Forks, pipes and shared memory

Edward Zhang

SOFTENG 370 T2

Fork

Fork creates a new process, which becomes a child of the caller. Both processes then contiune execution from the line where fork was called.

1. Fork returns negative when forking failed
2. Fork returns zero within the new child process
3. Fork returns a positive value with the process id within the parent. This is a `pid_t`

Fork example

```c
int main(void) {
  printf("Hello from pid: %d\n", getpid());
  int fork1 = fork(); // actually a pid_t
  printf("Forked! I'm pid %d and fork returned %d\n", getpid
      (), fork1);
  int fork2 = fork(); // actually a pid_t
  printf("Forked again! I'm pid %d and fork returned %d\n",
      getpid(), fork2);
  return 0;
}
```

How many times are each line printed, and when is the return zero and non-zero?

Forked data

When forked, all pages allocated for a process are copied. This includes pages that store the stack, or memory on the heap (i.e. from malloc).
Copy-on-Write (CoW) is used so unless the child process modifies the data, a needless copy is not made.

A note on addresses

Consider this code. What's returned?

```
int main(void) {
    int *data = malloc(sizeof(int));
    *data = 9001;
    int fork1 = fork(); // actually a pid_t
    if (fork1 == 0) {
        *data = 9000;
    }
    printf("Forked! I'm pid %d and the data is at %p is %d\n
        ", getpid(), data, *data);
    return 0;
}
```

A note on addresses

Consider this code. What's returned?

```
int main(void) {
    int *data = malloc(sizeof(int));
    *data = 9001;
    int fork1 = fork(); // actually a pid_t
    if (fork1 == 0) {
        *data = 9000;
    }
    printf("Forked! I'm pid %d and the data is at %p is %d\n
        ", getpid(), data, *data);
    return 0;
}
```

How do we have different data at the same memory address?

A note on addresses

Consider this code. What's returned?

```
int main(void) {
    int *data = malloc(sizeof(int));
    *data = 9001;
    int fork1 = fork(); // actually a pid_t
    if (fork1 == 0) {
        *data = 9000;
    }
    printf("Forked! I'm pid %d and the data is at %p is %d\n
        ", getpid(), data, *data);
    return 0;
}
```

How do we have different data at the same memory address?
A: Virtual memory space is unchanged, even if a copy is made in physical memory.

Waiting for children

`waitpid` can be used to wait for children. Its use is quite simple.

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

- `pid` is the pid to wait on. Use -1 for any children, 0 for any child with same group ID or $x < -1$ for children in group $|x|$
- `status` is a pointer to an int to store the return status of the terminated process
- `options` options such as WNOHANG - see manpage
- Return value is the id of the terminated process (if you used pid < 1)

Pipes

Consider the case where we have two processes after a fork - how can we communicate between them? We can use a pipe.

`int pipe(int fds[2])`

- `fd[0]` will be the file descriptor for the read end
- `fd[1]` will be the file descriptor for the write end
- 0 returned on success, -1 on error (no other returns)

We should create this pipe before forking, such that after forking both processes have a reference to the pipe.

Pipe example

Sending strings through a pipe, from the child process to the parent process.

Other notes

- Pipes are unidirectional. If you need bidirectional communication use two pipes
- Remember to close the side of the pipe you're not using in a given process
- `read` will read 0 bytes when there are no more open write fds on the pipe
- While you can have more than 2 processes interact with a pipe, be careful about how you're using it as it may not work the way you intend.

mmap

Allows you to map files or devices into memory. However, this can also be used for creating a shared memory mapping that can be used by multiple processes.

You can find it in `#include <sys/mman.h>`

mmap parameters

```
void *mmap(void *addr, size_t length, int
    prot, int flags, int fd, off_t offset)
```

- `addr` is the address we want to start mapping. NULL to let kernel pick
- `length`: size of memory we want to map
- `prot`: options about page (more later)
- `flags`: more page options (more later)
- `fd`: file descriptor, if mapping a file
- `offset`: this plus size is used to compute where we're accessing, in the case of file mapping.

Using mmap for shared memory

```
int *shared = mmap(NULL, sizeof(int),
    PROT_READ|PROT_WRITE, MAP_SHARED|
    MAP_ANONYMOUS, -1, 0);
```

- `addr` is NULL because we don't care where to place the mapping (let kernel pick)
- `length` is the size of an int, as we want to store an int
- `prot` is read/write because... obvious
- `flags` is `SHARED` (we want shared memory) and `ANONYMOUS` (as opposed to file mapping)
- `fd` is -1 as we're not mapping a file
- `offset` is 0 as we're not mapping a file

mmap example code

```
int main(void) {
  int *shared = mmap(NULL, sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED|
       MAP_ANONYMOUS, -1, 0);
  int childPid;
  int childStatus;

  *shared = 0;

  if ((childPid = fork()) == 0) {
    // Child
    for (int i = 0; i < 10; i++) {
      printf("I'm pid %d and shared is now %d\n", getpid(), (*shared)++);
    }
  } else {
    // Parent
    for (int i = 0; i < 10; i++) {
      printf("I'm pid %d and shared is now %d\n", getpid(), (*shared)++);
    }
    waitpid(childPid, &childStatus, 0); // wait for child
  }
  return 0;
}
```