command prompt (⇒) printing out in the middle of the `ls` output?

Fixing the issue

- We didn’t wait for the child process (ls) to be done so the parent process (the shell) just continued
- We should use wait(NULL) in the parent process

```c
    printf("args[%d]: \"%s\"\n", i, args[i]);
  }

  int pid = fork();
  if(pid == 0) {
    execvp(args[0], args);
  }
  else {
    wait(NULL); // waits for child process
  }
}
```

```
[jpolitz@ieng6-203]:02-03-layout:512$ gcc shell.c -o shell
[jpolitz@ieng6-203]:02-03-layout:513$ ./shell
⇒ ls ..
Now computer run this: ls ..
args[0]: "ls"
args[1]: ".."
01-13-utf8              01-20-utf8-analyzer   exam.txt
01-15-utf8-analyzer     02-03-layout
⇒ █
```

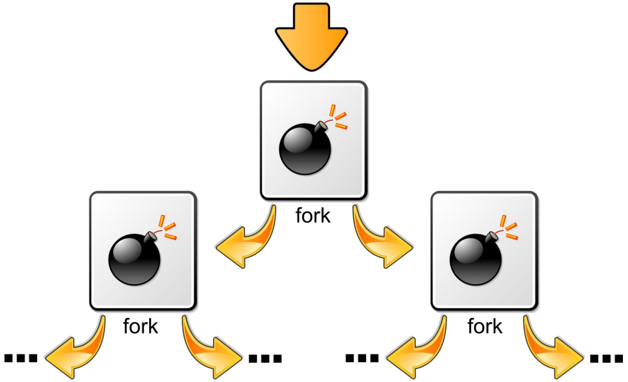
We added this line!

Questions

- If you create multiple forks, what would `wait(NULL)` wait for?
  - `wait(NULL)` waits for some child process to finish
  - If we use `wait(pid)`, we can wait for specific child processes
- If you `fork()` in the child process what happens?
  - The child process becomes a parent process and creates its own child process (a grandchild process)
- If we have a parent and a child process, if we change a global variable in the parent process, will this also affect the child process?
  - If we have a global variable in the parent, and we change it, it will not change in the child
  - They are two different processes

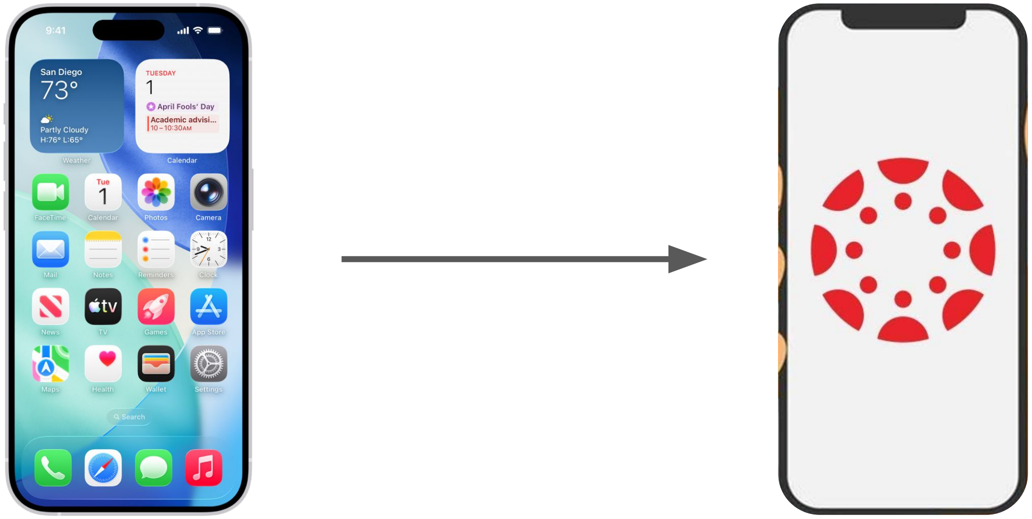
Side Note: Fork bomb

- If you misuse fork you can inadvertently make too many processes very quickly
- Imagine a loop that does not properly wait or exec and if fork is called many times it can crash the machine it is running on or make it otherwise very slow
- Fork bombs exponentially (x2) increases the amount of processes



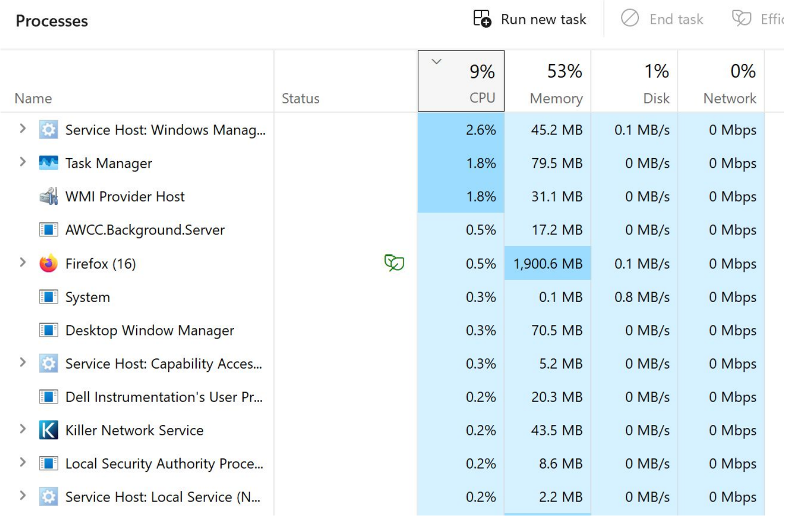
Your Phone has an OS, and does things similarly!

