EC 440 – Introduction to Operating Systems

Manuel Egele

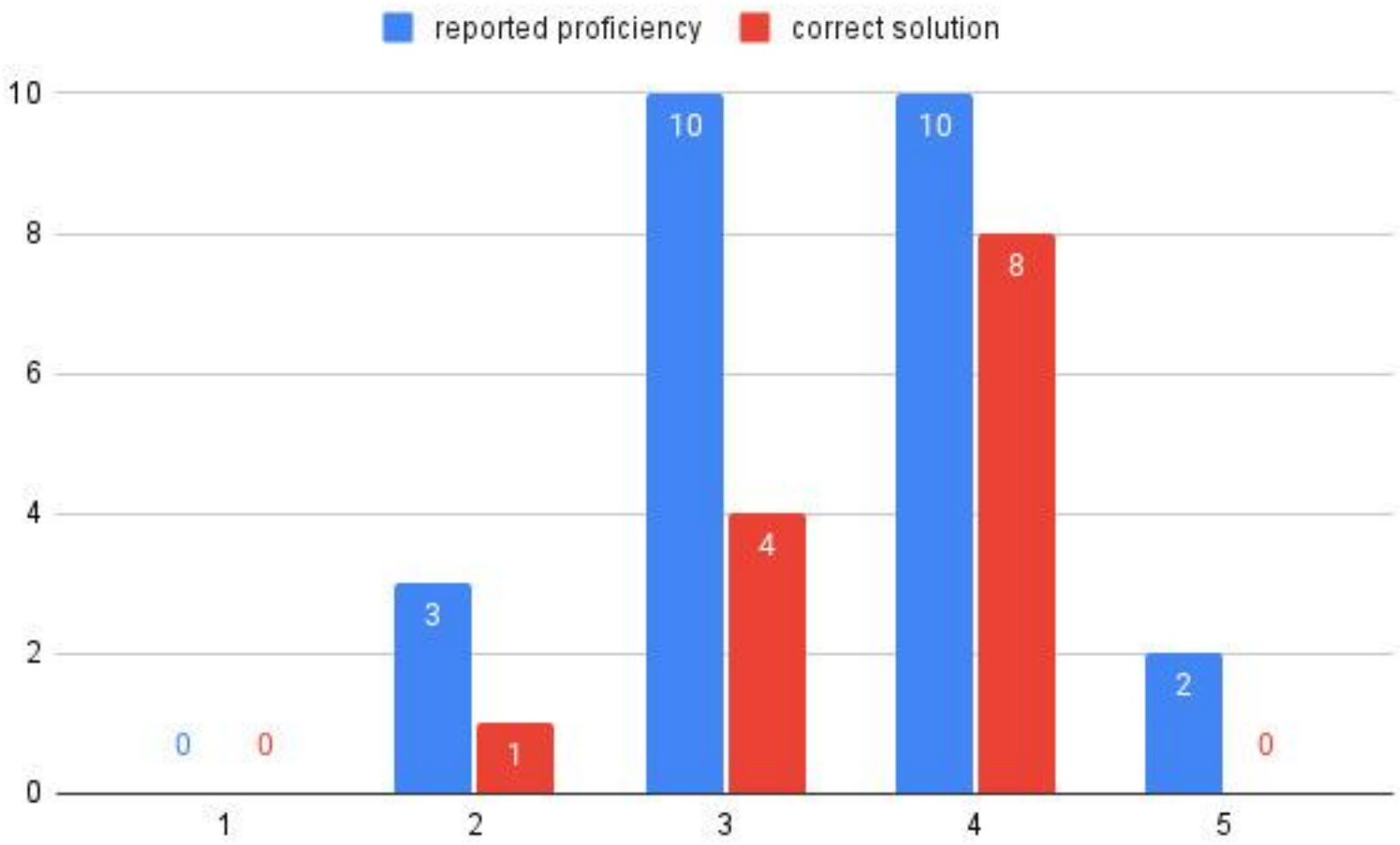
Department of Electrical &
Computer Engineering

