CSE 29
Lecture 10 Summary

February 5, 2026

📣 Logistical Things

- No updates

🧠 Review Questions

Answers in speaker notes and explained in next slides!

```c
// Question 1

// in fork2.c
#include <stdio.h>
#include <unistd.h>

int main() {
  int pid1 = fork();
  int pid2 = fork();
  printf("The pids are: %d %d\n", pid1, pid2);
}


$ gcc fork2.c -o fork2
$ ./fork2

What prints?

```

```c
// Question 2 (just 2 questions)

// In exec-arg0.c
#include <unistd.h>
#include <stdio.h>

int main() {
  char* args[] = { "alpaca", NULL };
  execvp("./arg0", args);
  printf("Done executing\n");
}

// In arg0.c
#include <stdio.h>

int main(int argc, char** args) {
  printf("The 0th arg is: %s\n", args[0]);
}

$ gcc arg0.c -o arg0
$ gcc exec-arg0.c -o exec-arg0
$ ./exec-arg0

What prints?

```

Q1
The pids are: 102411 102412
The pids are: 0 102413
The pids are: 102411 0
The pids are: 0 0

(positive integers may differ but will be consistently the same as and different from each other.)

Q2
The 0th arg is: alpaca

🧠 RQ1 Answer

```
Original Process

int pid1 = fork();
int pid2 = fork();
printf(...);


Original Process                                Child Process c1
pid1 = <pid of c1>                              pid1 = 0
int pid2 = fork();                              int pid2 = fork();
printf(...)                                     printf(...)

Original Process      Child Process c2          Child Process c1      Child Process c3
pid1 = <pid of c1>    pid1 = <pid of c1>        pid1 = 0              pid1 = 0
pid2 = <pid of c2>    pid2 = 0                  pid2 = <pid of c3>    pid2 = 0
printf(...)           printf(...)               printf(...)           print(...)

12284 12285           12284 0                   0 12287               0 0

```

🧠 RQ2 Answer

The 0th arg is: alpaca

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

int main(){
    char* args[] = { "alpaca", NULL};
    execvp("./arg0", args);
    printf("Done executing\n");
}

```

- Why does line 7 not print?
  - execvp runs on line 6, the entire program is replaced by the args0 program

So, how would we get line 7 to print?

Feed in a program that doesn't exist into execvp

Exit Codes from returning from main()

Why do we always `return 0` at the end of `main()`?

- This is an **exit code**
- After a process is done running, it returns an exit code to the operating system to tell it about the status of the exit
  - Did the process exit successfully?
  - Did the process exit with an error?
  - Was there a major error during the execution of the program?
- We `return 0` at the end of `main()` to tell the OS that we exited successfully!

```
int main(){
    //insert some code here
    return 0;
}
```

What happens if we `return 37` from `main()`?

