```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>

#define BLUE "\033[0;34m"
#define GREEN "\033[0;32m"
#define RED "\033[0;31m"
#define RESET "\033[0m"
#define SCRIPT_NAME "uefi_stub_gen_output.sh"

void usage(const char *prog_name) {
    printf("Usage: %s\n", prog_name);
    printf("This is a simple interactive tool to automatically generate UEFI boot entries.\n");
    printf("It generates efibootmgr commands and exports them to a small executable.\n");
    printf("No changes will be written to disk before confirmation.\n");
    printf("The EFI partition must be mounted to /boot and the kernel and initramfs image must be located at the root of it!\n");
    printf("Some UEFI systems don't allow to create more than one EFI STUB entry.\n");
    printf("Unfortunately, efibootmgr is not able to change EFI entries. You always have to delete/overwrite entries to make changes happen.\n");
    printf("Please don't use this program if you don't exactly know what you are doing here and what EFI STUB means.\n");
    printf("You can get some great info at: https://wiki.archlinux.org/title/EFISTUB\n");
    printf("And now good luck with EFI STUB booting.\n");
    printf("Options:\n");
    printf("    -h, --help      Display this help message\n");
}

void execute_command(const char *cmd) {
    int status = system(cmd);
    if (status == -1) {
        printf("%sError: Failed to execute command: %s%s\n", RED, strerror(errno), RESET);
    } else if (WIFEXITED(status) && WEXITSTATUS(status) != 0) {
        printf("%sError: Command exited with status %d.%s\n", RED, WEXITSTATUS(status), RESET);
    } else {
        printf("%sCommand executed successfully.%s\n", GREEN, RESET);
    }
}

char* get_uuid(const char *device) {
    char cmd[512];
    snprintf(cmd, sizeof(cmd), "blkid -o value -s UUID %s", device);
    FILE *fp = popen(cmd, "r");
    if (!fp) {
        return NULL;
    }

    char *uuid = malloc(256);
    if (!uuid) {
        pclose(fp);
        return NULL;
    }

    if (fgets(uuid, 256, fp) != NULL) {
        uuid[strcspn(uuid, "\n")] = '\0';
    }
    pclose(fp);
    return uuid[0] ? uuid : NULL;
}

char* get_device_for_mountpoint(const char *mountpoint) {
    static char device[256];
    char cmd[512];
    snprintf(cmd, sizeof(cmd), "findmnt -n -o SOURCE %s", mountpoint);
    FILE *fp = popen(cmd, "r");
    if (!fp) {
        return NULL;
    }

    if (fgets(device, sizeof(device), fp) != NULL) {
        device[strcspn(device, "\n")] = '\0';
    }
    pclose(fp);
    return device[0] ? device : NULL;
}

int main(int argc, char *argv[]) {
    if (geteuid() != 0) {
        printf("%sThis script must be run with root privileges!%s\n", RED, RESET);
        printf("type: sudo %s -h for usage and more info.\n", argv[0]);
        return 1;
    }

    if (argc > 1) {
        if (strcmp(argv[1], "-h") == 0 || strcmp(argv[1], "--help") == 0) {
            usage(argv[0]);
            return 0;
        } else {
            printf("Unknown option: %s\n", argv[1]);
            usage(argv[0]);
            return 1;
        }
    }

    char choice;
    printf("%sWelcome to LUSC - A Linux UEFI STUB Creator\n", BLUE);
    printf("----------------------------------------\n");
    printf("----------------------------------------%s\n", RESET);
    printf("Start creating UEFI boot entries? (y/N) ");
    if (scanf(" %c", &choice) != 1) {
        printf("%sError reading input. Exiting.%s\n", RED, RESET);
        return 1;
    }
    choice = tolower(choice);

    if (choice != 'y') {
        printf("Goodbye. Exiting...\n");
        return 0;
    }

    char efi_partition[256];
    printf("Please specify EFI partition (e.g., /dev/nvme0n1p1): ");
    if (scanf("%255s", efi_partition) != 1) {
        printf("%sError reading EFI partition. Exiting.%s\n", RED, RESET);
        return 1;
    }

    char root_partition[256];
    printf("Please specify root partition (e.g., /dev/nvme0n1p2): ");
    if (scanf("%255s", root_partition) != 1) {
        printf("%sError reading root partition. Exiting.%s\n", RED, RESET);
        return 1;
    }

    char blkid_cmd[512];
    snprintf(blkid_cmd, sizeof(blkid_cmd), "blkid | grep -q %s", efi_partition);
    if (system(blkid_cmd) != 0) {
        printf("%sError: EFI partition '%s' not found!%s\n", RED, efi_partition, RESET);
        return 1;
    }

    snprintf(blkid_cmd, sizeof(blkid_cmd), "blkid | grep -q %s", root_partition);
