Codes:

🌑 All codes:

Table of Contents

Header Inclusions

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "traps.h"
#include "spinlock.h"
#include "sleeplock.h"
```

```
#include "fs.h"
#include "file.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
```

These are the necessary header files that provide type definitions, constants, function prototypes, and macros needed for the console implementation in an operating system context.

- **types.h**: Defines basic data types like `uint`, `uchar`, etc.
- **defs.h**: Contains function prototypes for kernel functions.
- **param.h**: Contains system parameters like `NPROC`, `NOFILE`, etc.
- **traps.h**, **spinlock.h**, **sleeplock.h**: For handling traps (interrupts), spinlocks, and sleep locks.
- **fs.h**, **file.h**: Filesystem and file-related structures and functions.
- **memlayout.h**, **mmu.h**: Memory management unit and memory layout definitions.
- **proc.h**: Process control structures and function prototypes.
- **x86.h**: x86-specific definitions and functions.

Global Variables and Structures

Console Lock State

```
static struct {
    struct spinlock lock;
    int locking;
} console_lock_state;
```

This structure maintains the state of the console lock:

- **lock**: A spinlock to ensure mutual exclusion when accessing console resources.
- **locking**: A flag indicating whether locking is enabled.

Copy Buffer

```
#define COPY_BUF_SIZE 256

struct {
    char buffer[COPY_BUF_SIZE]; // Buffer to store copied cha
racters
    int is_copying;             // Flag to indicate if copyin
g is active
    int index;                  // Index in the copy buffer
} copy_buffer;
```

This structure handles the copy-paste functionality:

- **buffer**: Stores the copied characters.
- **is_copying**: Indicates whether the copy mode is active.
- **index**: Points to the next position in the buffer to store a copied character.

Input Buffer

```
#define INPUT_BUFFER_SIZE 128

struct {
    char buffer[INPUT_BUFFER_SIZE];
    uint read_index;   // Read index
    uint write_index;  // Write index
    uint edit_index;   // Edit index
    uint cursor_shift; // Number of positions the cursor has
been shifted to the left (>= 0)
} input_buffer;
```

This structure manages the input from the user:

- **buffer**: Stores the characters typed by the user.
- **read_index**: Points to the position from where data is read.
- **write_index**: Points to the position where new data is written.
- **edit_index**: Points to the current editing position.
- **cursor_shift**: Tracks the cursor's shift to handle left and right arrow movements.

Command History

```
#define COMMAND_HISTORY_SIZE 10

struct {
    char buffer[COMMAND_HISTORY_SIZE][INPUT_BUFFER_SIZE]; // Buffer to store commands
    int read_index;                   // Read index (range [1, COMMAND_HISTORY_SIZE])
    int write_index;                  // Write index
    int in_tab_mode;                  // Whether we are in tab completion mode
    char temp_command[INPUT_BUFFER_SIZE]; // Temporary command
    int last_used_index;              // Index of last used command
} command_history;
```

This structure handles command history and prediction:

- **buffer**: Stores the history of commands entered by the user.
- **read_index**: Used when navigating through the history (e.g., with arrow keys).
- **write_index**: Points to the next position to store a new command.
- **in_tab_mode**: Indicates whether tab completion mode is active.

- **temp_command**: Temporarily stores the current command when navigating history.
- **last_used_index**: Index of the last command used for prediction.

Console Output Functions

`print_integer()`

```
static void print_integer(int value, int base, int is_signed)
{
    // Function implementation...
}
```

This function prints an integer `value` to the console in the specified `base` (e.g., decimal or hexadecimal):

- **value**: The integer to print.
- **base**: The numerical base (10 for decimal, 16 for hexadecimal).
- **is_signed**: Indicates whether the number is signed (1) or unsigned (0).

It handles negative numbers, converts the integer to the specified base, and outputs each digit using `console_output_char()`.

`cprintf()`

```
void cprintf(char* fmt, ...) {
    // Function implementation...
}
```

A formatted console print function, similar to `printf()` in C. It supports:

- **%d**: Decimal integer.
- **%x**, **%p**: Hexadecimal integer.
- **%s**: String.
- **%%**: Literal percent sign.

It uses variable arguments (`...`) to process the format string and output the corresponding values.

`panic()`

```
void panic(char* s) {
    // Function implementation...
}
```

This function is called when a critical error occurs:

- Disables interrupts using `cli()`.
- Outputs a panic message along with the current CPU ID (`lapicid()`).
- Retrieves and prints the call stack for debugging purposes.
- Sets `panicked` to 1 to prevent further processing.
- Enters an infinite loop to halt the system.

CGA Display Functions

These functions interact with the CGA (Color Graphics Adapter) display, which is the console output in text mode.

`get_cursor_position()`

```
static int get_cursor_position(void) {
    // Function implementation...
}
```

Retrieves the current cursor position on the screen by reading from the CRT controller registers (`CRTPORT`).

`set_cursor_position()`