```c
1 int main() {
2   return 37;
3 }
```

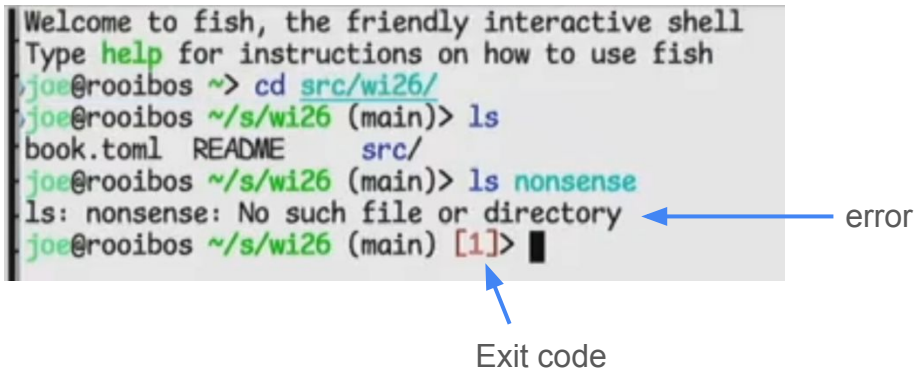
```
[jpolitz@ieng6-201]:02-05-fork-exec-more:500$ gcc re
return.c      reviewq1.txt  reviewq.txt
[jpolitz@ieng6-201]:02-05-fork-exec-more:500$ gcc re
return.c      reviewq1.txt  reviewq.txt
[jpolitz@ieng6-201]:02-05-fork-exec-more:500$ gcc return.c -o r
eturn
[jpolitz@ieng6-201]:02-05-fork-exec-more:501$ ./return
[jpolitz@ieng6-201]:02-05-fork-exec-more:502$ echo $?
37
[jpolitz@ieng6-201]:02-05-fork-exec-more:503$
```

return.c

- We can do `echo $?` to see the **exit code**
  - `$?` is bash code for the exit code from the last process ran
- **Exit codes** - an integer value that a program returns to the operating system or parent process when it terminates
  - 0 - exited successfully
  - Non-zero - there was an error

Just for fun: Joe likes the fish shell

Notice that the last exit code of `[1]` is displayed in the prompt, as would any non-zero value since those are indicative of errors



Questions

- Do different non-zero return codes mean different kinds of errors
  - There is not a universal standard for what error exit codes are, but it is universal that non-zero return codes does mean some kind of error
  - Let's look at the man pages of `ls`

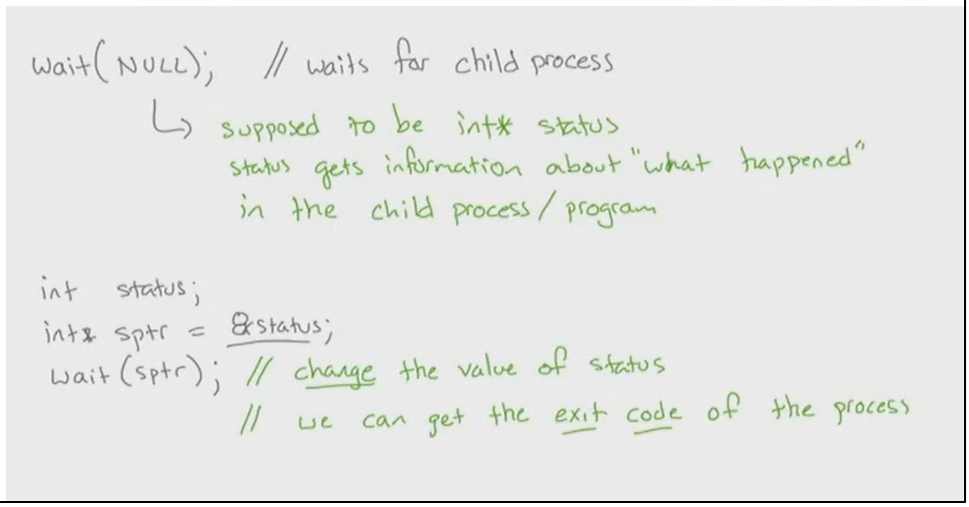
```
Exit status:
    0      if OK,

    1      if minor problems (e.g., cannot access subdirectory),

    2      if serious trouble (e.g., cannot access command-line argument).
```

How to get status of child process

If we feed `wait()` an `int *` (int pointer), we'll be able to see the exit status code after the wait call



(from previous lectures)

Lets try getting an error from processes in our shell

```
23    printf("→ ");
24    char input[2048];
25    fgets(input, 2048, stdin);
26    if(input[strlen(input) - 1] == '\n') {
27      input[strlen(input) - 1] = 0;
28    }
29    printf("Now computer run this: %s\n", input);
30
31    char* args[1000];
32    int argc = parse_args(input, args);
33    args[argc] = NULL; // execvp expects a NULL
34                       // at the end of the arg list
35    for(int i = 0; i < argc; i += 1) {
36      printf("args[%d]: \"%s\"\n", i, args[i]);
37    }
38
39    int pid = fork();
40    if(pid == 0) {
41      execvp(args[0], args);
42    }
43    else {
44      int status;
45      int* sptr = &status;
46      wait(sptr); // waits for child process
47      // Here we should be able to inspect status
48      printf("Status was: %d %b\n", status, status);
49    }
50  }
51 }
```