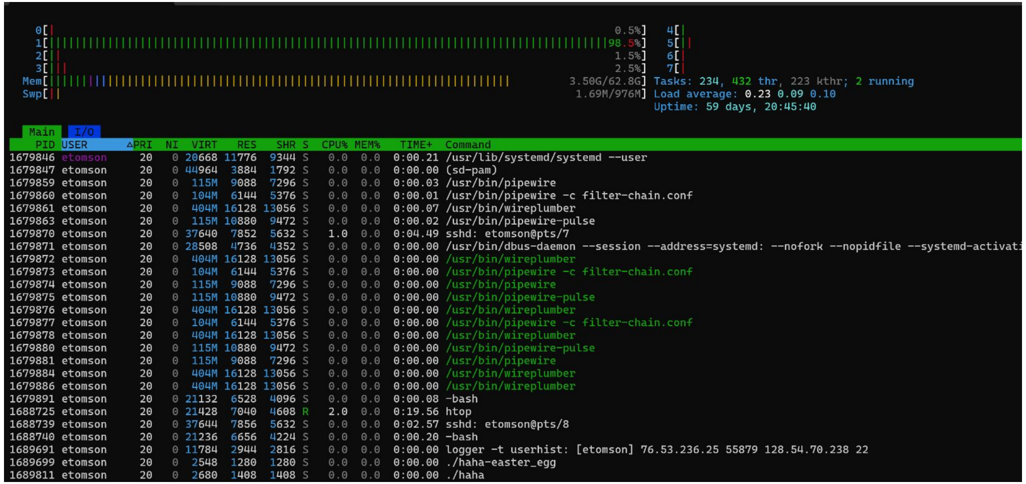
- Your home screen is like a shell where you can start processes
- When you click on an app, the process running your shell is fork'd and then exec'd to start running a new process (the app you want to open



Check out your Activity Monitor/Task Manager!

You can see all of the currently running processes on your computer!

