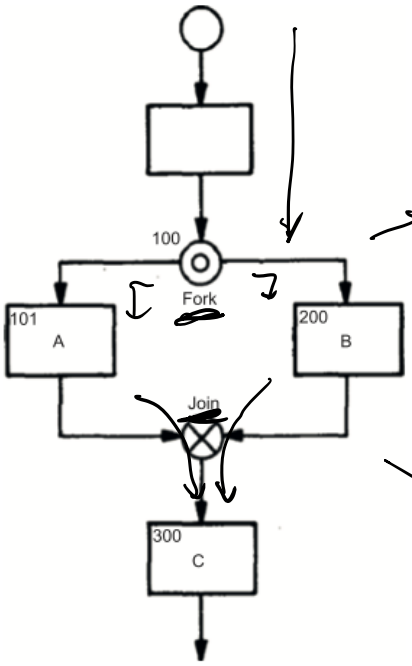


Figure 1. Fork system call. With this original illustration, Melvin Conway envisioned "fork and join" system calls in 1963.

→ fork()

child

pid_t

→ wait(int *wstatus)

"$ man wait"

1. from handout week2.a

1.a C code

```
...
31 uint64_t f(uint64_t* ptr)
32 {
33     uint64_t x = 0;
34     x = g(*ptr);
35     return x + 1;
36 }
37
38 uint64_t g(uint64_t a)
39 {
40     uint64_t x = 2*a;
41     q = &x; // <-- THIS IS AN ERROR (AKA BUG)
42     return x;
43 }
...
```

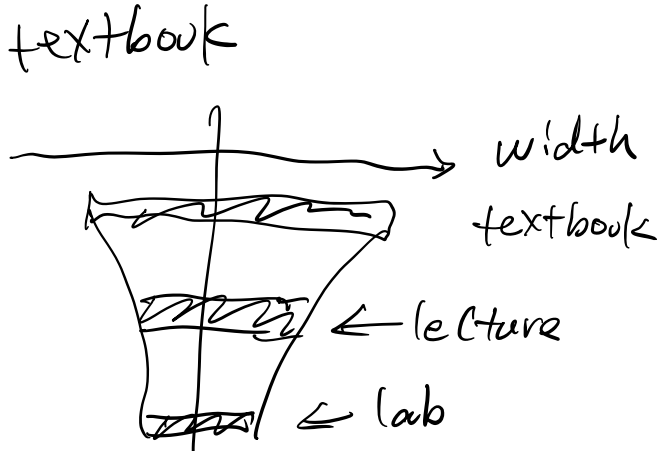
1.b assembly code (by "gcc -O0")

```
...
28  f:
29      pushq %rbp          # prologue: store caller's frame pointer
30      movq %rsp, %rbp     # prologue: set frame pointer for new frame
31
32      subq $32, %rsp      # make stack space
33      movq %rdi, -24(%rbp) # Move ptr to the stack
34                          # (ptr now lives at rbp - 24)
35      movq $0, -8(%rbp)   # x = 0 (x's address is rbp - 8)
36
37      movq -24(%rbp), %r8  # move 'ptr' to %r8
38      movq (%r8), %r9      # dereference 'ptr' and save value to %r9
39      movq %r9, %rdi       # Move the value of *ptr to rdi,
40                           # so we can call g
41
42      call g               # invoke g
43
44      movq %rax, -8(%rbp)  # x = (return value of g)
45      movq -8(%rbp), %r10  # compute x + 1, part I
46      addq $1, %r10        # compute x + 1, part II
47      movq %r10, %rax      # Get ready to return x + 1
48
49      movq %rpb, %rsp      # epilogue: undo stack frame
50      popq %rbp            # epilogue: restore frame pointer from caller
51      ret                  # return
...
```

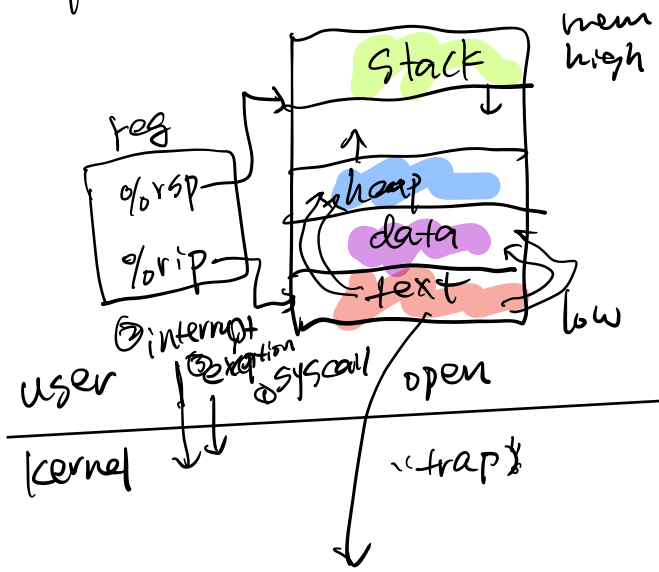

2. "gcc -O3 -S example.c"

