PennOS Shell - Interview Key Points

Overview

The PennOS Shell is a command-line interface that allows users to interact with the kernel through various built-in commands. It demonstrates process management, file I/O redirection, job control, and signal handling.

1. Shell Architecture

Command Execution Flow

```
User Input → Parser → Command Dispatcher → Process Spawning → Execution
```

1. **Input Parsing**: Shell reads user commands and parses them into tokens
2. **Command Dispatch**: Identifies command type and routes to appropriate handler
3. **Process Creation**: Spawns new process via `s_spawn()` with optional I/O redirection
4. **Execution**: Process runs user function and exits

Key Components

- **Parser** (`util/parser.c`): Tokenizes input commands
- **Job Manager** (`util/job.c`): Tracks background/foreground jobs
- **User Functions** (`user_function.c`): Implements shell built-in commands
- **Syscall Interface** (`syscall.c`): Provides process and file system APIs

2. I/O Redirection

Problem: File Descriptor Management

Issue: When redirecting stdout to a file (e.g., `echo 1 > test`), the file remains open after the process exits, preventing subsequent processes from opening the same file.

Root Cause: File descriptors were not being closed when processes terminated.

Solution: Modified `k_terminate()` in `process.c` to close all open file descriptors (fd 3+) before marking a process as zombie:

```
// Close all open file descriptors
for (int i = 3; i < MAX_FD; i++) {
  if (proc->fd_table[i] >= 0) {
    s_close(i);
  }
}
```

Redirection Mechanism

- **Stdout Redirection**: `command > file` (truncate) or `command >> file` (append)
- **Stdin Redirection**: `command < file` (read from file)
- **Implementation**: `s_spawn()` accepts optional `stdin_file` and `stdout_file` parameters
- **Wrapper Function**: Handles file opening/closing before executing user function

File Descriptor Table

Each process maintains a local file descriptor table (`fd_table[MAX_FD]`):

- **FD 0**: STDIN (standard input)
- **FD 1**: STDOUT (standard output)
- **FD 2**: STDERR (standard error)
- **FD 3+**: User-opened files

---

3. Process Management Commands

Process Spawning

`s_spawn(func, argv, stdin_file, stdout_file, is_append)`

- Creates new child process with optional I/O redirection
- Child inherits parent's file descriptors
- Default priority: 1 (normal)
- Returns child PID on success

Process Listing

`ps` command (`u_ps()`)

- Lists all processes with: PID, PPID, Priority, State, Command name
- Process states: R (Ready/Running), B (Blocked), S (Stopped), Z (Zombie)
- Displays command name stored in PCB

Process Termination

`kill` command (`u_kill()`)

- Sends signals to processes: `-term` (terminate), `-stop` (suspend), `-cont` (resume)
- Supports multiple target PIDs
- Uses `s_kill()` syscall

Priority Management

`nice` and `nice_pid` commands (`u_nice()`, `u_nice_pid()`)

- Changes process priority (0=high, 1=normal, 2=low)
- `nice`: Spawns new process with specified priority

- `nice_pid`: Changes priority of existing process
- Uses `s_nice()` syscall

---

4. Job Control

Job States

- **Running**: Process executing in foreground
- **Stopped**: Process suspended (via SIGSTOP)
- **Background**: Process running in background
- **Done**: Process completed

Job Control Commands

`bg` - Background Execution

- Resumes stopped job in background
- Sends SIGCONT signal to process
- Updates job state to JOB_BACKGROUND
- Usage: `bg [job_id]` or `bg` (most recent stopped job)

`fg` - Foreground Execution

- Brings background/stopped job to foreground
- Blocks shell until job completes or stops
- Sends SIGCONT if job was stopped
- Handles job state transitions based on exit status
- Usage: `fg [job_id]` or `fg` (most recent job)

`jobs` - List Jobs

- Displays all active jobs with: Job ID, PID, State, Command
- Shows Running, Stopped, Background, or Done status

Job Table Structure

```c
typedef struct {
  int job_id;           // Job identifier
  pid_t pid;            // Process ID
  char cmd[256];        // Command string
  job_state_t state;    // Current state
  pcb_t* pcb;           // Pointer to process control block
  bool used;            // Whether slot is in use
} job_t;
```

---

5. File System Commands

File Operations

- `cat`: Display file contents (supports multiple files)
- `ls`: List directory contents with permissions, size, mtime
- `touch`: Create empty files or update timestamps
- `mv`: Rename/move files
- `cp`: Copy files (supports host ↔ PennFAT transfers)
- `rm`: Delete files
- `chmod`: Change file permissions (rwx encoding)

Implementation Details

- All file commands use kernel syscalls (`s_open`, `s_read`, `s_write`, `s_close`)
- Support proper error handling with `u_perror()`
- Respect file permissions and filesystem constraints

---

6. Process State Testing

Zombie Process Testing

`zombify` command (`u_zombify()`)

- Spawns child process that immediately exits
- Parent continues running (infinite loop)
- Child becomes zombie, waiting for parent to reap it
- Demonstrates zombie state in `ps` output

Orphan Process Testing

`orphanify` command (`u_orphanify()`)