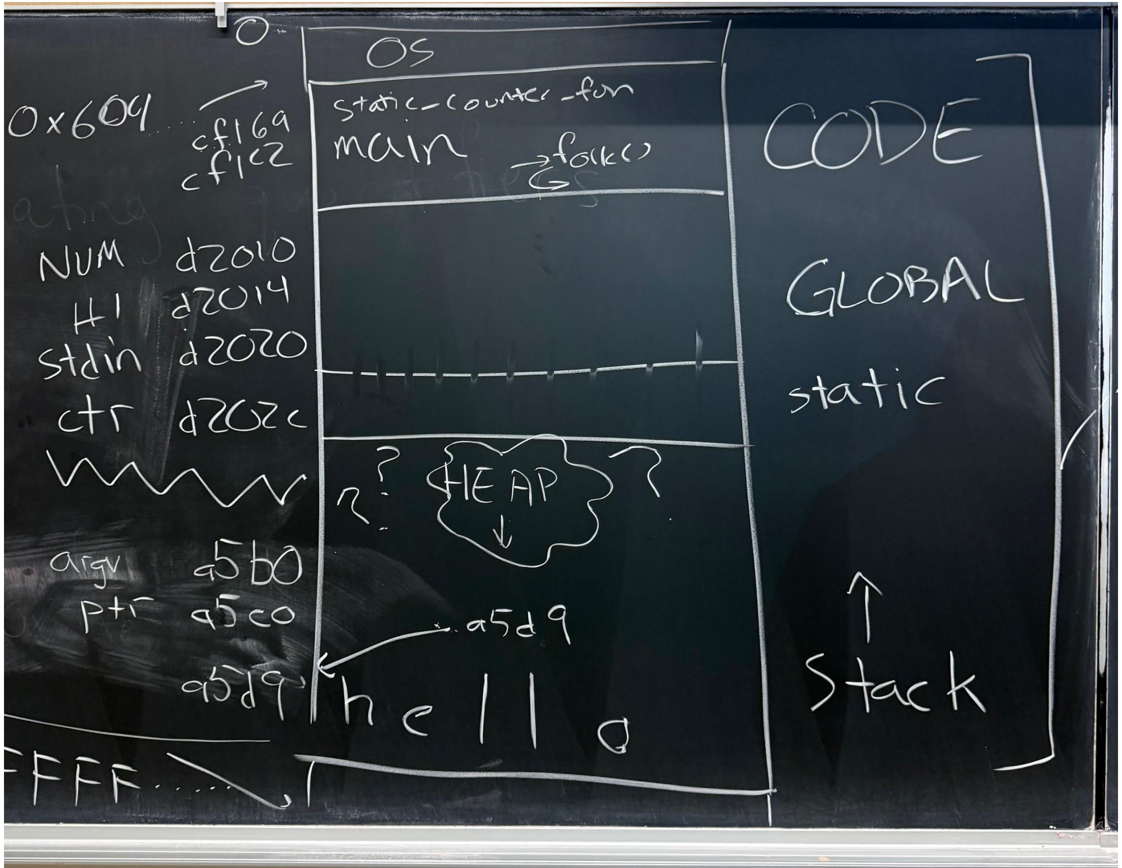




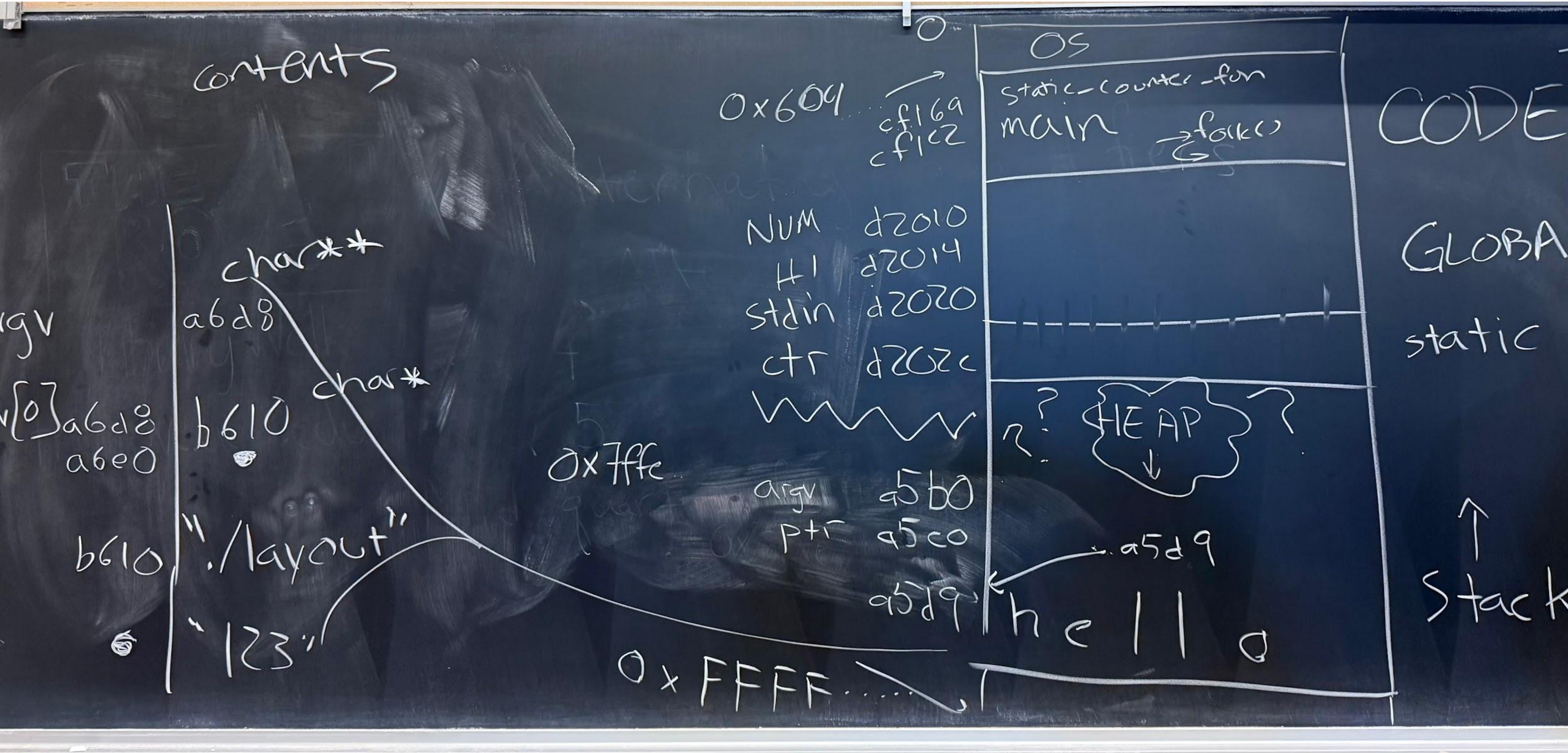
ieng6 ^ `htop` -> f6 ->USER -> scroll to myself (etomson)