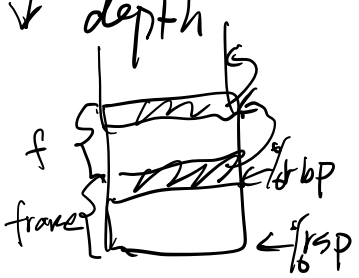
```
...
f:
    subq    $24, %rsp       // push frame: allocate stack frame 24B
                            // [Security]
    movq    %fs:40, %rax    // canary value (%fs:40) for detecting stack
    movq    %rax, 8(%rsp)   // smashing attacks
                            // put canary at 8(%rsp)
    xorl    %eax, %eax      // %eax=0 [%eax is the low 32bit of %rax]
    movq    %rsp, %rdx      // %rdx = %rsp
    movq    (%rdi), %rax    // %rax = *ptr (%rdi contains the first arg to
                            // function f, which is "ptr")
    movq    %rdx, q(%rip)   // "q" is the global variable; set "q" to %rdx:

                            // [Security]
    movq    8(%rsp), %rcx   // copy the canary at stack
    xorq    %fs:40, %rcx    // check if the value has been changed
    jne     .L9             // if so, alert!!!

    leaq    1(%rax,%rax), %rax  // %rax = %rax + %rax + 1
                                // (!!! this is function g!)
    addq    $24, %rsp       // pop frame
    ret                     // return to main
...
```

0. Admin
1. Last time
2. Process birth
3. Shell, crash course
4. Shell internals I
5. File descriptors
6. Shell internals II

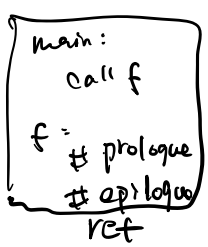


process.





- global vars → data
- local vars → stack
- heap
  malloc / free



Calling Conventions

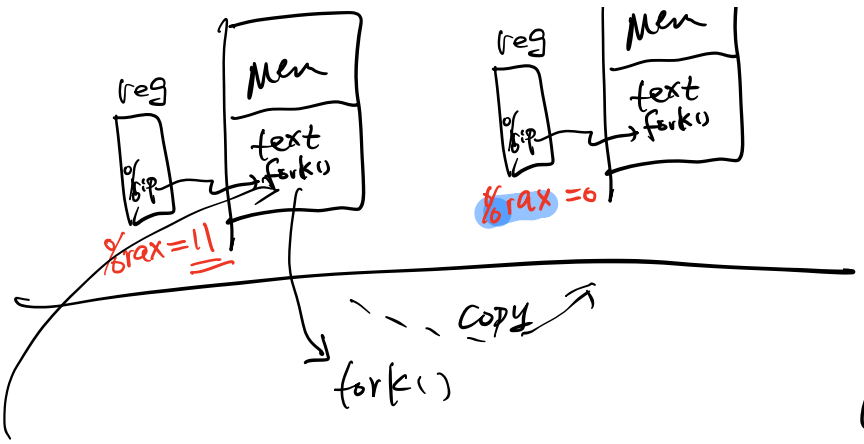
---

process birth

syscall: fork()

process 10 (parent) process 11 (child)

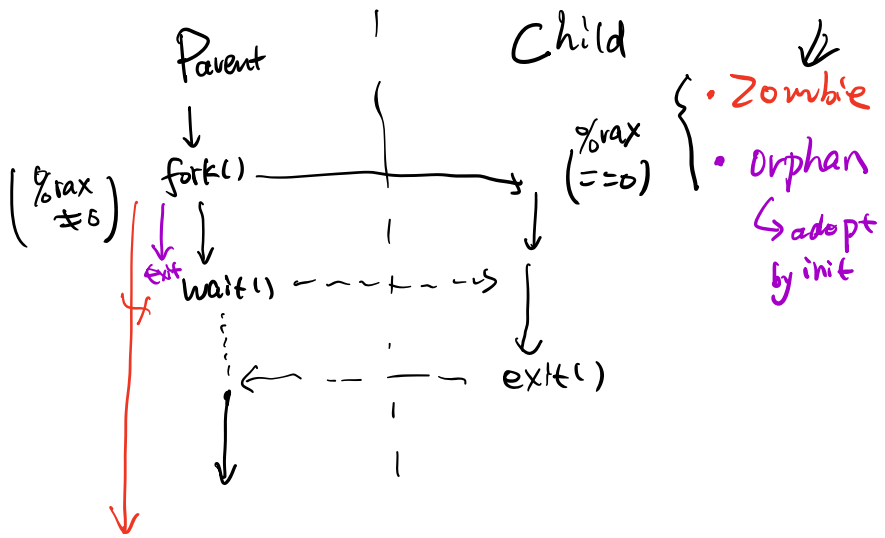
only one diff



CreateProcess(...)