Boston University

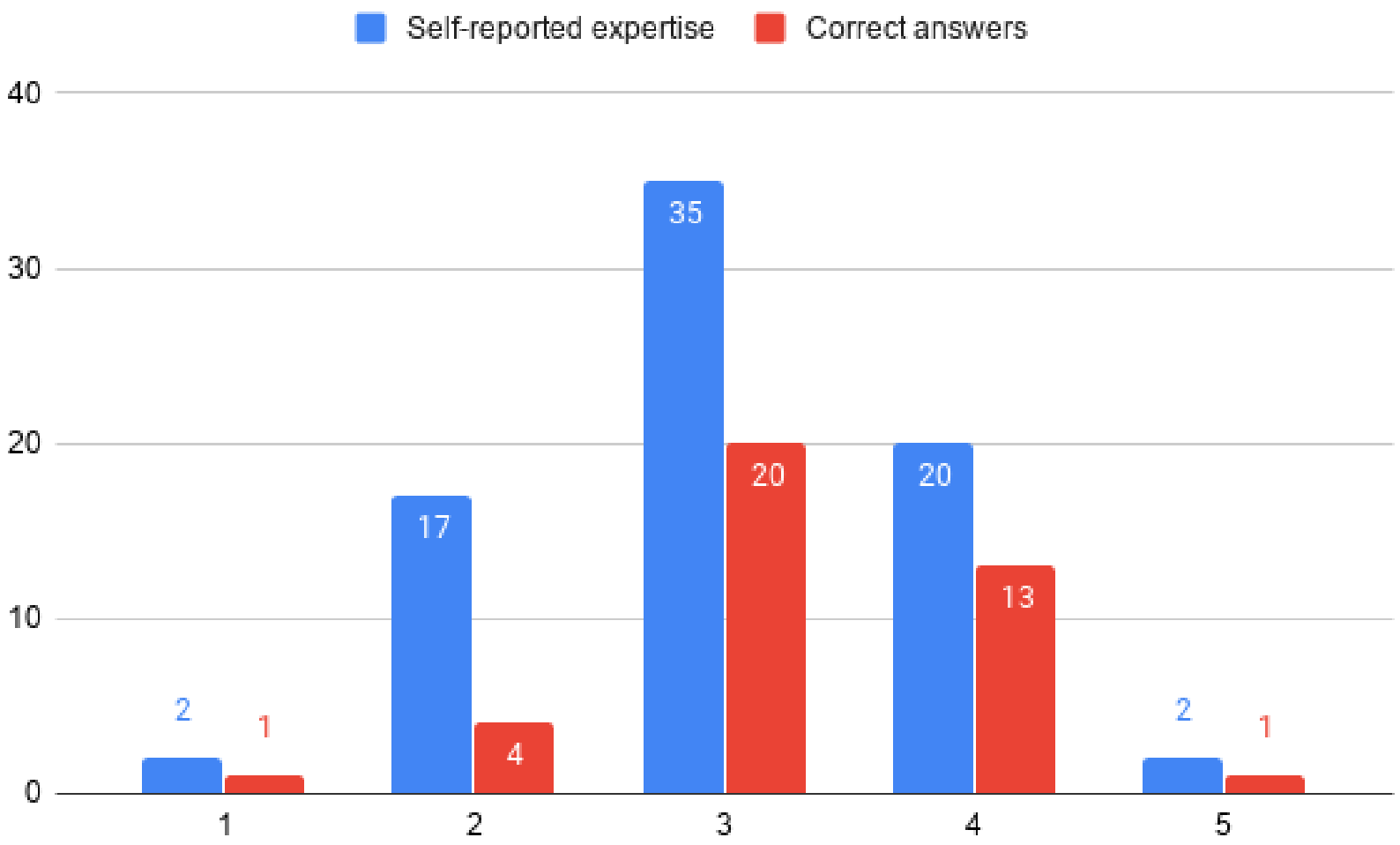
Simple C Program

```
 1  int main(int argc, char **argv) {
 2      char *str = (char *)malloc(7);
 3      char *p = str + 2;
 4      strncpy(str, "foobar", 6);
 5      p += 1;
 6      printf("str1 %s\n", str);
 7      *p += 1;
 8      printf("str2 %s\n", str);
 9      free(str);
10  }
```

Self-Assessment (2022)



Self-Assessment (2020)



Project Update

- Homework 1 has been distributed on Piazza
- Deadline1: Sept. 21st 23:59 ET (no extension)
- Deadline2: Oct. 4th 23:59 ET (no extension)
- Must work on lab server: ec440.bu.edu
- Connect as:
  `$ ssh -p 10001 terrierXXX@ec440.bu.edu`
- Should have gotten username/password in your BU email
  - If not, let us know!
- Learn about (how to use) "ssh public key authentication" e.g., https://do.co/1sUHGrL
- Password-based login will be disabled in one week (need help resetting ssh-key -> -1% on your grade)

