Processes:
`fork()`, `exec()`

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

- How can we create a **new process?**
- How can we run a **different program?**

…and **why aren't these the same thing?**

Making a New Process With **`fork()`**

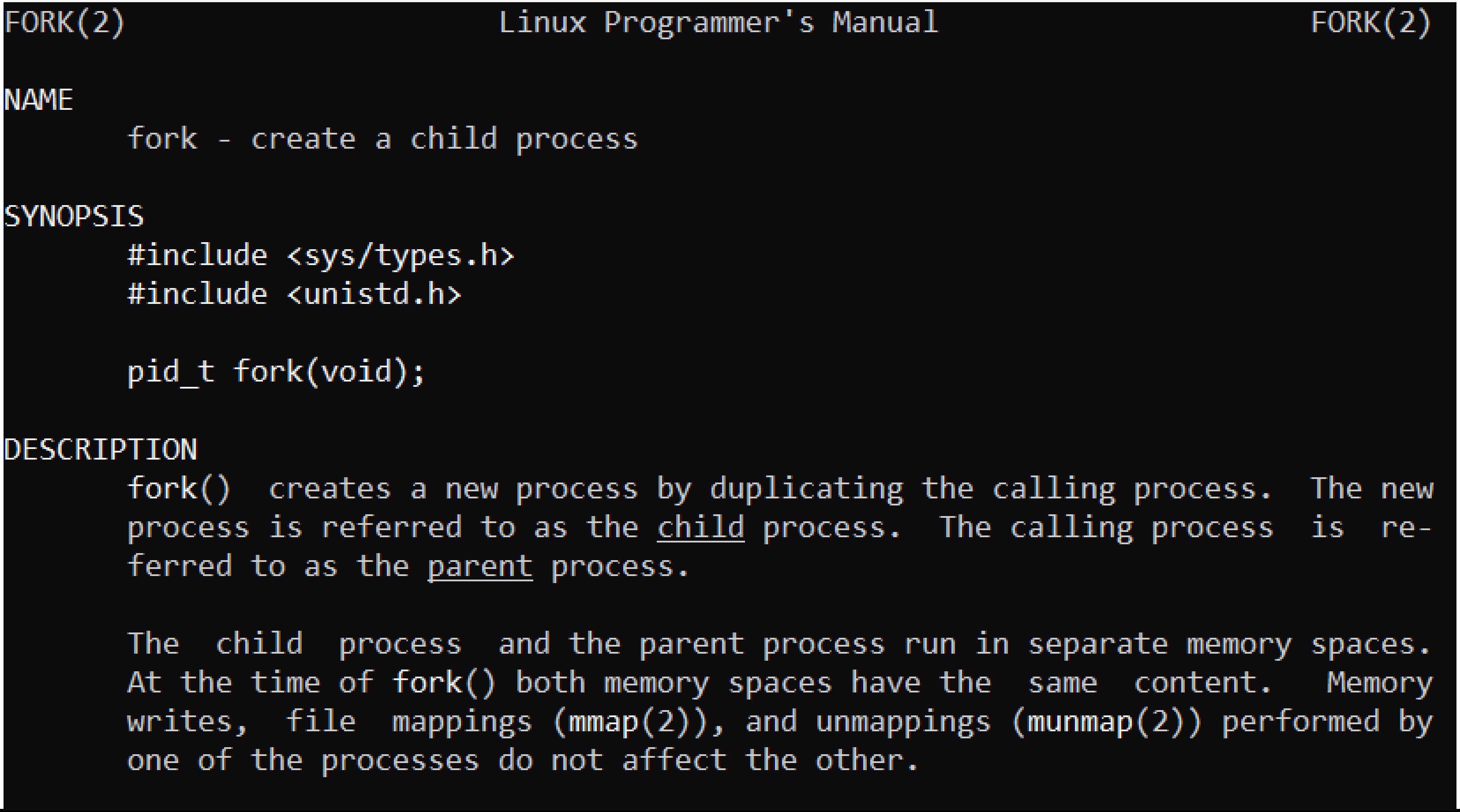
Making a New Process

- *Each process has its own address space*
  - Changing a variable's value in one process *does not affect any variables in the other process*
  - *Pointers in one process cannot access memory in another*
  - Processes can only communicate with each other through the OS, and only if they both agree how
- *Making a new process*
  - Initial process (the **parent**) wants to make a new process (the **child**)
  - Parent will call `fork()` to have the OS start a new process
  - `fork()` is a system call (syscall), as well as a POSIX function

