Домашнє завдання №28_3

** для компіляції і запуску на ПК потрібно використати компілятори gcc або clang у ОС Linux(Debian, Ubuntu, Arch Linux, тощо)*

Виконати домашнє завдання №27_1 повторно як альтернативну низькорівневу реалізацію домашнього завдання №28_1 мовою С. Для цього використати poll(системний виклик ОС Linux) з відстеженням POLLIN у revents з структури pollfd при передачі даних за допомогою неіменованого каналу(pipe) до обробника вводу(в прикладі він називається inputHandler). *(Це імітує process.stdin.on('keypress', inputHandler) з прикладу коду домашнього завдання №28_1).* Саму передачу даних виконує додатковий скануючий обробник(в прикладі він називається stdinInputHandler), який виступає в ролі аналога поведінки вбудованої реалізації process.stdin з NodeJS.

Замість використання системного виклику poll також можна використати системний виклик select, тоді відстежуватися буде безпосередньо неіменований канал.

**** коментар:*** *це завдання аналогічне №27_1, №27_2 та №27_3 і є повністю тотожне до завдань №28_2 та №28_2; таким чином можна порівняти різні засоби програмування; далі наводиться приклад повністю виконаного завдання.*

Вибір варіанту

Варіант завдання відповідає варіанту домашнього завдання №27_1

Приклад коду

Наведений зразок коду у першому лістингу реалізовує завдання для 5-ти максимально допустимих спроб введення ключа ліцензії за допомогою системного виклику poll.

Максимальна кількість спроб для введення ключа ліцензії	5
Оголошення в коді	`#define ATTEMPTS_COUNT 5`