Windows ^ ctrl+alt+delete -> Task Manager

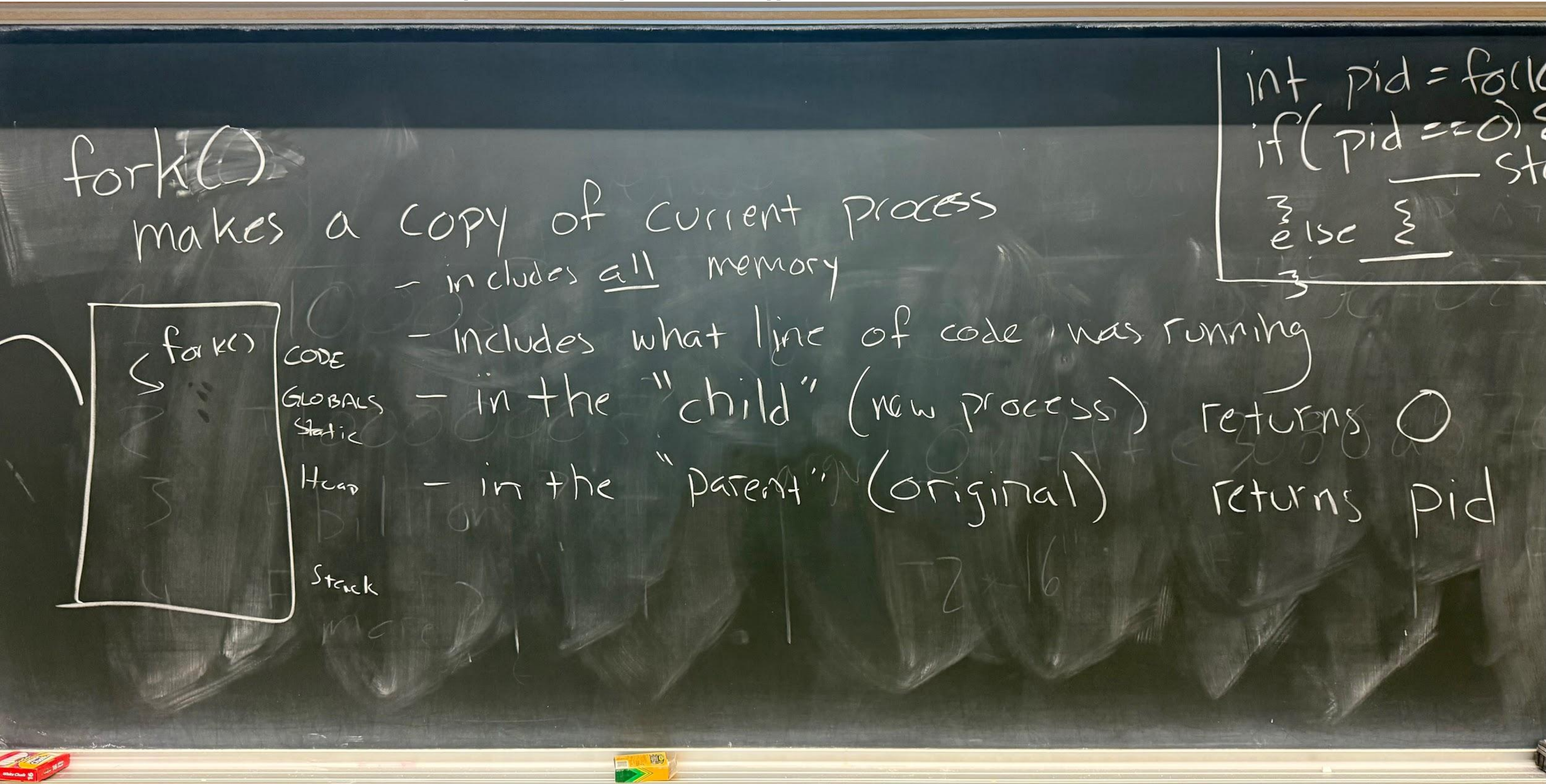
1 Joe's Board (11am) memory



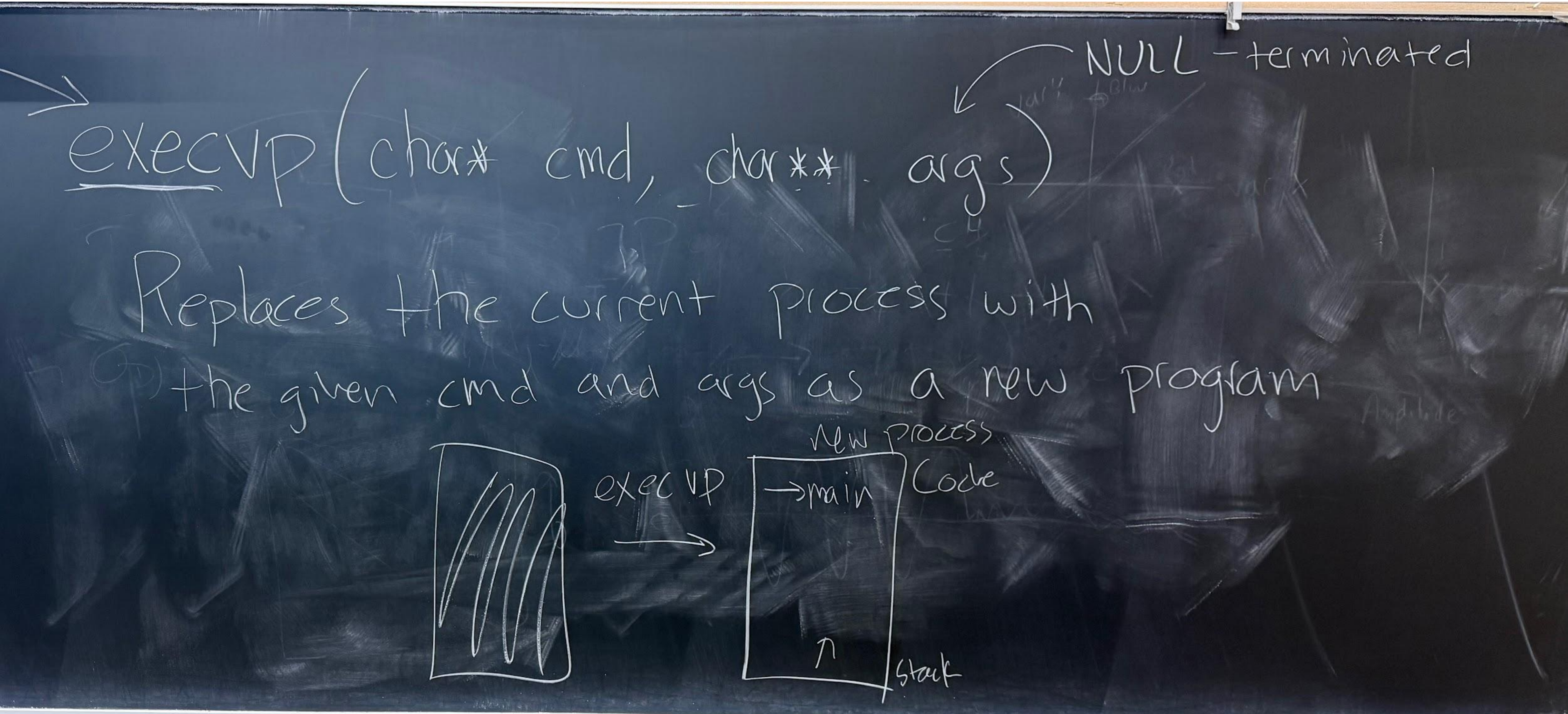
2 Joe's Board (11am) more on memory



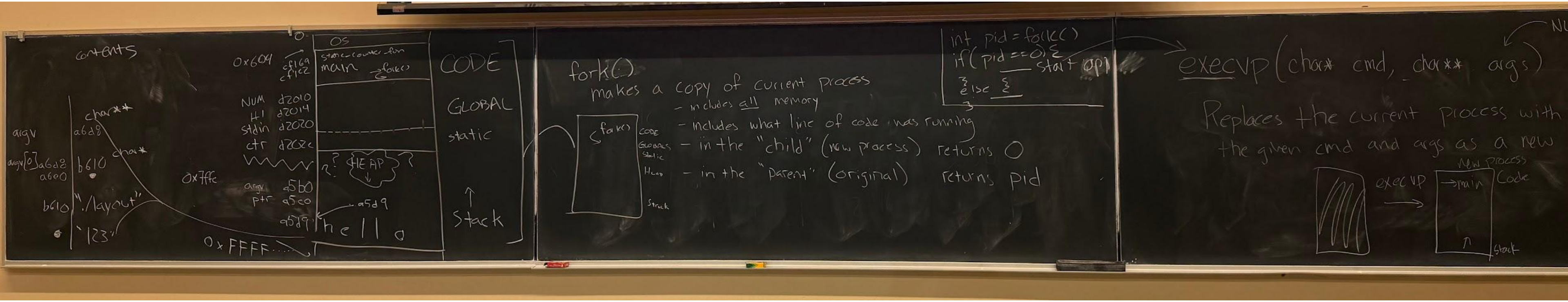