```
static void set_cursor_position(int pos) {
    // Function implementation...
}
```

Sets the cursor position to `pos` by writing to the CRT controller registers.

`console_erase_character()`

```
static void console_erase_character(int pos) {
    crt[pos] = ' ' | 0x0700;
}
```

Erases the character at position `pos` by writing a space character with the default attribute (black background, white foreground).

`console_write_character()`

```
static void console_write_character(int pos, int c) {
    crt[pos] = (c & 0xff) | 0x0700;
}
```

Writes character `c` at position `pos` on the screen.

`cga_put_character()`

```
static void cga_put_character(int c) {
    // Function implementation...
}
```

Outputs character `c` to the CGA display:

- Handles newline (`\\n`) by moving the cursor to the next line.
- Handles backspace by moving the cursor back.
- Handles scrolling when the cursor reaches the bottom of the screen.
- Updates the cursor position.

`console_output_char()`

```
void console_output_char(int c) {
    // Function implementation...
}
```

Outputs character `c` to both the UART (serial port) and the CGA display:

- If the system is panicked, enters an infinite loop.
- For backspace, sends appropriate control characters to the UART.
- Calls `cga_put_character()` to display the character on the screen.

Input Handling Functions

Cursor Movement Functions

These functions manage cursor movement on the console.

`move_cursor_to_end()`

```
static void move_cursor_to_end(void) {
    set_cursor_position(get_cursor_position() + input_buffer.
cursor_shift);
}
```

Moves the cursor to the end of the input buffer, accounting for any shifts due to left/right arrow keys.

`move_cursor_left()`

```
static void move_cursor_left(void) {
    set_cursor_position(get_cursor_position() - 1);
}
```

Moves the cursor one position to the left.

`move_cursor_right()`

```
static void move_cursor_right(void) {
    set_cursor_position(get_cursor_position() + 1);
}
```

Moves the cursor one position to the right.

`move_cursor_to_start()`

```
static void move_cursor_to_start(void) {
    input_buffer.cursor_shift = input_buffer.edit_index - inp
ut_buffer.write_index;
    set_cursor_position(get_cursor_position() - input_buffer.
cursor_shift);
}
```

Moves the cursor to the start of the current input line.

Line Editing Functions

`console_erase_line()`

```
static void console_erase_line(void) {
    // Function implementation...
}
```

Erases the current input line:

- Moves the cursor to the end of the line.
- Deletes characters by moving back and overwriting with spaces.
- Resets the edit index.

`console_clear_screen()`

```
static void console_clear_screen(void) {
    int pos = get_cursor_position();
    while (pos >= 0)
```

```
        console_erase_character(pos--);
}
```

Clears the entire console screen by erasing all characters.

`console_new_command_prompt()`

```
static void console_new_command_prompt(void) {
    console_write_character(0, '$');
    set_cursor_position(2);
}
```

Displays a new command prompt (e.g., `$`) at the start of a new line.

Input Buffer Shifting Functions

These functions handle inserting and deleting characters within the input buffer, especially when the cursor is not at the end of the line.

`input_shift_left()`

```
static void input_shift_left(void) {
    // Function implementation...
}
```

Shifts the input buffer to the left when a character is deleted (e.g., backspace) and the cursor is in the middle of the line.

`input_shift_right()`

```
static void input_shift_right(void) {
    // Function implementation...
}
```

Shifts the input buffer to the right to make space for a new character when inserting in the middle of the line.

`console_shift_left()`

```
static void console_shift_left(void) {
    // Function implementation...
}
```

Updates the console display after the input buffer has been shifted left:

- Moves the cursor to the end.
- Deletes characters and re-displays the updated line.

`console_shift_right()`

```
static void console_shift_right(void) {
    // Function implementation...
}
```

Updates the console display after the input buffer has been shifted right:

- Moves the cursor to the end.
- Re-displays the line with the new character inserted.

`input_put_character()`

```
static void input_put_character(char c) {
    // Function implementation...
}
```

Handles inserting a character `c` into the input buffer:

- If the cursor is at the end, simply appends the character.
- If the cursor is in the middle, shifts the buffer to the right and inserts the character.
- Updates the console display accordingly.

Command History Functions

`store_command()`

```
static void store_command(void) {
    // Function implementation...
}
```

Stores the current command into the command history buffer:

- Shifts existing commands down to make room for the new command at the top.
- Copies the command from the input buffer to the history buffer.
- Updates the write index.

`load_command()`

```
static void load_command(void) {
    // Function implementation...
}
```

Loads a command from the history buffer into the input buffer:

- Erases the current line.
- Copies the command from the history buffer to the input buffer.
- Displays the command on the console.

`copy_current_command()`

```
static void copy_current_command(void) {
    // Function implementation...
}
```

Copies the current command from the input buffer to a temporary storage:

- Used when navigating command history to restore the current command if needed.

`recover_command()`

```
static void recover_command(void) {
    // Function implementation...
}
```

Restores the command from temporary storage back into the input buffer:

- Used when navigating back to the current command after viewing previous commands.

Command Prediction Functions

`is_prefix()`