`fork()`

- fork() creates a child process that is an *identical copy of the calling process* (except for one tiny detail...)
  - fork() is called once, but *it returns twice!*
    1) In the initial process **(parent),** just as we expect
    2) In the new process **(child)!**
- Analogy: It's like waking up after being cloned
  - Are you the original person?
  - Are you the clone?
- `fork()` returns a process ID (PID)
  - For the parent, the PID of the child (or -1 on failure)
  - For the child, `fork()` returns 0

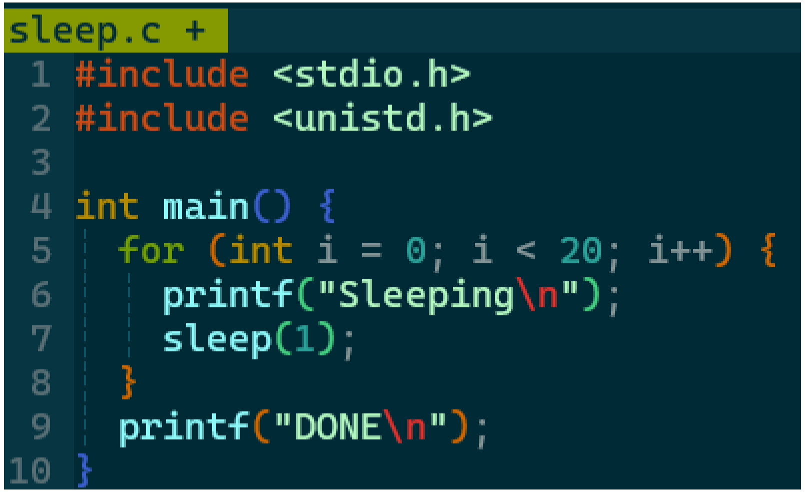
- Check its return value out



```
FORK(2)                    Linux Programmer's Manual                   FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates a new process by duplicating the calling process.  The new
       process is referred to as the child process.  The calling process  is  re-
       ferred to as the parent process.

       The  child  process  and the parent process run in separate memory spaces.
       At the time of fork() both memory spaces have the  same  content.   Memory
       writes,  file  mappings (mmap(2)), and unmappings (munmap(2)) performed by
       one of the processes do not affect the other.
```

Activity – `fork()`

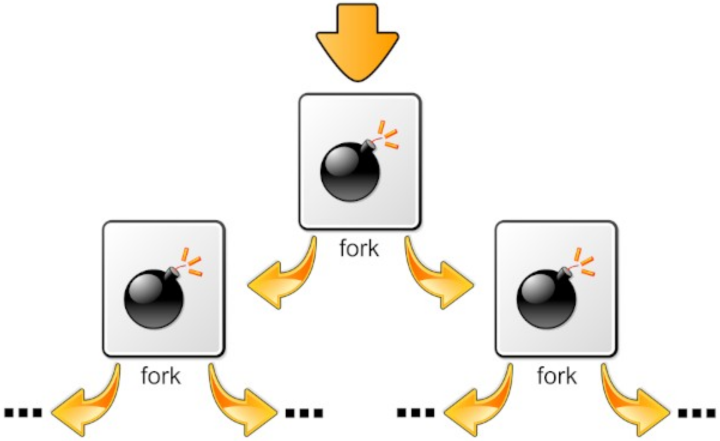
- *Write a program that* (5m)
    - *Calls fork()*
    - Keeps calling `sleep()` with some timeout value.
- *Hint*
    - Modify the `sleep()` example.
    - Get more info: man fork
    - You need to write one line of code.
- *Discussion*
    - Run it; check `btop` in tree mode
    - There should be a new child process
    - Look at the PID in `btop`
    - Kill both processes