3 Joe’s Board (11am) fork()



4 Joe's Board (11am) execvp()

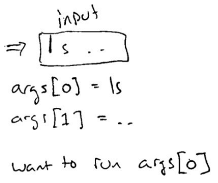


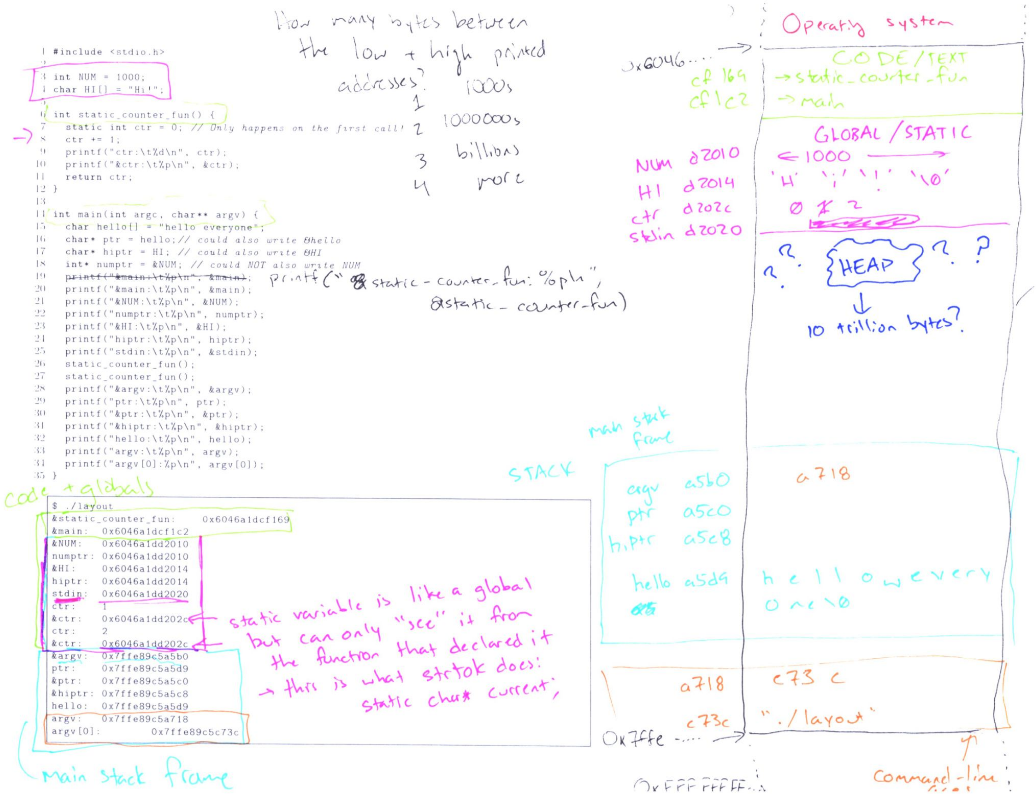
5 Joe's Board (11am) whole board



Joe’s Notes (12:30pm)