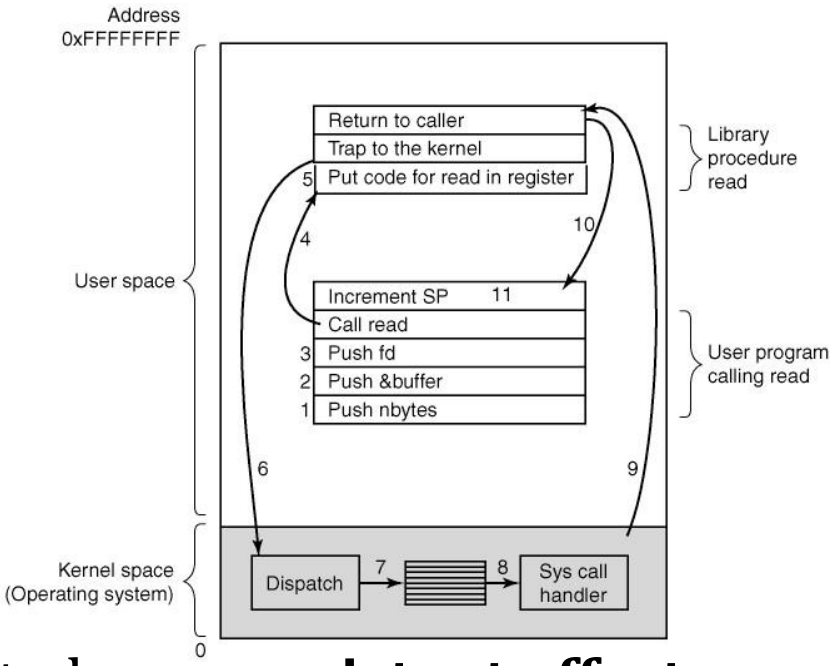
Overview

- System calls (definition and overview)
- Processes and related system calls
- Signals and related system calls
- Memory-related system calls
- Files and related system calls

System Calls

System calls are the interface to operating system services - they are how we tell the OS to do something on our behalf

- API to the OS
- Hide OS and hardware complexity from us



- The ***only*** way for a process to have **persistent effect** or communicate with the environment (e.g., file system, network, terminal, etc.) is via system calls.

System Call Interface

- Hardware & OS specific
- Special CPU instructions (hardware)
  - `SYSENTER/SYSCALL` (AMD64)
  - `int` (x86)

    0x80/Linux

    0x2e/Windows
  - `swi` (ARM)

System Call Arguments

Arguments (Hardware & OS specific)

- AMD64/Linux & *BSD
  - Syscall#: %rax
  - Args: %rdi, %rsi, %rdx, %r10, %r8, %r9
  - Result: %rax
- x86/Linux
  - Syscall#: %eax
  - Args: %ebx, %ecx, %edx, %esi, %edi, %ebp
  - Result: %eax
- x86/*BSD
  - Syscall#: %eax
  - Args: on the stack right-to-left
  - Result: %eax

System Calls (Example)

```
#include <unistd.h>

int main(int argc, char* argv[])
{
  int fd, nread;
  char buf[1024];

  fd = open("my_file",0);    /* Open file for reading */
  nread = read(fd,buf,1024);     /* Read some data */

  /* Presumably we do something with data here */

  close(fd);
}
```

System Calls vs. Libc

- Many system calls have corresponding mostly thin wrapper functions (i.e., with the same name) in libc (e.g., read, write, exec, wait, etc.)
- For the homework it's OK to use these wrappers in libc; you don't have to write the assembly code that is necessary to make proper system calls.

DEMO – Hello World!

System Calls

How the system calls communicate back to us?