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

int main() {
  for (int i = 0; i < 20; i++) {
    printf("Sleeping\n");
    sleep(1);
  }
  printf("DONE\n");
}
```

Activity – `fork()` Bomb!

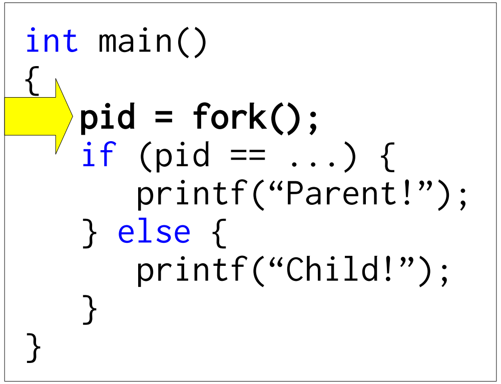
- *Write a fork bomb* (5m)
  - i.e., a program that continually calls fork()
  - **DO NOT RUN IT** (yet), but you can compile it
- *Demo fork bomb*
  - This might kill the container
  - Docker might not respond
  - Your computer might give up
- *Why did this happen?*
  - Each process calls `fork()`
  - Exponential growth in processes
  - Denial of service attack consuming kernel resources



Understanding fork()

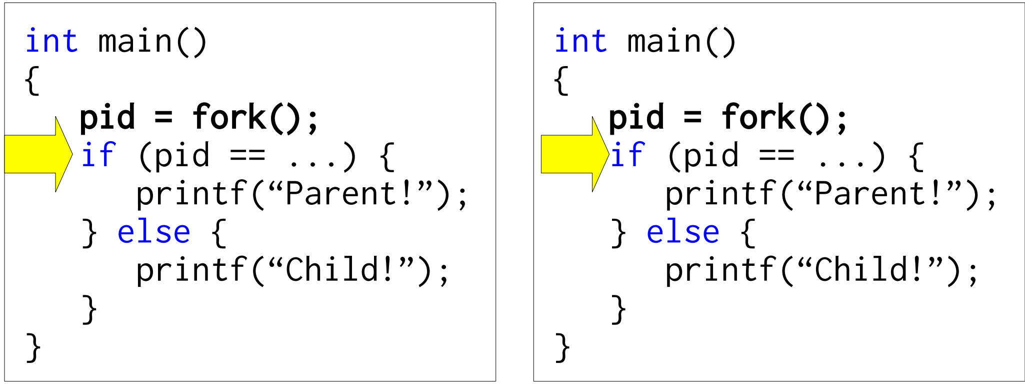
- *Understanding fork*
  - We have one C program, which clones itself with fork()
  - Until we call fork(), there is only one process.
- *fork() "returns twice"; once into each process*
  - The parent and the child are *the same program (same source file)*
  - After fork() each process executes independently
  - Both processes (and the shell!) all share the screen, so output gets mixed up

At the start: one process



```
int main()
{
   pid = fork();
   if (pid == ...) {
      printf("Parent!");
   } else {
      printf("Child!");
   }
}
```

After fork(): two processes

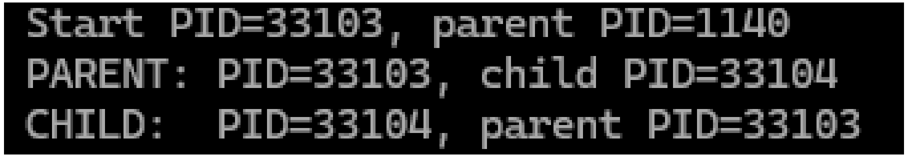


```
int main()
{
   pid = fork();
   if (pid == ...) {
      printf("Parent!");
   } else {
      printf("Child!");
   }
}
```

```
int main()
{
   pid = fork();
   if (pid == ...) {
      printf("Parent!");
   } else {
      printf("Child!");
   }
}
```

Activity - `fork()` with PIDs

- *Write a program that* (15m)
  1) Prints **its PID** and **its parent's PID**
     - “`man getpid`” and “`man getppid`” to get the PIDs
  2) Calls **fork()**
     - If *parent*: print "parent", its PID, and the child PID
     - If *child*: print "child", its PID, and the parent's PID



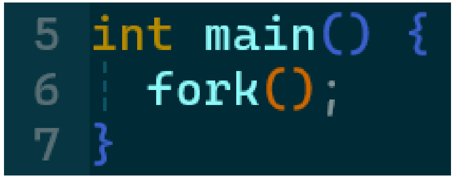
```
Start PID=33103, parent PID=1140
PARENT: PID=33103, child PID=33104
CHILD:  PID=33104, parent PID=33103
```

- *Hints*
  - This is a single program, but becomes multiple processes
  - The parent and the child need to do different things
  - Use “if-else” on the return value of “`fork()`” to differentiate the behaviour

Audience Participation - fork()

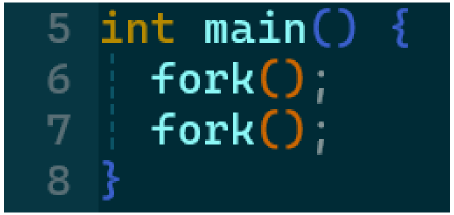
- How many processes will have been created by running this code (at least 1 for the original)?

a)



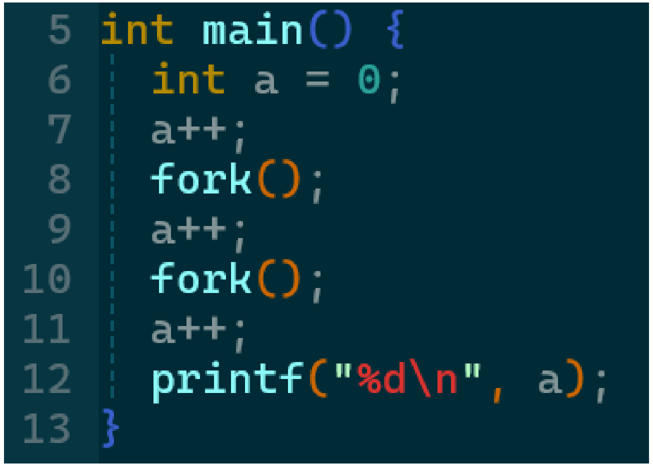
```c
int main() {
    fork();
}
```

b)