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




int parse_args(char* input, char** args) {
  char* current = strtok(input, " ");
  int current_index = 0;
  while(current != NULL) {
    args[current_index] = current;
    current = strtok(NULL, " ");
    current_index += 1;
  }
  return current_index;
}

int main() {
  while(1) {
    printf("  ");
    char input[2048];
    fgets(input, 2048, stdin);
    printf("Now computer run this: %s\n", input);

    char* args[1000];
    int argc = parse_args(input, args);
    for(int i = 0; i < argc; i += 1) {
      printf("args[%d]: %s\n", i, args[i]);
    }
















  }
}
```



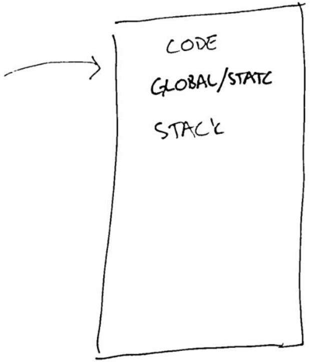


Joe's Notes (12:30pm)

fork()

makes a <u>copy</u> of the current process

- same code keeps running in both



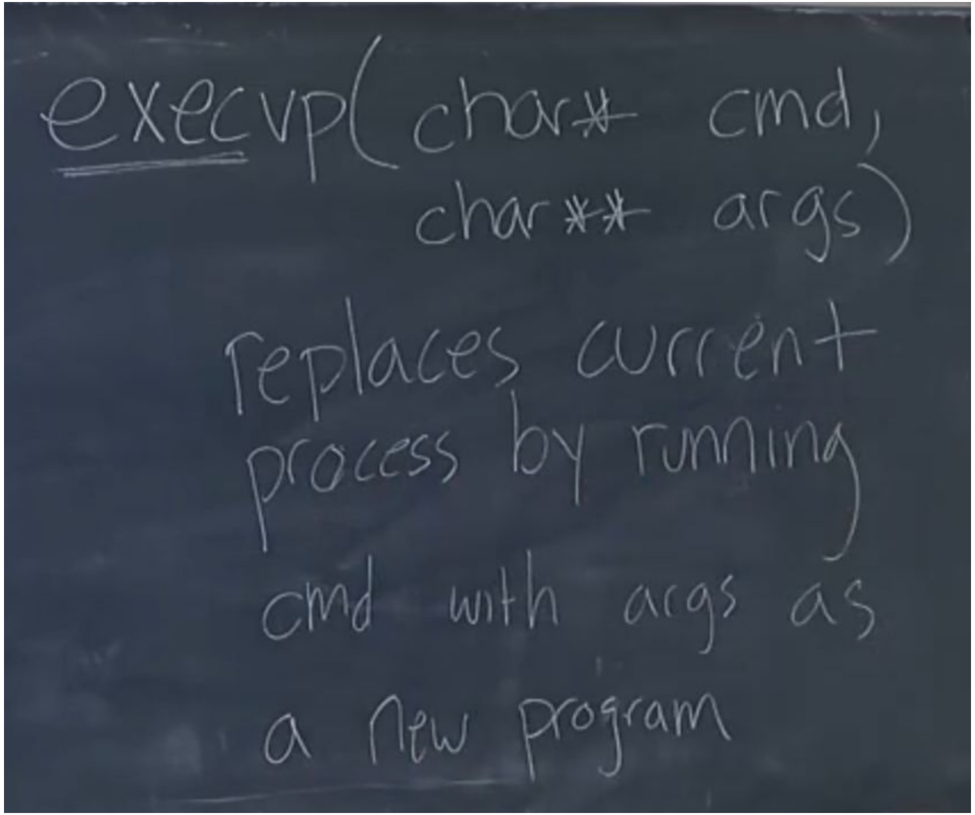