- Spawns child process that runs indefinitely
- Parent exits immediately
- Child becomes orphan, adopted by init process
- Demonstrates orphan adoption mechanism

---

7. Shell Built-in Commands

Information Commands

- `man`: Display help text for all shell commands
- `echo`: Print text to stdout
- `sleep`: Sleep for specified seconds
- `busy`: Busy-wait loop (CPU load testing)

System Commands

- `logout`: Gracefully shutdown PennOS

    - Calls `s_shutdown()` to set shutdown flag
    - Init process detects flag and terminates all processes

---

8. Error Handling

Error Codes

PennOS uses `P_ERRNO` global variable (similar to Unix `errno`):

- `P_ENOENT`: File not found
- `P_EBUSY`: File in use (cannot open for write)
- `P_ENOSPC`: Disk full
- `P_EACCES`: Permission denied
- `P_EBADF`: Invalid file descriptor

Error Reporting

- `u_perror(const char* msg)`: Prints formatted error message to stderr
- System calls return -1 on error and set `P_ERRNO`
- User programs check return value and inspect `P_ERRNO`

---

9. Signal Handling

Signals in PennOS

- **P_SIGTERM** (0): Terminate process → P_ZOMBIE state
- **P_SIGSTOP** (1): Stop process → P_STOPPED state
- **P_SIGCONT** (2): Continue process → P_READY state
- **P_SIGCHLD**: Child state change notification (internal)

Signal Delivery

- Delivered via `s_kill(pid, signal)` syscall
- Kernel function `k_signal_deliver()` handles state transitions
- Signals take effect immediately (no queuing/masking)

---

10. Key Implementation Details

Process Cleanup

When a process terminates via `k_terminate()`:

1. Close all open file descriptors (fd 3+)
2. Remove from scheduler queues
3. Mark as zombie
4. Adopt orphaned children to init
5. Unblock waiting parent

File Descriptor Inheritance

- Child processes inherit parent's file descriptor table
- Allows shell to redirect I/O before spawning command
- File descriptors are shared (reference counted)

Wrapper Function Pattern

For I/O redirection:

1. Create wrapper arguments structure
2. Spawn process with wrapper function
3. Wrapper opens redirection files
4. Wrapper calls actual user function
5. Wrapper closes redirection files on exit

---

11. Interview Questions & Answers

Q1: How does the shell handle I/O redirection?

A: The shell uses `s_spawn()` with optional `stdin_file` and `stdout_file` parameters. A wrapper function opens these files before executing the user command, then closes them after execution. File descriptors are properly closed when the process terminates via `k_terminate()`.

Q2: What was the "file is in use" bug and how was it fixed?

A: When a process redirected stdout to a file and exited, the file descriptor wasn't closed, preventing subsequent processes from opening the same file. Fixed by adding file descriptor cleanup in `k_terminate()` to close all open fds (3+) before marking process as zombie.

Q3: How does job control work?

A: The shell maintains a job table tracking background/foreground jobs. `bg` resumes stopped jobs in background, `fg` brings jobs to foreground and blocks shell until completion, `jobs` lists all active jobs. Job state transitions are managed based on signals and process state changes.

Q4: How are zombie processes handled?

A: When a process terminates, it becomes a zombie. Parent must call `s_waitpid()` to reap it. If parent exits before reaping, the zombie is adopted by init process, which reaps it. Zombies are displayed in `ps` output with state 'Z'.

Q5: What happens when a process exits?

A: `s_exit()` sets exit status to P_EXIT_EXITED and calls `k_terminate()`, which:

1. Closes all open file descriptors
2. Removes process from queues
3. Marks as zombie
4. Adopts orphaned children

5. Unblocks waiting parent
6. Process remains zombie until reaped

---

12. Testing Scenarios

Test 1: Basic Redirection

```
$ echo "hello" > test.txt
$ cat test.txt
hello
$ echo "world" > test.txt  # Should work (file closed after first echo)
$ cat test.txt
world
```

Test 2: Job Control

```
$ sleep 100 &
[1] 1234
$ jobs
[1] 1234 Running sleep 100
$ kill -stop 1234
$ jobs
[1] 1234 Stopped sleep 100
$ bg 1
[1] 1234 sleep 100
$ fg 1
sleep 100
^C
```

Test 3: Process Listing

```
$ ps
     PID    PPID PRI STAT   CMD
       1       0   1 R      init
       2       1   1 R      shell
       3       2   1 R      ps
```

Test 4: Zombie Process

```
$ zombify &
[1] 1234
$ ps
     PID    PPID PRI STAT   CMD
       1       0   1 R      init
```

```
         2       1   1 R      shell
         3       2   1 Z      zombie_child
```

Summary

The PennOS Shell demonstrates:

- **Process Management**: Spawning, termination, priority scheduling
- **I/O Redirection**: File descriptor manipulation and proper cleanup
- **Job Control**: Background/foreground execution, signal handling
- **File System Integration**: File operations with proper error handling
- **Resource Management**: File descriptor cleanup, orphan adoption, zombie reaping

Key insight: Proper resource cleanup (especially file descriptors) is critical for system stability and preventing resource exhaustion.