Building Shell Commands `ls` and `ps` in Novix OS

We now have a working shell that can start, run, and manage programs interactively.
But what's a shell without the ability to see **what's running** or **what files exist**?

In this chapter, we'll implement two fundamental utilities: - `ps` — show the process list
- `ls` — list files in the TARFS filesystem

These are our first real *userland tools*, each running as an independent process, using **system calls** to ask the kernel for information.

Learning Goals

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

- How to create custom shell commands as standalone ELF programs
- How user programs can access kernel information using syscalls
- How the shell integrates built-in and external commands (`ps`, `ls`)
- How to extend Novix with more utilities in the future

Recap: The Shell Structure

In the previous lesson, we built a minimal shell that could:

- Read user input
- Parse commands
- Launch ELF binaries using `fork()` and `exec()`
- Wait for them to finish using `wait()`

Now, we'll expand that design by adding **new syscalls** and **userland binaries** for system introspection.

Adding Syscalls for ps and ls

Let's start with the kernel side.

`kernel/syscall.c`

We define two new syscall numbers — `SYS_PS` and `SYS_LS` — and implement their handlers.

```
case SYS_PS:
    sys_ps();
    f->regs.a0 = 0;
    break;

case SYS_LS:
    sys_ls();
    f->regs.a0 = 0;
    break;
```

Each of these calls a helper function (`sys_ps()` or `sys_ls()`) that performs the actual work.

Implementing sys_ps()

`ps` (process status) prints the process table with IDs, states, and names.

```c
void sys_ps(void) {
    uart_puts("PID\tSTATE\t\tNAME\n");
    for (int i = 0; i < PROCS_MAX; i++) {
        proc_t *p = &procs[i];
        if (p->pid != 0) {
            uart_printf("%d\t%s\t%s\n", p->pid, proc_state_str(p->state), p->name);
        }
    }
}
```

What happens:

- The kernel iterates through the global `procs[]` array.
- For each valid process (`pid != 0`), it prints:
  - The PID
  - The current state (`RUNNING`, `SLEEPING`, `ZOMBIE`, etc.)
  - The process name

Output appears directly via `uart_printf()`, visible in the user's terminal.

Implementing sys_ls()

`ls` (list storage) scans the **TARFS filesystem** in memory and lists all contained files.

```c
void sys_ls(void) {
    uint8_t *ptr = _binary_initramfs_tar_start;

    uart_puts("Files in tarfs:\n");

    while (ptr < _binary_initramfs_tar_end) {
        struct tar_header *hdr = (struct tar_header *)ptr;

        if (hdr->name[0] == '\0') break;  // End of tar

        uart_printf("  %s\n", hdr->name);

        // Calculate file size from octal field
        size_t size = 0;
        for (int i = 0; i < 11; ++i) {
            size = (size << 3) + (hdr->size[i] - '0');
        }

        size_t blocks = (size + TAR_BLOCK_SIZE - 1) / TAR_BLOCK_SIZE;
        ptr += (1 + blocks) * TAR_BLOCK_SIZE;
    }
}
```

Key insights:

- `_binary_initramfs_tar_start` and `_binary_initramfs_tar_end` mark the TARFS archive in memory.
- Each file header contains a name, size, and other metadata.
- We iterate over each header, print its name, and skip the data section (rounded to 512 bytes).

This is a **minimal but effective filesystem listing**.

Userland Wrappers

User programs don't call `sys_ps()` or `sys_ls()` directly — they use **syscall wrappers**.

`user/lib/syscall.c`

```
void ps() {
    syscall(SYS_PS, 0, 0, 0);
}

void ls() {
    syscall(SYS_LS, 0, 0, 0);
}
```

These provide a simple C interface so user programs can just call `ps()` or `ls()`.

Creating the ps and ls Programs

Let's define two standalone ELF binaries.

`user/ps.c`

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

void main() {
    ps();
    exit(0);
}
```

`user/ls.c`

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

void main() {
    ls();
    exit(0);
}
```

Both simply call their respective syscall and exit immediately.

Each program runs **entirely in user space**, but requests privileged information from the kernel through `syscall()`.

Integrating into the Shell

Finally, let's update the shell to recognize and describe these new commands.

`user/shell.c` (excerpt)

```
void print_help(void) {
    puts("Available commands:\n");
    puts("  help         Show this overview\n");
    puts("  hello        Shows the hello message\n");
    puts("  ps           Show list of active processes\n");
    puts("  ls           Show files in the current directory\n");
```

```
    puts("  clear        Clear the screen\n");
}
```

Since both `ps` and `ls` are external ELF programs, no special case is needed —
the shell will simply find them in TARFS and launch them like any other command.

Example Session

```
Novix RISC-V 64 OS, (c) NovixManiac, Shell version : 0.0.1
$ ls
Files in tarfs:
  hello.elf
  ps.elf
  ls.elf
  shell.elf
  idle.elf

$ ps
PID    STATE           NAME
0      RUNNING         idle
1      RUNNING         shell

$ hello
[hello] Hello from exec!
```

- `ls` lists all ELF files in memory.
- `ps` shows all active processes, including the shell and idle tasks.
- Commands execute as separate processes — launched, executed, and cleaned up dynamically.

Design Reflection

Concept	Purpose
`sys_ps()`	Kernel prints process table
`sys_ls()`	Kernel lists TARFS files
`ps.elf` / `ls.elf`	User programs calling kernel syscalls
Shell	Provides user interface for launching programs
TARFS	Embedded filesystem storing executables

This demonstrates **true multitasking** and **process isolation** — the userland tools communicate with the kernel but never access privileged memory directly.

Summary

In this chapter, we've achieved:

- Added `ps` and `ls` system calls in the kernel
- Built matching userland wrappers (`ps()`, `ls()`)
- Created standalone ELF programs
- Integrated them seamlessly into the shell

Novix OS now feels *alive*: you can inspect running processes and list available programs — just like in Unix.

```
*A smile on my face - because Novix OS shell feels alive!*
```