```c
int main() {
    fork();
    fork();
}
```

- What number will this code output?



```c
int main() {
    int a = 0;
    a++;
    fork();
    a++;
    fork();
    a++;
    printf("%d\n", a);
}
```

a) 2

b) 3

c) 4

d) 7

Bonus Activity

- *Write a program that*
  - Spawns 10 child processes.
  - Each child finds 10 big prime numbers.
  - Parent process waits 10s and exits.
    - While waiting, parent prints "Still waiting..." each second

Replace Current Program in Process With exec( )

Purpose of exec()

- *When called,* ***exec()*** *will*
  - *Remove the currently running program* from this process' memory
  - Load a new program into memory
  - Start executing the new program
- **exec()** completely replaces the calling process; it is replaced by a new program.

Audience Participation - exec( ) Idea

- What words will the following pseudo-code program output?

```
int main()
{
   printf(“Hi\n”);
   fork();
   exec(....);
   printf(“Bye\n”);
}
```

a) Hi

b) Hi, Bye

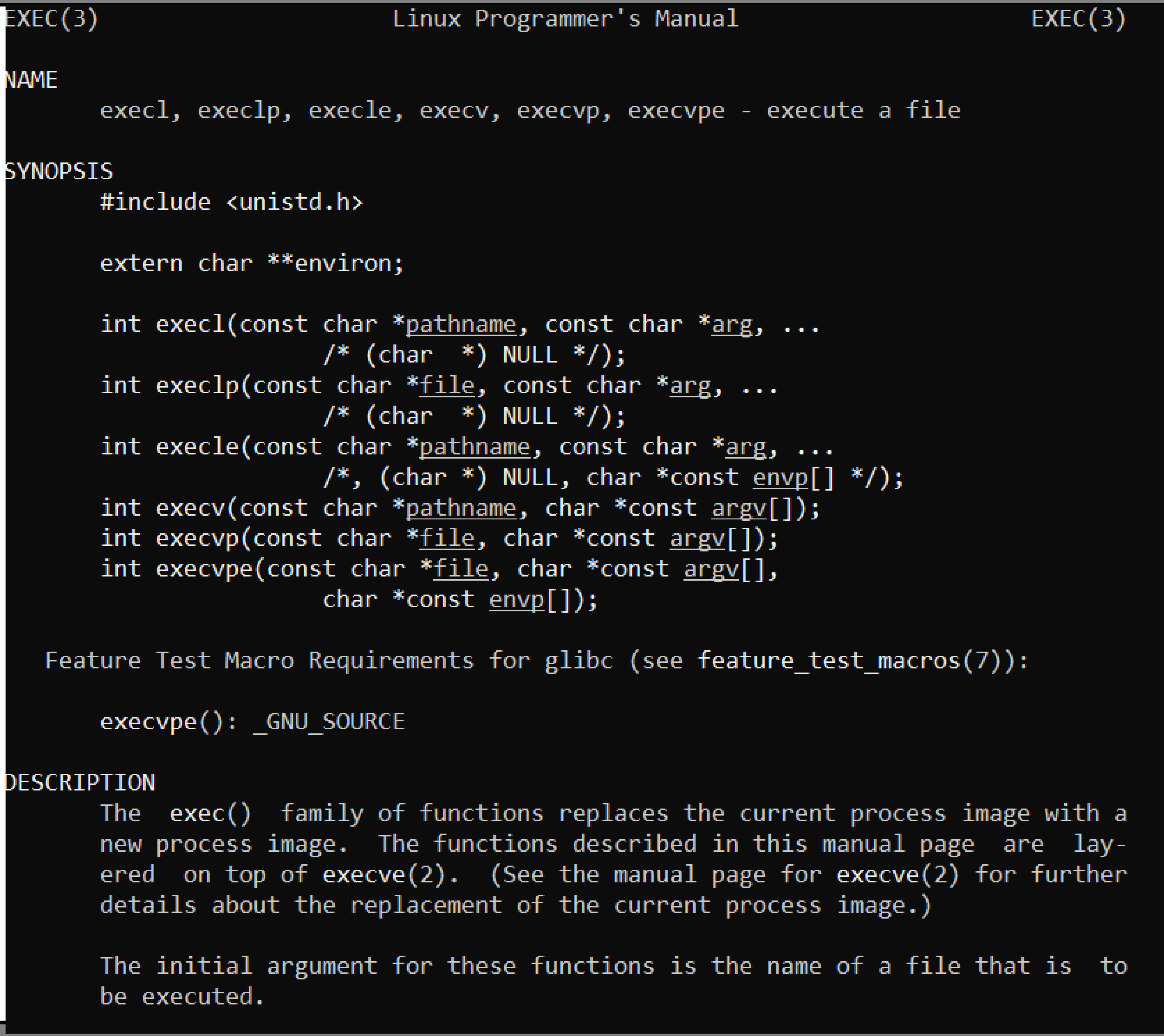
c) Hi, Bye, Bye,

d) Hi, Bye, Hi, Bye,

- *What happens to rest of a program after calling `exec( )`?*
  - It won't get executed; it's replaced in memory.
  - Analogy:
    - If a process is like a body, then `exec( )` is a brain transplant

man 3 exec

- *Many different exec() flavours*