    if (system(blkid_cmd) != 0) {
        printf("%sError: Root partition '%s' not found!%s\n", RED, root_partition, RESET);
        return 1;
    }

    char efi_disk[256];
    char efi_part_num[256];
    if (strstr(efi_partition, "/dev/nvme") == efi_partition) {
        sscanf(efi_partition, "%[^p]p%s", efi_disk, efi_part_num);
    } else {
        sscanf(efi_partition, "%[^0-9]%s", efi_disk, efi_part_num);
    }

    char boot_label[256];
    printf("Please specify the label for the boot entry (e.g., Arch Linux): ");
    if (scanf("%255s", boot_label) != 1) {
        printf("%sError reading boot label. Exiting.%s\n", RED, RESET);
        return 1;
    }

    char *efi_uuid = get_uuid(efi_partition);
    if (!efi_uuid) {
        printf("%sError retrieving UUID for EFI partition.%s\n", RED, RESET);
        return 1;
    }

    char *root_uuid = get_uuid(root_partition);
    if (!root_uuid) {
        printf("%sError retrieving UUID for root partition.%s\n", RED, RESET);
        free(efi_uuid); // Free allocated memory before returning
        return 1;
    }

    char default_params[512];
    snprintf(default_params, sizeof(default_params), "root=UUID=%s rw", root_uuid);

    char extra_params[512] = {0};
    printf("Current kernel parameters: %s\n", default_params);
    printf("%sinitrd and initrd-fallback will be added automatically!%s\n", GREEN, RESET);
    printf("Add additional kernel parameters (or press Enter to keep current): ");
    getchar(); // To consume the newline character left by the previous scanf
    if (fgets(extra_params, sizeof(extra_params), stdin)) {
        extra_params[strcspn(extra_params, "\n")] = '\0';
    }

    char kernel_params[1024];
    if (strlen(extra_params) > 0) {
        snprintf(kernel_params, sizeof(kernel_params), "%s %s", default_params, extra_params);
    } else {
        snprintf(kernel_params, sizeof(kernel_params), "%s", default_params);
    }

    const char *initramdisk = "\\initramfs-linux.img";
    const char *initfallback = "\\initramfs-linux-fallback.img";

    char linux_cmd[1024];
    char fallback_cmd[1024];
    snprintf(linux_cmd, sizeof(linux_cmd),
             "efibootmgr --create --disk %s --part %s --label \"%s\" --loader /vmlinuz-linux --unicode \"%s initrd=%s\" --verbose",
             efi_disk, efi_part_num, boot_label, kernel_params, initramdisk);
    snprintf(fallback_cmd, sizeof(fallback_cmd),
             "efibootmgr --create --disk %s --part %s --label \"%s (Fallback)\" --loader /vmlinuz-linux --unicode \"%s initrd=%s\" --verbose",
             efi_disk, efi_part_num, boot_label, kernel_params, initfallback);

    printf("Detected partitions:\n");
    printf("EFI: %s (%s)\n", efi_partition, efi_uuid);
    printf("Root: %s (%s)\n", root_partition, root_uuid);
    printf("\nComposed commands:\n");
    printf("%s\n", linux_cmd);
    printf("%s\n", fallback_cmd);

    char action;
    printf("\nCreate executable only, create and execute (sets UEFI boot entries), or abort? (c/ce/a) ");
    if (scanf(" %c", &action) != 1) {
        printf("%sError reading action choice. Exiting.%s\n", RED, RESET);
        free(efi_uuid); // Free allocated memory before returning
        free(root_uuid); // Free allocated memory before returning
        return 1;
    }
    action = tolower(action);

    FILE *script_fp = fopen(SCRIPT_NAME, "w");
    if (script_fp == NULL) {
        printf("%sError: Unable to create script file: %s%s\n", RED, strerror(errno), RESET);
        free(efi_uuid); // Free allocated memory before returning
        free(root_uuid); // Free allocated memory before returning
        return 1;
    }

    fprintf(script_fp, "#!/bin/bash\n");
    fprintf(script_fp, "# Generated UEFI boot entries by LUSC\n");
    fprintf(script_fp, "%s\n", fallback_cmd);
    fprintf(script_fp, "%s\n", linux_cmd);
    fprintf(script_fp, "exit 0\n");
    fclose(script_fp);
    chmod(SCRIPT_NAME, S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH);

    printf("Script file '%s' created.\n", SCRIPT_NAME);

    if (action == 'c') {
        printf("Executable created. Exiting...\n");
    } else if (action == 'ce') {
        printf("Executing script...\n");
        execute_command(SCRIPT_NAME);
    } else {
        printf("Aborted. Exiting...\n");
    }

    free(efi_uuid); // Free allocated memory before returning
    free(root_uuid); // Free allocated memory before returning
    return 0;
}
```