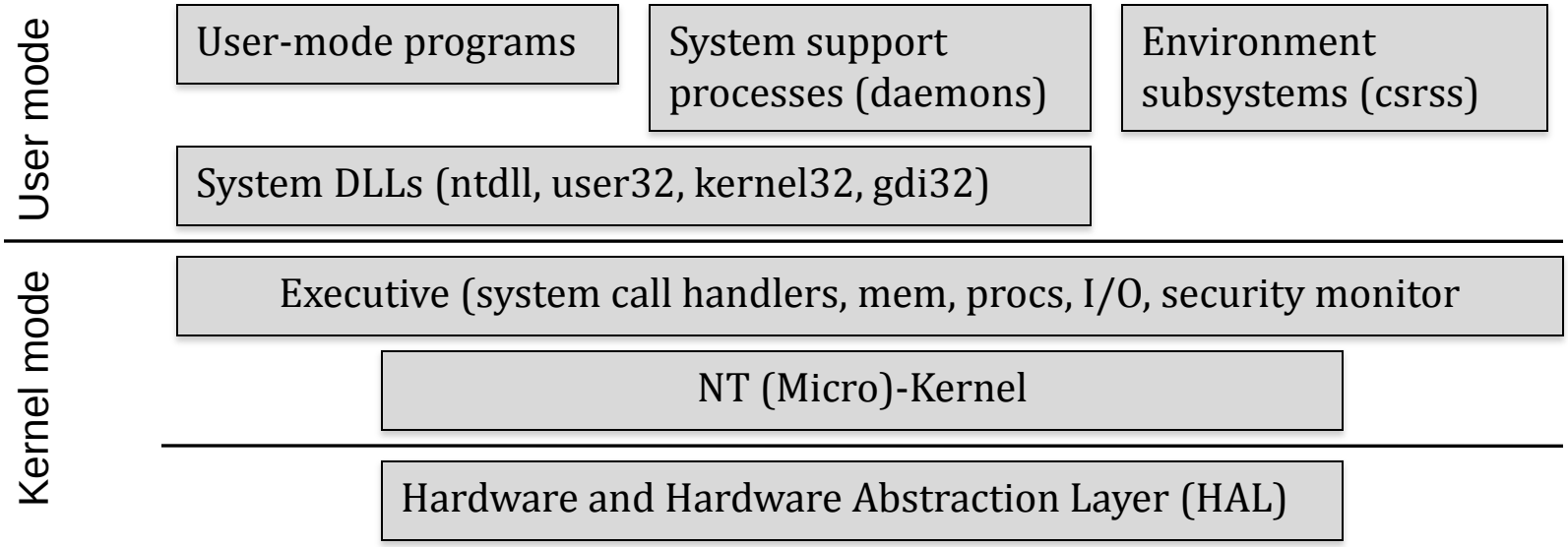
- Return value – usually return -1 on error, >= 0 on success
  - library functions set a global variable "errno" based on outcome
    - 0 on success,
    - positive values encode various kinds of errors
  - can use `perror` library function to get a string

- Buffers pointed to by system call arguments
  - e.g., in case of a read system call
  - values need to be copied between user and kernel space

Windows NT

Competitor to Unix

- Yes, Windows 11 is based on the NT kernel (mostly rewritten)
- true multi-user (as opposed to Windows < 9x)
- emphasis on portability and object-oriented design
- isolation for applications and resource access control
- similar to Unix, kernel and user mode



Processes

Concept

- processes - program in execution
  (along with all it's memory/data contents, process table entry, etc.)

Each process has its own memory space and process table entry

Process table entry

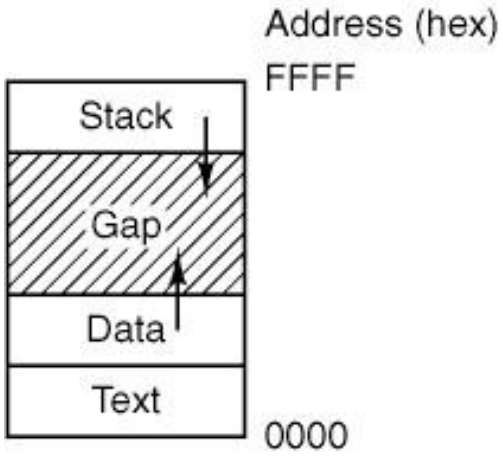
- stores all information associated with a process (except memory)
- register values, open files, user ID (UID), group ID (GID), ..

Processes are indexed by the process ID (PID)

- integer that indexes into process table

Processes

Memory layout



OS responsible for changing between multiple processes

Process System Calls

- **`fork`** (duplicate current process, create a new process)
- **`exec`** (replace currently running process with executable)
- **`exit`** (end process)
- **`wait`** (wait for a child process)
- **`getpid`** (get process PID)
- **`getpgrp`** (get process GID)

fork()

Syntax: `pid = fork();`

Get almost identical copy (child) of the original (parent)

- File descriptors, arguments, memory, stack … all copied
- Even current program counter
- But not completely identical - why?