```
                                    unsigned int
[jpolitz@ieng6-201]:02-05-fork-exec-more:516$ gcc shell.c -o sh
ell
[jpolitz@ieng6-201]:02-05-fork-exec-more:517$ ./shell
→ ./return
Now computer run this: ./return
args[0]: "./return"
Status was: 9472 10010100000000
→ ls nonsense
Now computer run this: ls nonsense
args[0]: "ls"
args[1]: "nonsense"
ls: cannot access 'nonsense': No such file or directory
Status was: 512 1000000000
→ ls
Now computer run this: ls
args[0]: "ls"
arg0       exec-arg0.c  fork2.c   reviewq1.txt  shell.c
arg0.c     exec-funny   return    reviewq.txt
exec-arg0  fork2        return.c  shell
Status was: 0 0
→ cat nonexistent
Now computer run this: cat nonexistent
args[0]: "cat"
args[1]: "nonexistent"
cat: nonexistent: No such file or directory
Status was: 256 100000000
→
```

37 * 256
= 9472

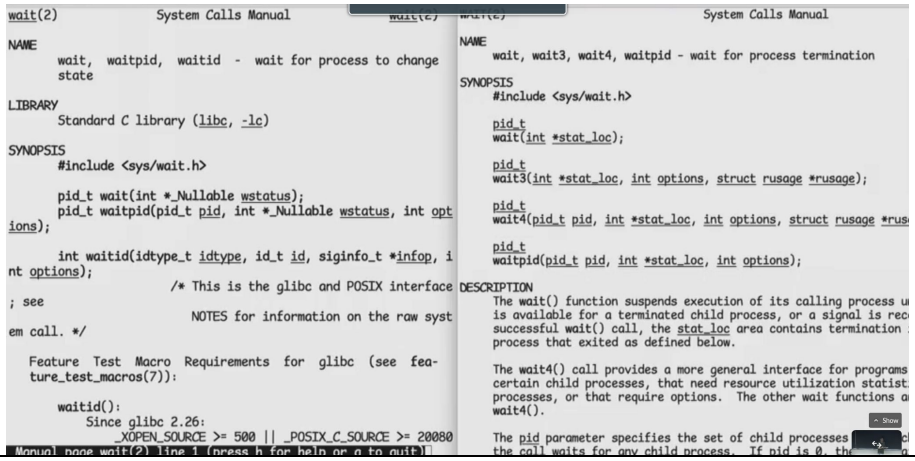
The exit statuses are multiplied by 256!

So why the exit statuses multiplied by 256?

- 37 became 9472
  - 37 in binary is **100101**
- 2 became 512
  - 2 in binary is **10**
- This is because the first 8 least significant bits are used to store other information
  - 9472 in binary is **10010100000000**
  - 512 in binary is **1000000000**
- We can use WEXITSTATUS to get the correct exit code
  - int actualstatus = WEXITSTATUS(status)
  - Note that returning anything greater than 256 will just use the least significant byte. 280 -> 24 (280 % 256 = 24) for example

I wish I knew what these functions did, `man`

Google and LLMs have many answers but there are different versions of things. The `man`ual pages are Operating System (OS) specific. We observe here the `man wait` from ieng6 on the left and Joe's mac on the right. We see these are different, and using instructions for one on the other could result in odd bugs.



wait(2) System Calls Manual

NAME

wait, waitpid, waitid - wait for process to change state

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

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

pid_t wait(int *_Nullable wstatus);
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                    /* This is the glibc and POSIX interface; see
                       NOTES for information on the raw system call. */

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

waitid():
    Since glibc 2.26:
        _XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 20080
```

Manual page wait(2) line 1 (press h for help or q to quit)

System Calls Manual

NAME

wait, wait3, wait4, waitpid - wait for process termination

SYNOPSIS

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

pid_t
wait(int *stat_loc);

pid_t
wait3(int *stat_loc, int options, struct rusage *rusage);