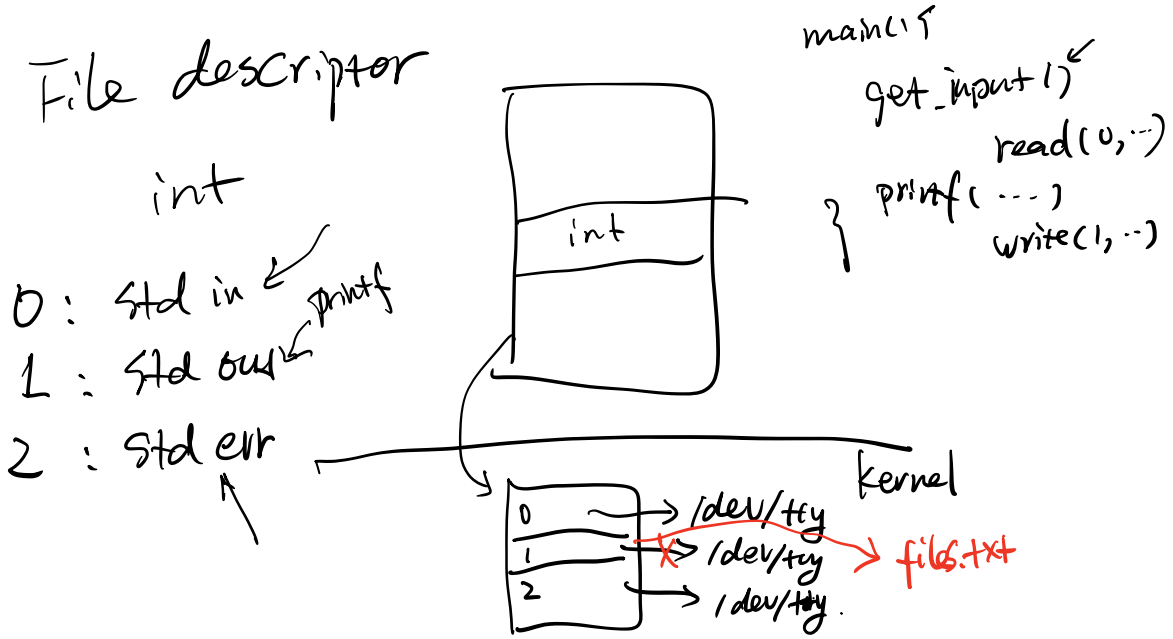
↳ fork/exec separation

```
if (fork()
        ==0) {
    // do A   ← process 1
} else {       ← process 1o
    // do B
}
```

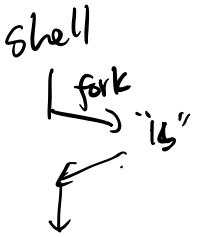


$ ./hellworld ↙

↑ shell


File descriptor
int
0 : Std in
1 : Std out
2 : Std err
printf
int
main()
get_input()
read(0, ...)
printf(...)
write(1, ...)
kernel
0
1
2
/dev/tty
/dev/tty
/dev/tty
file.txt

Crash course, Shell



1. run cmd
   "$ ls" and "$ ls -a"

2. output redirection
   "$ ls" prints to screen
   "$ ls > files.txt"

bash
zshell

3. backgrounding
   "$ web-server &
   $ "

4. pipe
   -- "$ cat students.txt | shuf -n 1"
   -- equivalent to
      "$ cat students.txt > /tmp/tmpfile
       $ shuf -n 1 /tmp/tmpfile
       $ rm /tmp/tmpfile"

5. Shell builtin cmds vs. program
   -- "echo/pwd/which" vs. "ls"
   -- use "which" to tell
      program: "$ which ls"   => "/bin/ls"
      built-in: "$ which which" => "which: shell built-in command"