```
...... happy c code
fork()
```

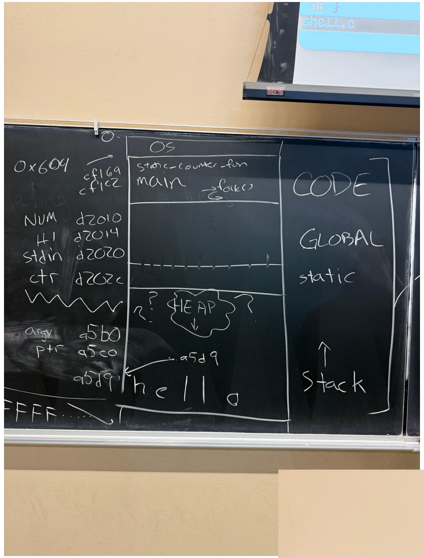
printf("hi")   printf("hi")

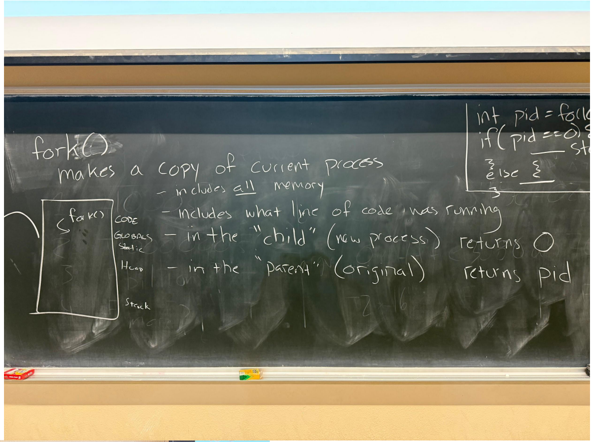
- same globals, same stack, etc.
- only difference is in "child" (new) process returns 0
  in "parent" (original) returns pid

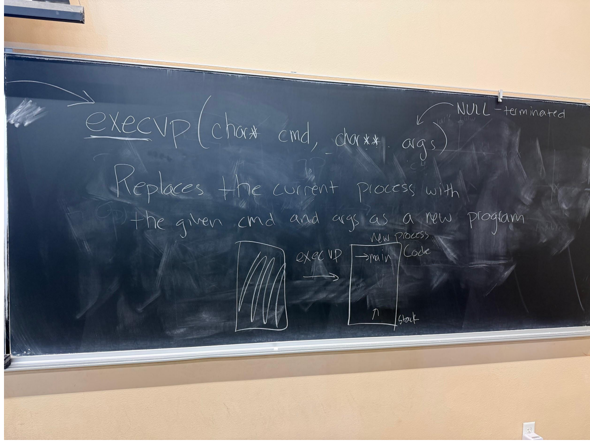
```
int pid = fork();
if (pid == 0) {
    // child stuff (new work)
}
else {
    // parent stuff (continue loop/monitoring/etc)
}
```

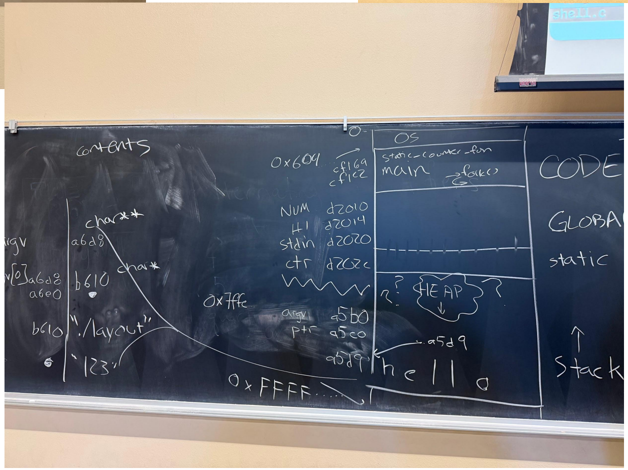
Joe’s Notes (12:30pm)

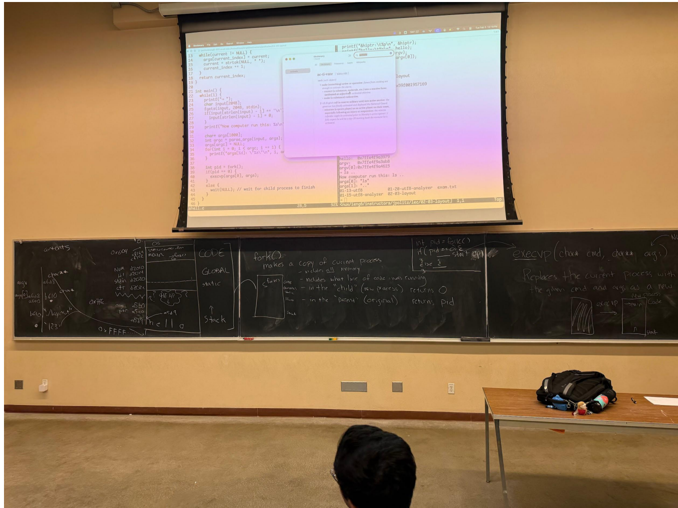


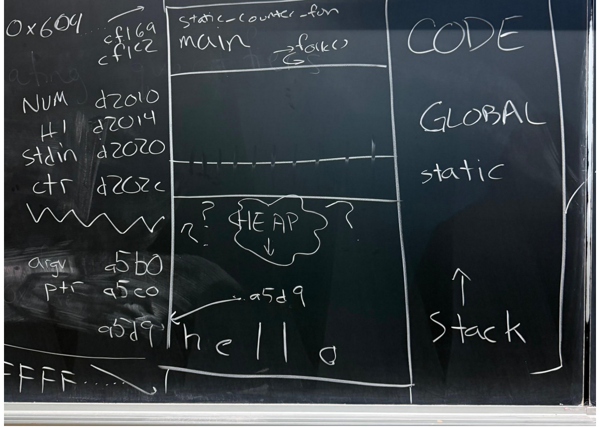

OS
static_counter_fun
main
fork()
CODE
GLOBAL
static
0x604
cf169
cf1c2
NUM d2010
#1 d2014
stdin d2020
ctr d202c
HEAP
argv a5b0
ptr a5c0
a5d9
a5d9
h e l l o
Stack
FFFF


fork()
makes a copy of current process
- includes all memory
- includes what line of code was running
- in the "child" (new process) returns 0
- in the "parent" (original) returns pid
int pid = fork
if (pid == 0)
else
CODE
GLOBALS
static
Heap
Stack

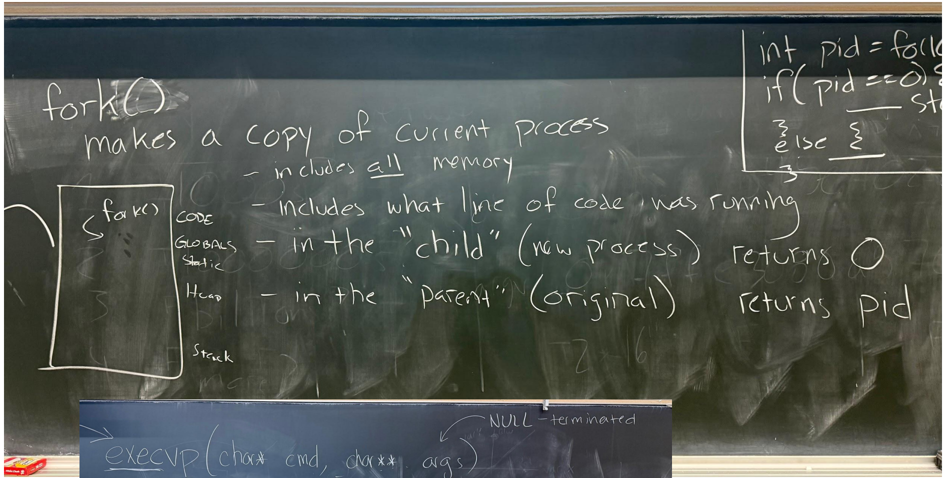

execvp(char* cmd, char** args)
NULL-terminated
Replaces the current process with the given cmd and args as a new program
execvp
new process
main
code
stack


contents
char**
argv
a6d8
char*
b610
a6e0
"./layout"
"123"
0x7ffe
0x604
cf169
cf1c2
NUM d2010
#1 d2014
stdin d2020
ctr d202c
argv a5b0
ptr a5c0
a5d9
a5d9
h e l l o
0xFFFF
OS
static_counter_fun
main
fork()
HEAP