pid_t
wait4(pid_t pid, int *stat_loc, int options, struct rusage *rus

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

DESCRIPTION

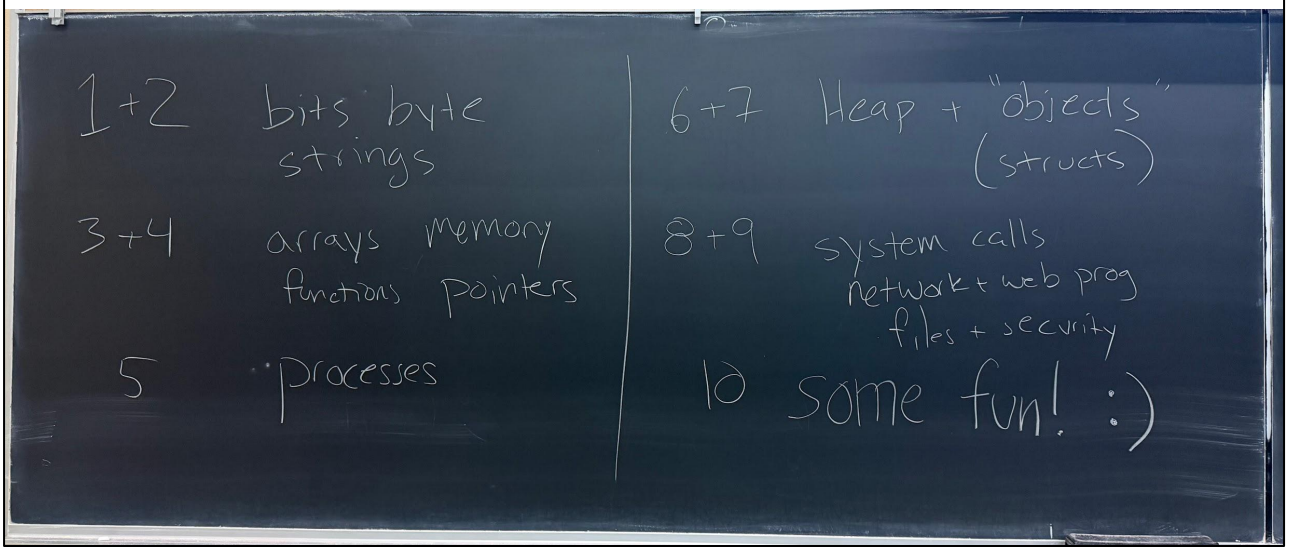
The wait() function suspends execution of its calling process u
is available for a terminated child process, or a signal is rec
successful wait() call, the stat_loc area contains termination
process that exited as defined below.

The wait4() call provides a more general interface for programs
certain child processes, that need resource utilization statist
processes, or that require options. The other wait functions a
wait4().

The pid parameter specifies the set of child processes
the call waits for any child process. If pid is 0, th

The Rest of the Quarter + Q&A

Timeline of the Course



- Week 1-2 : bits, bytes, C strings, numbers in binary
- Week 3-4: arrays, pointers, memory (stack)
- Week 5: Processes (fork + exec)
- Week 6-7: Heap, structs (“objects”)
- Week 8-9: system calls/OS, network/web programming, security
- Week 10: Recreational (Guest Lecturers, Fun lectures)

A Q&A session

- What kind of projects do you want to use C instead of a different programming language?
  - Any case where it's useful to have direct access to memory, including making a new programming language (CPython is written in C), small devices where it writes data to something (thermometer, solar panel, quadcopter …)
- Does C have a notion of exception handling?
  - No. Other than the exit status returning with 0 for success and nonzero for an error, there is not. (setjmp and longjmp mentioned but out of cse29 scope)
- What's an interesting usage of fork()?
  - Password cracking -> fork 25 times so you have a process for each letter and have 1 check all passwords starting with a, b, c and so on. Hardware is such that doing 2 at once is faster than doing 2 in a row
  - Don't they run on the same thread?
    - No threading in cse29 at any point, fork makes a copy of a *process*

Q&A (cont.)

- How do you communicate between processes?
  - Shared memory
  - msg passing (pipes)
- How does wait know which process to wait for?
  - Wait just waits for the first child to finish
  - waitpid() can be more specific. Check man page of wait using `man wait` for lots of info
- How many processes can be created?
  - 100000, 1000000, not 100% sure, depends on operating system, but somewhere in that range

Virtual Memory Sneak Peak

What would print here

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

int main() {
  int a = 100;
  printf("%d %p\n", a, &a);
  int pid = fork();
  if(pid == 0) { a = 999; }
  printf("[%d]: %d %p\n", pid, a, &a);

  // Virtual Memory

  // What will this print?

  // 100 0xfff71234

  // A
  // 999 0xfff71234
  // 999 0xfff71234

  // B
  // 999 0xfff71234
  // 999 0xfff71900


}

```

What was printed

```
$ gcc vm.c -o vm
$ ./vm
100 0x7fff050c1c40
[1431785]: 100 0x7fff050c1c40
[0]: 999 0x7fff050c1c40
$
```

Wait, that's the same address in the parent and the child?!

- It looks like the same address holds two different values at the same time!?!
- Joe has been saying that the addresses correspond to physical memory, but two different things can't be at the same address at the same time?
- What we learned: **Joe has been LYING**
- We'll learn more about this later in the quarter 🤗

Joe's Notes (11:00am)

```
wait(NULL);   // waits for child process
    ↳ supposed to be int* status
      status gets information about "what happened"
      in the child process / program

int status;
int* sptr = &status;
wait(sptr); // change the value of status
            // we can get the exit code of the process
```

Joe’s Notes (12:30pm)

No notes