Tutorial 1: Processes

Vaishaal Shankar, Erik Krogen,
Dmitry Shkatov, Craig Bester

August 24, 2017

Contents

1 Vocabulary

- **process** - A process is an instance of a program that is being executed. It contains the program text as well as the information about the process's current activity. Another way of thinking of a process is as an address space plus a set of threads of execution.

- **address space** - The address space for a process is a set of memory addresses that it can use. The address space for each process is private: it cannot be accessed by other user-level processes unless it is explicitely shared. (The kernel, on the other hand, can access addresses with the address space of any process.)

- **exit code** - The exit status or return code of a process is a 1 byte-long integer passed from a child process (callee) to a parent process (caller) when it has finished executing a specific procedure or delegated task.

- **stack** - The stack is a region of 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 the arguments passed to the function, the local variables, and the address of the instruction to be executed on return from the call. When the function returns, the block becomes unused and can be used the next function call. 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 a region 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 to create a new process by duplicating the calling process. The new process, referred to as the child, is an almost exact duplicate of the calling process, referred to as the parent: the parent and the child have identical data, stack, and heap segments and share the text segment of their address space. A fork() call returns twice: once to the child and once to the parent. If the call has been successful, the PID of the child process is returned to the parent, and 0 is returned to the child. On failure, -1 is returned to the parent, and 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. The following state changes are recognised: the child terminated, the child was stopped by a signal, the child was resumed by a signal.

- **exec** - The exec family of functions replaces the current process image with a new process image. The first argument for all of these functions is the name of a file that to be executed.

2 Questions

2.1 Hello World

What is the output produced by the following code? Assume the child's PID is 90210.
If you're not sure, try executing it on your machine. (Hint: There is more than one correct answer.)

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

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

2.2 Forks

How many processes are created by the following code?
(Including the process created when you execute the program.)

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

2.3 Fork Bomb

Why is the following code bad for your system? (Don't try executing this code!)

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

2.4 Stack Allocation

What is the output of the following code?

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

2.5 Heap Allocation

What is the output of the following code?

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

2.6 Simple Wait

1. What is the output of the following code? Assume the child PID is 90210.
2. Rewrite the code using the waitpid function instead of wait.

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

2.7 Nontrivial Wait

1. What is the exit code of the following code? Try to recognise a pattern. (You can view the exit code of the last command in bash with "echo $?")
2. Why does this program start exhibiting odd behaviour when $n > 13$?
   (Hint: What is the datatype of the exit code?)
3. What is the exit code for foo(14)?

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

2.8 Exec

What is the output produced by the following code?

```
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);
    }
}
```

2.9 Exec + Fork

Modify the above code so it prints both the output of ls and all the numbers from 0 to 9 (in an arbitrary order). You may not remove any lines from the original program; you can only add statements. (Hint: use fork()).