```
static int is_prefix(const char* cmd, const char* input, int
input_size) {
    // Function implementation...
}
```

Checks if `input` is a prefix of `cmd`:

- Returns 1 if it is a prefix, 0 otherwise.

`get_predicted_command_index()`

```
static int get_predicted_command_index(const char* cmd, uint
cmd_size, int last_used_index) {
    // Function implementation...
}
```

Searches the command history for a command that starts with `cmd`:

- Starts searching from `last_used_index` to allow cycling through possible completions.
- Returns the index of the matching command or -1 if none found.

`predict_command()`

```
static void predict_command(void) {
    // Function implementation...
}
```

Handles command prediction when the user presses the Tab key:

- If not in tab mode, starts prediction from the beginning.
- If already in tab mode, continues searching for the next matching command.
- Displays the predicted command on the console.

`reset_command_history()`

```
static void reset_command_history(void) {
    // Function implementation...
}
```

Resets command history state after a command has been executed:

- Exits tab mode.
- Stores the executed command into history.
- Resets indices.

Expression Evaluation Functions

`evaluate_expression_from_end()`

```
static int evaluate_expression_from_end(int end_idx, int* res
ult) {
    // Function implementation...
}
```

Evaluates a simple arithmetic expression at the end of the input buffer:

- Supports addition (+), subtraction (-), multiplication (*), and division (/).
- Extracts numbers and operator by parsing backward from `end_idx`.

- Stores the result in `result`.
- Returns the length of the expression evaluated.

`process_input_buffer()`

```
static void process_input_buffer(void) {
    // Function implementation...
}
```

Processes the input buffer to replace patterns of the form `N O N=?` with the computed result:

- Searches for `=?` indicating an expression to evaluate.
- Calls `evaluate_expression_from_end()` to compute the result.
- Replaces the expression in the input buffer with the result.
- Adjusts the buffer and cursor positions accordingly.

Console Interrupt Handler

`consoleintr()`

```
void consoleintr(int (*getc)(void)) {
    // Function implementation...
}
```

Handles console interrupts, typically triggered by keyboard input:

- Acquires the console lock to ensure mutual exclusion.
- Reads characters using `getc()` and processes them based on their value.
- Handles special control characters and arrow keys for editing:
    - **CTRL-P** (`^P`): Triggers process listing.
    - **CTRL-U** (`^U`): Erases the current line.
    - **Backspace**: Deletes the character before the cursor.

    - **CTRL-L** (`^L`): Clears the screen.
    - **Arrow keys**: Moves the cursor left/right or navigates command history.
    - **Tab**: Triggers command prediction.
    - **CTRL-S** (`^S`): Starts copying mode.
    - **CTRL-F** (`^F`): Ends copying mode and pastes the copied text.
    - **CTRL-N** (`^N`): Deletes numbers from the input.
- Handles normal character input:
    - Inserts the character into the input buffer.
    - If Enter (`\\n`) is pressed, processes the input buffer (e.g., evaluates expressions).
    - Checks for special commands like `history` to display command history.
- Releases the console lock after processing.

Console Read and Write Functions

`consoleread()`

```
int consoleread(struct inode* ip, char* dst, int n) {
    // Function implementation...
}
```

Reads input from the console:

- Waits for input to be available in the input buffer.
- Copies characters from the input buffer to `dst` .
- Stops reading when:
    - End-of-file character (`^D`) is encountered.
    - The requested number of bytes (`n`) has been read.
    - A newline character (`\\n`) is encountered.

- Returns the number of bytes read.

`consolewrite()`

```
int consolewrite(struct inode* ip, char* buf, int n) {
    // Function implementation...
}
```

Writes output to the console:

- Writes `n` bytes from `buf` to the console by calling `console_output_char()`.
- Returns the number of bytes written.

Console Initialization

`consoleinit()`

```
void consoleinit(void) {
    initlock(&console_lock_state.lock, "console");

    devsw[CONSOLE].write = consolewrite;
    devsw[CONSOLE].read = consoleread;
    console_lock_state.locking = 1;

    ioapicenable(IRQ_KBD, 0);
}
```

Initializes the console subsystem:

- Initializes the console lock.
- Sets the console's read and write functions in the device switch table (`devsw`).
- Enables locking for console operations.
- Enables keyboard interrupts (`IRQ_KBD`) to allow console input.

Conclusion

The refactored console code provides a comprehensive set of features for an operating system's console, including:

- **Input Handling**: Captures and processes user input, supports line editing, and cursor movement.
- **Output Handling**: Outputs characters to both the UART (serial port) and the CGA display.
- **Command History**: Stores and retrieves previous commands, navigable via arrow keys.
- **Command Prediction**: Predicts commands based on input, accessible via the Tab key.
- **Copy-Paste Functionality**: Allows copying and pasting text within the console.
- **Expression Evaluation**: Evaluates simple arithmetic expressions entered by the user.
- **Special Commands**: Supports special commands like `history` to display command history.