Arguments in Novix OS

Until now, our userland programs could be launched from the shell but had no way to receive parameters. That changes now.

In this chapter, we'll implement **command-line arguments** (`argc`, `argv`) — a foundational feature of any Unix-like OS.

Learning Goals

By the end of this lesson, you'll understand:

- How Novix passes arguments (`argv`, `argc`) from shell → kernel → user program
- How the kernel builds a **user stack** containing argument data
- How `exec()` copies and prepares arguments for the new process
- How to write programs that use `argc` and `argv`, like `echo`

Overview of Argument Passing

When a program calls `exec("echo.elf", argv, argc)`, the following sequence occurs:

1. The **shell** tokenizes the user's input into `argv[]`
2. The **kernel's sys_exec()** copies these strings from user space into a kernel buffer
3. The **setup_user_arguments()** function rebuilds them on the new user stack
4. Finally, the new process starts at `main(int argc, char** argv)`

1. Setting Up User Arguments (Kernel)

Defined in **`kernel/arguments.c`**, this is where the magic happens —
the kernel allocates stack space for arguments, copies each string, and builds the `argv[]` array for the new process.

`kernel/arguments.c`

```
char** setup_user_arguments(char argv_kernel[MAX_ARGS][MAX_ARG_LENGTH], int argc, uint64_t *sp_ptr) {
    uint64_t sp = *sp_ptr;
    char* argv_ptrs[MAX_ARGS + 1];  // temporary in kernel

    // Copy argument strings
    for (int i = argc - 1; i >= 0; i--) {
        size_t len = strnlen(argv_kernel[i], MAX_ARG_LENGTH) + 1;
        sp -= len;
        sp &= ~0x7UL;
        enable_sum();
        memcpy((void*)sp, argv_kernel[i], len);
        disable_sum();
        argv_ptrs[i] = (char*)sp;
    }

    argv_ptrs[argc] = NULL;

    // Copy argv[] pointers to user stack
    sp -= (argc + 1) * sizeof(char*);
```

```
    sp &= ~0x7UL;
    enable_sum();
    memcpy((void*)sp, argv_ptrs, (argc + 1) * sizeof(char*));
    disable_sum();

    *sp_ptr = sp;
    return (char**)sp;
}
```

Key concepts:

- Each argument string is **copied** to the new user stack
- Pointers to these strings are also written to the stack (`argv[]`)
- The function returns a **user-space pointer** to `argv`

This ensures that when user code runs `main(int argc, char** argv)`, the arguments are already ready and accessible!

2. Extending exec() for Arguments

The `exec()` syscall now accepts three parameters — filename, argv, and argc.

kernel/syscall.c

```
case SYS_EXEC:
    enable_sum();

    const char* filename = (const char*)f->regs.a0;
    char** argv = (char**)f->regs.a1;
    int argc = f->regs.a2;

    f->regs.a0 = sys_exec(filename, argv, argc);

    disable_sum();
    break;
```

Inside `sys_exec()`, we prepare the program image, set up the user stack, and call `setup_user_arguments()` if arguments exist.

sys_exec() (excerpt)

```
int sys_exec(const char *progname, char** argv, int argc) {
    char progname_buf[PROC_NAME_MAX_LEN];
    size_t fs;

    // Find ELF in TARFS
    const void *elf_data = tarfs_lookup(progname, &fs, 0);
    if (!elf_data) return -1;

    // Copy arguments
    char (*argv_buf)[MAX_ARG_LENGTH] = (char (*)[MAX_ARG_LENGTH])alloc_pages(1);
    for (int i = 0; i < argc; ++i) {
        strncpy(argv_buf[i], argv[i], MAX_ARG_LENGTH - 1);
        argv_buf[i][MAX_ARG_LENGTH - 1] = '\0';
    }
```

```
    // Load ELF and prepare new process
    struct process *ep = extract_flat_binary_from_elf(elf_data, EXEC_PROCESS);

    uint64_t sp = g_user_stack_top;
    char **argv_ptr = setup_user_arguments(argv_buf, argc, &sp);

    free_page((paddr_t)argv_buf);

    trap_frame_t *tf = ep->tf;
    memset(tf, 0, sizeof(*tf));
    tf->epc = USER_BASE;
    tf->regs.ra = (uint64_t)user_return;
    tf->sp = sp;

    proc->argc = argc;
    proc->argv_ptr = (uint64_t)argv_ptr;

    yield();
    return 0;
}
```

What's happening:

- Arguments are copied from user → kernel → new user stack
- The kernel sets up the new process's trapframe (`sp`, `epc`, etc.)
- `argc` and `argv_ptr` are stored in the process control block (PCB)
- `user_return()` restores CPU state and jumps into user mode

3. Jumping to User Mode

The function `user_return()` ensures that registers are correctly restored and that `argc` and `argv` are passed to the new program.

kernel/user.c

```
__attribute__((naked)) void user_return(void) {
    __asm__ __volatile__ (
        ...
        // Load argc (a0)
        "li   t3, %[argc_offset]\n"
        "add  t3, t2, t3\n"
        "ld   a0, 0(t3)\n"

        // Load argv (a1)
        "li   t3, %[argv_offset]\n"
        "add  t3, t2, t3\n"
        "ld   a1, 0(t3)\n"
        ...
        "sret\n"
    );
}
```

When control returns to user space, registers `a0` and `a1`

contain `argc` and `argv`, just like in C programs.

4. The User Program: `echo`

We'll create our first argument-aware program!

`user/echo.c`

```c
#include "include/stdio.h"
#include "include/syscall.h"

void main(int argc, char** argv) {
    for (int i = 1; i < argc; i++) {
        puts(argv[i]);
        if (i < argc - 1) putc(' ');
    }
    putc('\n');
    exit(0);
}
```

Example Output

```
$ echo Hello Novix OS
Hello Novix OS
```

Each argument is printed in sequence, separated by spaces.

5. Shell Integration

Our shell now supports arguments automatically,
thanks to proper tokenization and passing through `exec()`.

`user/shell.c` (excerpt)

```c
int pid = fork();
if (pid == 0) {
    exec(filename, argv, argc);
    puts("[shell] exec failed\n");
    exit(1);
} else if (pid > 0) {
    int status;
    wait(&status);
}
```

The shell tokenizes user input with `strtok()`, builds `argv[]`,
and sends both to the kernel through `exec()`.

Example Session

```
Novix RISC-V 64 OS, (c) NovixManiac, Shell version : 0.0.1
$ echo Hello from Novix!
Hello from Novix!

$ echo This OS rocks
This OS rocks
```

- The shell tokenizes user input

- `exec()` passes argv/argc to kernel
- Kernel rebuilds the user stack and starts program
- `echo` prints the arguments — just like Unix

Design Reflection

Concept	Purpose
`setup_user_arguments()`	Copies and aligns arguments on user stack
`sys_exec()`	Loads ELF and prepares argv/argc for new process
`user_return()`	Transfers control to user mode with correct args
`echo.elf`	First user program with argument support
Shell	Tokenizes input and passes argv/argc

Summary

In this chapter, we've achieved:

- Full `argc` / `argv` argument support in Novix OS
- Extended `exec()` to pass and copy arguments safely
- Added stack alignment and protection
- Created the first argument-based program (`echo`)
- Enabled the shell to handle complex commands

Novix OS can now execute user programs **with parameters** —
a huge milestone that makes the shell *truly interactive*.

A smile on my face — because Novix OS shell support now parameters!