

# Section 2: Processes

August 29, 2015

## Contents

<b>1</b>	<b>Warmup</b>	<b>2</b>
1.1	Hello World . . . . .	2
<b>2</b>	<b>Vocabulary</b>	<b>2</b>
<b>3</b>	<b>Problems</b>	<b>3</b>
3.1	Forks . . . . .	3
3.2	Fork Bomb . . . . .	3
3.3	Stack Allocation . . . . .	4
3.4	Heap Allocation . . . . .	4
3.5	Simple Wait . . . . .	5
3.6	Complicated Wait . . . . .	6
3.7	Exec . . . . .	7
3.8	Exec + Fork . . . . .	7

# 1 Warmup

## 1.1 Hello World

What can C print in the below code? Assume the child's PID is 90210  
(Hint: There is more than one correct answer)

```
int main() {
    pid_t pid = fork();
    printf("Hello World: %d\n", pid);
}
```

```
Hello World 90210
Hello World 0
(either can come first)
```

# 2 Vocabulary

- **process** - a process is an instance of a computer program that is being executed. It contains the program code and its current activity.
- **exit code** - The exit status or return code of a process is a 1 byte number passed from a child process (or callee) to a parent process (or caller) when it has finished executing a specific procedure or delegated task
- **address space** - The address space for a process is the set of memory addresses that it can use. The address space for each process is private and cannot be accessed by other processes unless it is shared.
- **stack** - The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some book-keeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed.
- **heap** - The heap is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time.
- **fork** - A C function that calls the fork syscall that creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process (except for a few details, read more in the man page). Both the newly created process and the parent process return from the call to fork. On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

- **wait** - A class of C functions that call syscalls that are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.
- **exec** - The `exec()` family of functions replaces the current process image with a new process image. The initial argument for these functions is the name of a file that is to be executed.

### 3 Problems

#### 3.1 Forks

How many new processes are created in the below program?

```
int main(void)
{
    for (int i = 0; i < 3; i++) {
        pid_t pid = fork();
    }
}
```

7 (8 including the original process). Newly forked processes will continue to execute the loop from wherever their parent process left off.

#### 3.2 Fork Bomb

Why is the following code terrible for your system? (Don't try this at home)

```
int main(void)
{
    while(1)
        fork();
}
```

This program will constantly spin up new processes till your computer freezes up, each of the new processes will start spinning up new processes. So for every  $n$  runs of the while loop  $2^n$  processes will be created, and your computer will try to context switch between all of them.

#### 3.3 Stack Allocation

What can C print?

```
int main(void)
{
    int stuff = 7;
    pid_t pid = fork();
    printf("Stuff is %d", stuff);
    if (pid == 0)
        stuff = 6;
}
```

```
Stuff is 7
Stuff is 7.
(Since the entire address space is copied stuff will still remain the same)
```

### 3.4 Heap Allocation

What can C print?

```
int main(void)
{
    int* stuff = malloc(sizeof(int)*1);
    *stuff = 7;
    pid_t pid = fork();
    printf("Stuff is %d", *stuff);
    if (pid == 0)
        *stuff = 6
}
```

```
Stuff is 7
Stuff is 7.
(Since the entire address space is copied stuff will still remain the same)
```

### 3.5 Simple Wait

What can C print? Assume the child PID is 90210.

```
int main(void)
{
    pid_t pid = fork();
    int exit;
    if (pid != 0) {
        wait(&exit);
    }
    printf("Hello World\n: %d\n", pid);
}
```

```
Hello World 0
Hello World 90210
```

What is the equivalent program using the `waitpid` function instead of `wait`?

```
int main(void)
{
    pid_t pid = fork();
    int exit;
    if (pid != 0) {
        waitpid(-1, &exit, 0);
    }
    printf("Hello World\n: %d\n", pid);
}
```

### 3.6 Complicated Wait

What is the exit code of the following program? (Hint: Try to recognize a pattern)

```
int foo(n){
    if (n < 2) {
        exit(n);
    } else {
        int v1;
        int v2;
        pid_t pid = fork();
        if (pid == 0)
            foo(n - 1);
        pid_t pid2 = fork();
        if (pid2 == 0)
            foo(n - 2);
        waitpid(pid,&v1,0);
        waitpid(pid2,&v2,0);
        exit(WEXITSTATUS(v1) + WEXITSTATUS(v2));
    }
}

int main() {
    foo(10);
}
```

55 (This is the fibonacci sequence)

Why does this program start to behave (even more) strangely when we have  $n > 13$ ? What is the exit code for `foo(14)`? (You can use a calculator if you want)

121 (Exit codes are only 1 byte so only the last 8 bits of the 14th fibonacci number are used).

### 3.7 Exec

What can C print?

```
int main(void)
{
    char** argv = (char**) malloc(3*sizeof(char*));
    argv[0] = "/bin/ls";
    argv[1] = ".";
    argv[2] = NULL;
    for (int i = 0; i < 10; i++){
        printf("%d\n", i);
        if (i == 3)
            execv("/bin/ls", argv);
    }
}
```

```
0
1
2
3
<output of ls>
```

### 3.8 Exec + Fork

How would I modify the above program using fork so it both prints the output of `ls` and all the numbers from 0 to 9 (order does not matter)? You may not remove lines from the original program; only add statements (and use fork!).

```
int main(void)
{
    char** argv = (char**) malloc(3*sizeof(char*));
    argv[0] = "/bin/ls";
    argv[1] = ".";
    argv[2] = NULL;
    for (int i = 0; i < 10; i++){
        printf("%d\n", i);
        if (i == 3) {
            pid_t pid = fork();
            if (pid == 0)
                execv("/bin/ls", argv);
        }
    }
}
```