CODE
GLOBA
static
Stack

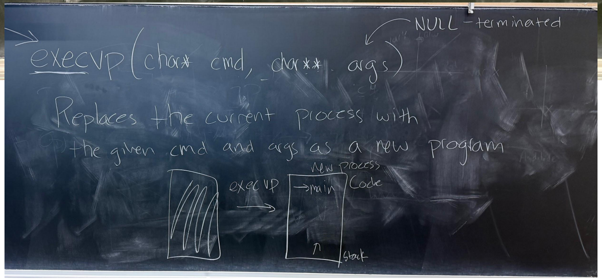




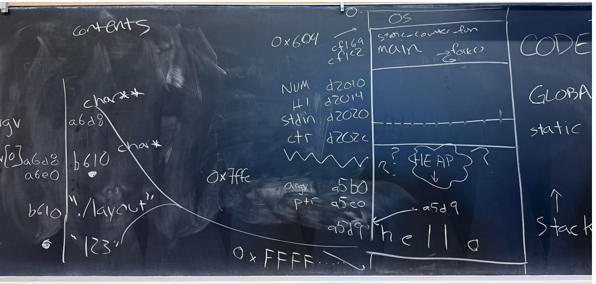
0x604
cf169
cf162
static_counter_fun
main
fork()
CODE
NUM d2010
H1 d2014
stdin d2020
ctr d202c
GLOBAL
static
HEAP
argv a5b0
ptr a5c0
a5d9
a5d9 h e l l o
Stack
FFFF.....



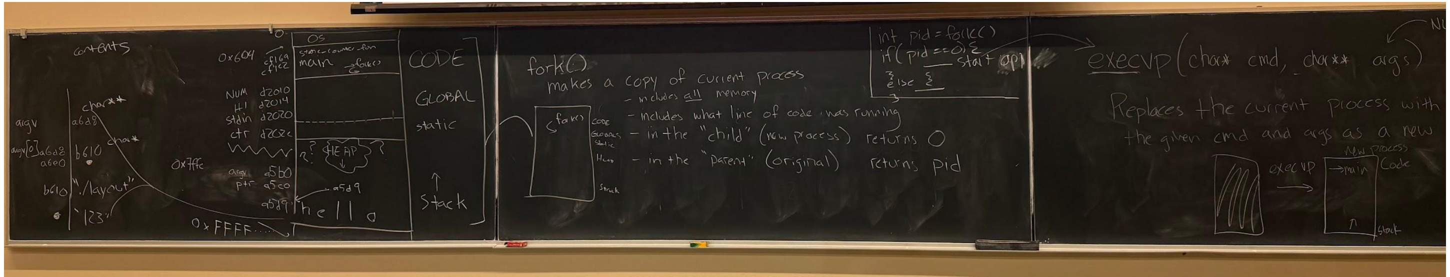
fork()
makes a copy of current process
- includes all memory
- includes what line of code was running
- in the "child" (new process) returns 0
- in the "parent" (original) returns pid
fork()
CODE
GLOBALS
Static
Heap
Stack
int pid = fork
if(pid == 0)
else



NULL-terminated
execvp(char* cmd, char** args)
Replaces the current process with
the given cmd and args as a new program
execvp
New process
main
Code
Stack



contents
OS
char**
a6d8
char*
b610
b610 "./layout"
"123"
0x7ffc
0x604
cf169
cf162
static_counter_fun
main
fork()
CODE
NUM d2010
H1 d2014
stdin d2020
ctr d202c
static
HEAP
argv a5b0
ptr a5c0
a5d9
a5d9 h e l l o
Stack
0xFFFF.....



contents
char**
char*
argv
b610
"./layout"
"123"
0x7ffc
0x604
main
CODE
GLOBAL
static
HEAP
h e l l o
Stack
0xFFFF.....
fork()
makes a copy of current process
- includes all memory
- includes what line of code was running
- in the "child" (new process) returns 0
- in the "parent" (original) returns pid
int pid = fork()
if(pid == 0)
else
execvp(char* cmd, char** args)
Replaces the current process with
the given cmd and args as a new
execvp
main
Code
Stack