```
EXEC(3)                    Linux Programmer's Manual                   EXEC(3)

NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl(const char *pathname, const char *arg, ...
                       /* (char  *) NULL */);
       int execlp(const char *file, const char *arg, ...
                       /* (char  *) NULL */);
       int execle(const char *pathname, const char *arg, ...
                       /*, (char *) NULL, char *const envp[] */);
       int execv(const char *pathname, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
       int execvpe(const char *file, char *const argv[],
                       char *const envp[]);

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

       execvpe(): _GNU_SOURCE

DESCRIPTION
       The  exec()  family of functions replaces the current process image with a
       new process image.  The functions described in this manual page  are  lay-
       ered  on top of execve(2).  (See the manual page for execve(2) for further
       details about the replacement of the current process image.)

       The initial argument for these functions is the name of a file that is  to
       be executed.
```

exec() Flavours

- `exec()` family has functions like:
  - exec**l**(...), exec**v**(...)
exec**l**p(...),exec**v**p(...)
exec**l**e(...),exec**v**pe(...)
- **l,v** *Distinguish how command line arguments are passed*
  - If it has an "l", pass each argument individually:

    ```
    execl("/bin/echo", "/bin/echo", "Yes!", "No!");
    ```

  - If it has a 'v', pass arguments in an array:

    ```
    char* args[] = {"/bin/echo", "hello", "world"};
    execv("/bin/echo", args);
    ```

- **p** *Use the search path to find the program*
  - With execlp() you can run "echo" and Linux will find it for you; with execl() you need to tell Linux where to find echo.
- **e** *Specify the environment variables as well*

Subtleties of Arguments

- *When a program is executed, OS hands it some command-line arguments*
    - args[0] (‘arg0’) is *the program’s name on disk – by convention!*
    - args[1] and beyond are the other arguments.
- `exec()` *calls take*
    - What program to execute
    - What arguments to pass the new process
- *When calling exec() functions, you specify the arguments*
    - We must make these arguments start with the program name
    - **So we end up listing it twice,** e.g. `execl("/bin/ls", "/bin/ls", "/home/", "-l", NULL);`

Activity - `exec()`

- *Write a program that* (15m)
  1) Creates a child process
  2) Parent: call any “exec” function that executes “ls -a”
  3) Child: call any “exec” function that executes “ls -a -l -h”
     - (same as “ls -alh” but spelled out)
- *Discussion*
  - At end of our program, if we add:
    `printf("%d\n", getpid())`
  - What will the parent print out?
  - What will the child print out?

Summary

- *Create a new process using fork()*
  - Clones current process.
  - **`fork()` returns twice:**
    - **Parent** knows it's the parent because
      return PID is non-zero (= the child's PID)
    - **Child** knows it's the child because
      return PID is zero
- *Replace a running program with exec()*
  - Pass in what program you want loaded
    into the current process.
  - **Completely replaces the process's memory space**