Return value from `fork` call is different:

- 0 in child
- PID > 0 of the child when returning in parent

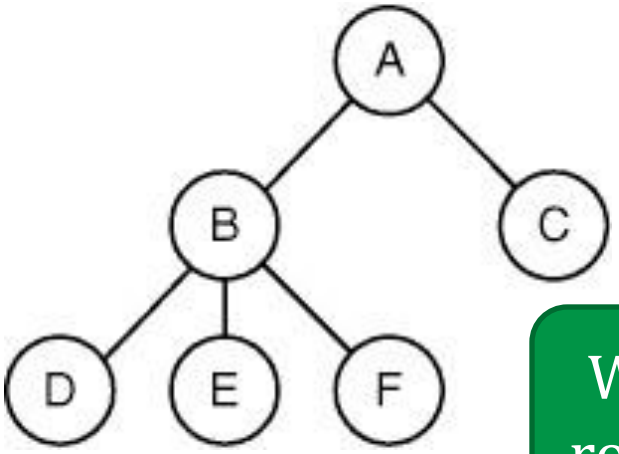
fork() cont.

```
int main(int argc, char* argv[])
{
   pid_t pid;
   if((pid = fork()) > 0)
   {
      /* Parent */
       printf(“hello parent\n”);
   } else {
       /* Child */
       printf(“hello child\n”);
   }
}
```

DEMO – fork()

Process Hierarchy

- Notion of a hierarchy (tree) of processes
- Each process has a single parent - parent has special privileges
- In Unix, all user processes have 'init' as their ultimate ancestor



Additional ways to group processes

- Process Groups (job control)
- Sessions (all processes for a user)

Why are sessions not always subtrees rooted at a process owned by the user?

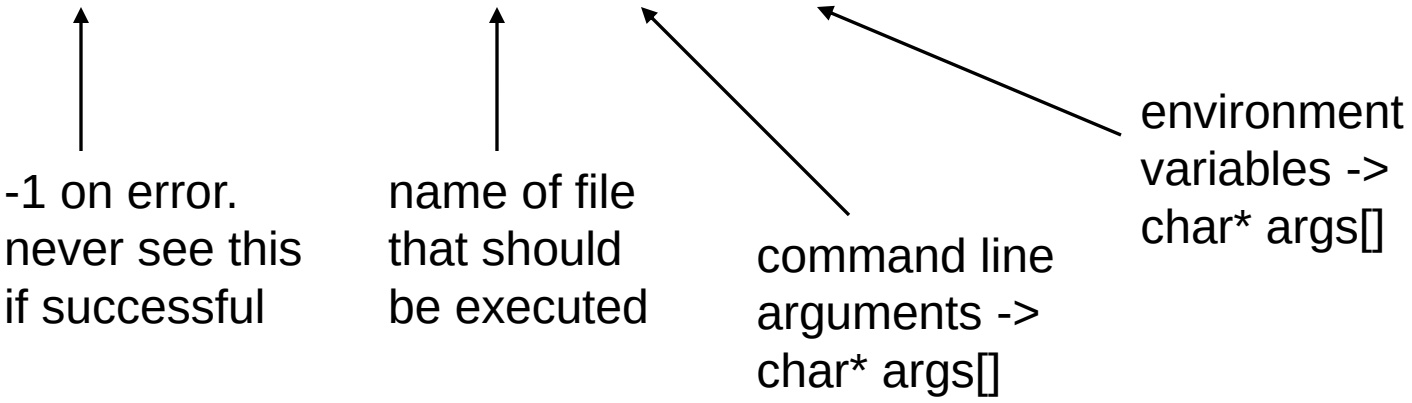
exec()

Change program in process

- i.e., launch a new program that **replaces** the current one

Several different forms with slightly different syntax

```
int status;
status = execve(prog, args, env);
```