"cd"

```
CS5600
Handout week03a

The handout is meant to:

  --illustrate how the shell itself uses syscalls

  --communicate the power of the fork()/exec() separation

  --give an example of how small, modular pieces (file descriptors,
  pipes, fork(), exec()) can be combined to achieve complex behavior
  far beyond what any single application designer could or would have
  specified at design time.

1. Pseudocode for a very simple shell

    while (1) {
      write(1, "$ ", 2);
      readcommand(command, args); // parse input
      if ((pid = fork()) == 0) {  // child?
        execve(command, args, 0);
      } else if (pid > 0) {       // parent?
        wait(0);                  //wait for child
      } else {
        perror("failed to fork");
    }}

2. Now add two features to this simple shell: output redirection and
   backgrounding

    By output redirection, we mean, for example:
        $ ls > list.txt
    By backgrounding, we mean, for example:
        $ myprog &
        $

    while (1) {
      write(1, "$ ", 2);
      readcommand(command, args);   // parse input
      if ((pid = fork()) == 0) {    // child?
        if (output_redirected) {
          close(1);
          open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
        }
        // when command runs, fd 1 will refer to the redirected file
        execve(command, args, 0);
      } else if (pid > 0) {       // parent?
        if (foreground_process) {
          wait(0);                //wait for child
        }
      } else {
        perror("failed to fork");
      }
    }
```



```
3. Another syscall example: pipe()

  The pipe() syscall is used by the shell to implement pipelines, such as
      $ ls | sort | head -4
  We will see this in a moment; for now, here is an example use of
  pipes.

  // C fragment with simple use of pipes

  int fdarray[2];
  char buf[512];
  int n;

  pipe(fdarray);
  write(fdarray[1], "hello", 5);
  n = read(fdarray[0], buf, sizeof(buf));
  // buf[] now contains 'h', 'e', 'l', 'l', 'o'

4. File descriptors are inherited across fork

  // C fragment showing how two processes can communicate over a pipe

  int fdarray[2];
  char buf[512];
  int n, pid;

  pipe(fdarray);
  pid = fork();
  if(pid > 0){
    write(fdarray[1], "hello", 5);
  } else {
    n = read(fdarray[0], buf, sizeof(buf));
  }
```

5. Putting it all together: implementing shell pipelines using fork(), exec(), and pipe().

```
// Pseudocode for a Unix shell that can run processes in the
// background, redirect the output of commands, and implement
// two element pipelines, such as "ls | sort"

void main_loop() {

  while (1) {
    write(1, "$ ", 2);
    readcommand(command, args);     // parse input
    if ((pid = fork()) == 0) {      // child?
      if (pipeline_requested) {
        handle_pipeline(left_command, right_command)
      } else {
        if (output_redirected) {
          close(1);
          open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
        }
        exec(command, args, 0);
      }
    } else if (pid > 0) {           // parent?
      if (foreground_process) {
        wait(0);                    // wait for child
      }
    } else {
      perror("failed to fork");
    }
  }
}

void handle_pipeline(left_command, right_command) {

  int fdarray[2];

  if (pipe(fdarray) < 0) panic ("error");
  if ((pid = fork ()) == 0) {     // child (left end of pipe)

    dup2 (fdarray[1], 1);         // make fd 1 the same as fdarray[1],
                                  // which is the write end of the
                                  // pipe. implies close (1).
    close (fdarray[0]);
    close (fdarray[1]);
    parse(command1, args1, left_command);
    exec (command1, args1, 0);

  } else if (pid > 0) {           // parent (right end of pipe)

    dup2 (fdarray[0], 0);         // make fd 0 the same as fdarray[0],
                                  // which is the read end of the pipe.
                                  // implies close (0).
    close (fdarray[0]);
    close (fdarray[1]);
    parse(command2, args2, right_command);
    exec (command2, args2, 0);

  } else {
    printf ("Unable to fork\n");
}}
```


6. Commentary

Why is this interesting? Because pipelines and output redirection are accomplished by manipulating the child's environment, not by asking a program author to implement a complex set of behaviors. That is, the *identical code* for "ls" can result in printing to the screen ("ls -l"), writing to a file ("ls -l > output.txt"), or getting ls's output formatted by a sorting program ("ls -l | sort").

This concept is powerful indeed. Consider what would be needed if it weren't for redirection: the author of ls would have had to anticipate every possible output mode and would have had to build in an interface by which the user could specify exactly how the output is treated.

What makes it work is that the author of ls expressed their code in terms of a file descriptor:
write(1, "some output", byte_count);
This author does not, and cannot, know what the file descriptor will represent at runtime. Meanwhile, the shell has the opportunity, *in between fork() and exec()*, to arrange to have that file descriptor represent a pipe, a file to write to, the console, etc.