Section 2: Processes

September 5-6, 2017

Contents

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)
or
Hello World -1
(if fork fails)
```

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 bookkeeping 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 assuming calls to fork succeeds?

```
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 Stack Allocation

What can C print?

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

3.3 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\n", *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 Slightly More Complex Heap Allocation

What does C print in this case ?

```
void printTenNumbers(int *arr)
{
    int i;
    printf("\n");
    for(i=0; i<10; i++) {
        printf("%d",arr[i]);
    }
    exit(0);
}

int main()
{
    int *arr, i;
    arr = (int *) malloc (sizeof(int));

    arr[0] = 0;
    for(i=1; i<10; i++) {
        arr = (int *) realloc( arr, (i+1) * sizeof(int));
        arr[i] = i;
        if (i == 7) {
            pid_t pid = fork();
            if (pid == 0) {
                printTenNumbers(arr);
            }
        }
    }
    printTenNumbers(arr);
}
```

01234567 ⟨ possible seg fault ⟩ for child
0123456789 for parent.
The array was not allocated to hold 10 integers when fork() was called.
Using fork() when multiple threads in the parent process
are operating critical sections or operating on globally accessible data
is dangerous for a similar reason ! Try to use fork() only when you want
to exec() !!

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 Fork and File Descriptors

What will be stored in the file "output.txt" ?

```
int main(void)
{
    int fd;

    fd = open("output.txt", O_CREAT|O_TRUNC|O_WRONLY, 0666);

    if(!fork()) {
        write(fd, "hello ", 6);
        exit(0);
    } else {
        int status;

        wait(&status);
        write(fd, "world\n", 6);
    }
}
```

"hello world"

File descriptors are copied and point to the same underlying file structure in the kernel's open file table.

3.7 Exec

What will 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);
        }
    }
}
```

3.9 Implementing fork() efficiently (Design)

Remember fork() makes the child process's address space exactly the same as its parent's. If you were designing an OS, list some steps you would take to make this address space copy more efficient ?

```
Copying the entire address space would take a long time.
So a more efficient method is copy-on-write.
Initially do not copy anything.
Write new pages and update address mapping as and when the child modifies its address space.
Read https://en.wikipedia.org/wiki/Copy-on-write for quick intuition.
Copy-On-Write will be introduced in more detail later in the course.
```