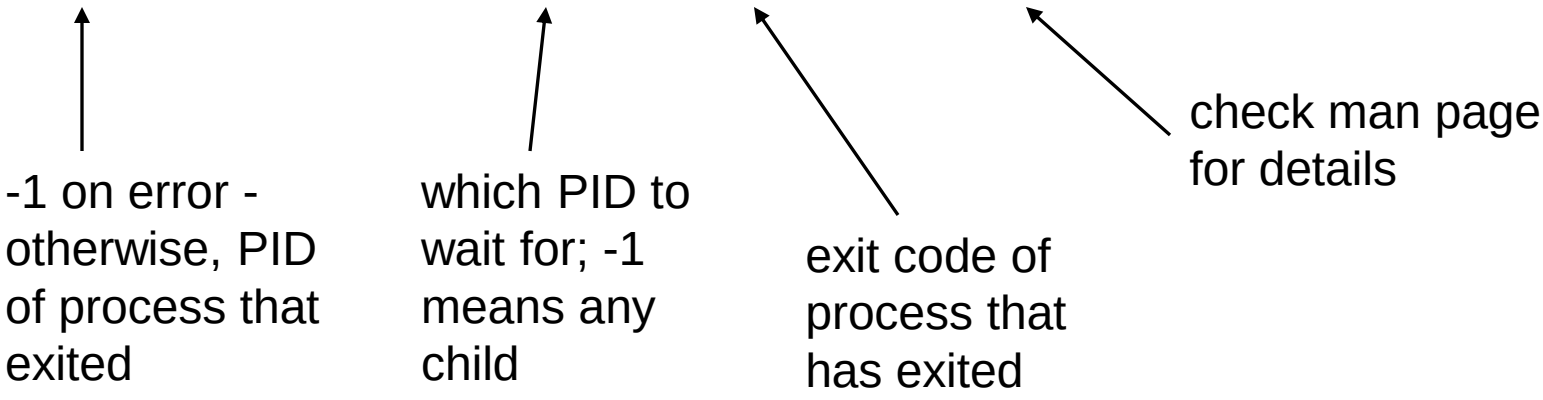
What does `execve` return?

DEMO – exec()

wait()

- When a process is done it can call `exit(status)`
- This is the status that "`echo $?`" can show you in the shell
- Parent can wait for its children (block until they are done)

```
status = waitpid(pid, &statloc, options);
```



Example

```
int main(int argc, char* argv[])
{
   pid_t pid;
   int status;
   char* ls_args[2];
   ls_args[0] = ".";
   ls_args[1] = 0;
   if((pid = fork()) > 0)
   {
      /* Parent */
      waitpid(pid,&status,0);
      exit(status);
   } else {
      /* Child */
      execve("/bin/ls", ls_args,0);
   }
}
```

What will happen & why?

DEMO – fork() & exec()

Shell

- Program that makes heavy use of basic process system calls
- Basic cycle REPL (read – eval – print – loop):
  1. prompt
  2. read line
  3. parse line
  4. fork (child execs the command, parent waits)

- Has to handle & (background job)
- Has to handle >, |, etc.,
  - i.e., somehow connect stdin and stdout of the child to files or other programs

Shell

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
      type_prompt( );                           /* display prompt on the screen */
      read_command(command, parameters);        /* read input from terminal */

      if (fork( ) != 0) {                       /* fork off child process */
           /* Parent code. */
           waitpid(–1, &status, 0);             /* wait for child to exit */
      } else {
           /* Child code. */
           execve(command, parameters, 0);      /* execute command */
      }
}
```

HW1 Write a simple shell that handles some special characters …

- `command < "filename"`
  - instead of reading from `stdin`, read from "filename"
- `command > "filename"`
  - instead of writing to `stdout`, write to "filename"
- `command1 | command2`
  - pipe `stdout` from cmd1 as `stdin` to cmd2
- `command &`
  - run command in background and don't wait until it completes before printing (and processing) the next prompt.

For Deadline 1 (next week), the special characters need not be handled yet (needed by the Oct. 4 deadline).

Example for HW1 – Deadline 1

- Normal shell (easy to try it yourself):

```
$ echo –e “a\nb\nc” | grep b | cat –n
1 b
```

command line

result

- Simple (deadline 1) shell:

```
$ echo –e “a\nb\nc” | grep b | cat –n
echo –e “a\nb\nc”
grep b
cat –n
a
b
c
```

individual commands of the command line

result of first command – i.e., echo –e “a\nb\nc”

Some More Process-Syscalls

`getpid()`

- Returns the pid of the calling process

`getppid()`

- Returns the pid of the parent
- How does the parent get the pid of the child? (i.e., the inverse of getppid())