Лістинг 1

```c
#include <stdio.h>
#ifdef __linux__
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <unistd.h> //
#include <poll.h> // #include <sys/poll.h>
//#include <sys/select.h>
```

```
#include <sys/wait.h>

#define ATTEMPTS_COUNT 5
int attemptsDownCount = ATTEMPTS_COUNT;

#define GROUPS_DIGITS_COUNT 5
#define GROUP_DIGITS_SIZE 5

const unsigned char PRODUCT_KEY_PART1[] = {
0xF, 0xF, 0xF, 0xF, 0xF,
0xD, 0xD, 0xD, 0xD, 0xD,
0x8, 0x8, 0x8, 0x8, 0x8,
0xB, 0xB, 0xB, 0xB, 0xB,
0xF, 0xF, 0xF, 0xF, 0xF
};

const unsigned char PRODUCT_KEY_PART2[] = {
0xE, 0xE, 0xE, 0xE, 0xE,
0xF, 0xF, 0xF, 0xF, 0xF,
0xB, 0xB, 0xB, 0xB, 0xB,
0xF, 0xF, 0xF, 0xF, 0xF,
0xA, 0xA, 0xA, 0xA, 0xA
};

#define DIGITS_COUNT (GROUPS_DIGITS_COUNT * GROUP_DIGITS_SIZE)

#define TYPER_FULL_RAW_MODE

#define IS_KEY_UP(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'A')
#define IS_KEY_DOWN(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'B')
#define IS_KEY_LEFT(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'D')
#define IS_KEY_RIGHT(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'C')
#define ESCAPE_SEQUENSE_INIT(CH0, CH1) { if (!kbhit()) CH1 = 0x1b; else if (CH0 == 0x1b) CH1 = getch(); }
#define IS_ESCAPE_SEQUENSE_PREPARE(CH0, CH1, CH2) {if(CH0 == 0x1b && CH1 == '[') CH2 = getch();}
#define IS_ESCAPE_KEY(CH0, CH1) (CH0 == 0x1b && CH1 == 0x1b)
#define IS_KEY_DELETE_PREPARE(CH0, CH1, CH2, CH3) {if(CH0 == 0x1b && CH1 == '[' && CH2 == '3') CH3 = getch();}
#define IS_KEY_DELETE(CH0, CH1, CH2, CH3) (CH0 == 0x1b && CH1 == '[' && CH2 == '3') // && CH3 == '^')
#define IS_KEY_BACKSPACE(CH0) (CH0 == 127)
#ifdef TYPER_FULL_RAW_MODE
#define IS_KEY_ENTER(CH0) (CH0 == 13)
#else
#define IS_KEY_ENTER(CH0) (CH0 == 10)
#endif
#define IS_KEY_CTRLC(CH0) (CH0 == 3)

int outOfEdgeIndex = 0;
int currIndex = 0;
unsigned char data[DIGITS_COUNT] = { 0 };

char checkProductKey(unsigned char * productKey){
    unsigned int index;
    for(index = 0; index < DIGITS_COUNT; ++index){
        if(productKey[index] ^ PRODUCT_KEY_PART1[index] ^ PRODUCT_KEY_PART2[index]){
            return 0;
        }
    }

    return ~0;
}

void toDigitPosition(unsigned int currIndex){
    char temp[16];
    int positionAddon = currIndex / GROUP_DIGITS_SIZE;
    positionAddon && positionAddon >= GROUPS_DIGITS_COUNT ? --positionAddon : 0;
    write(STDOUT_FILENO, "\033[64D", 5);
    if(currIndex += positionAddon){
        sprintf(temp, "\033[%dC", currIndex);
        write(STDOUT_FILENO, temp, strlen(temp));
    }
}

void printProductKey(unsigned char * productKey, unsigned int outOfEdgeIndex){
    unsigned int index;
    unsigned char value;
    for(index = 0; index < DIGITS_COUNT && index < outOfEdgeIndex; ++index){
        value = productKey[index];
        value > 9 ? (value += 'A' - 10) : (value += '0') ;
        write(STDOUT_FILENO, &value, 1);
    }
}
```

```
void printFormattedProductKey(unsigned char * productKey, unsigned int outOfEdgeIndex){
    unsigned int index;
    unsigned char value;
    for(index = 0; index < DIGITS_COUNT && index < outOfEdgeIndex; ++index){
        value = productKey[index];
        value > 9 ? (value += 'A' - 10) : (value += '0') ;
        write(STDOUT_FILENO, &value, 1);
        if(!((index + 1) % GROUP_DIGITS_SIZE) && (index + 1) < DIGITS_COUNT){
            write(STDOUT_FILENO, "-", 1);
        }
    }
}

static struct termios term, oterm;

static int getch(void){
    int c = 0;

    tcgetattr(0, &oterm);
    memcpy(&term, &oterm, sizeof(term));
#ifdef TYPER_FULL_RAW_MODE
    term.c_iflag |= IGNBRK;
    term.c_iflag &= ~(INLCR | ICRNL | IXON | IXOFF);
    term.c_lflag &= ~(ICANON | ECHO | ECHOK | ECHOE | ECHONL | ISIG | IEXTEN);
#else
      term.c_lflag &= ~(ICANON | ECHO);
#endif
    term.c_cc[VMIN] = 1;
    term.c_cc[VTIME] = 0;
    tcsetattr(0, TCSANOW, &term);
    c = getchar();
    tcsetattr(0, TCSANOW, &oterm);
    return c;
}

static int kbhit(void){
    int c = 0;

    tcgetattr(0, &oterm);
    memcpy(&term, &oterm, sizeof(term));
#ifndef TYPER_FULL_RAW_MODE
    term.c_iflag |= IGNBRK;
    term.c_iflag &= ~(INLCR | ICRNL | IXON | IXOFF);
    term.c_lflag &= ~(ICANON | ECHO | ECHOK | ECHOE | ECHONL | ISIG | IEXTEN);
#else
      term.c_lflag &= ~(ICANON | ECHO);
#endif
    term.c_cc[VMIN] = 0;
    term.c_cc[VTIME] = 1;
    tcsetattr(0, TCSANOW, &term);
    c = getchar();
    tcsetattr(0, TCSANOW, &oterm);
    if (c != -1) ungetc(c, stdin);
    return ((c != -1) ? 1 : 0);
}

void stdinInputHandler(int * ch0, int * ch1, int * ch2, int * ch3) {
    *ch0 = getch();
    *ch1 = 0; //
    ESCAPE_SEQUENSE_INIT(*ch0, *ch1);
    IS_ESCAPE_SEQUENSE_PREPARE(*ch0, *ch1, *ch2);
    IS_KEY_DELETE_PREPARE(*ch0, *ch1, *ch2, *ch3);
    if ( IS_KEY_CTRLC(*ch0) ) {
        // no action
    }
    else if ( IS_KEY_ENTER(*ch0) ) {
        // no action
    }
    else if (IS_KEY_BACKSPACE(*ch0)) {
        // no action
    }
    else if (IS_KEY_DELETE(*ch0, *ch1, *ch2, *ch3)) {
        // no action
    }
    else if (IS_KEY_LEFT(*ch0, *ch1, *ch2)) {
        // no action
    }
    else if (IS_KEY_RIGHT(*ch0, *ch1, *ch2)) {
        // no action
    }
        else if(IS_ESCAPE_KEY(*ch0, *ch1)){
```

```
        while (kbhit()) {
            getch();
        }
    }
}
void inputHandler(int ch0, int ch1, int ch2, int ch3){
    if(!attemptsDownCount){
      return;
    }
    if ( IS_KEY_ENTER(ch0) ) {
        if (checkProductKey(data) ) {
            write(STDOUT_FILENO, "\nThe product key is correct\n\n", 29);
            printProductKey(data, outOfEdgeIndex);
            write(STDOUT_FILENO, " (COMPLETE)", 11);
            write(STDOUT_FILENO, "\nFor exit press Ctrl + C\n", 25);
            attemptsDownCount = 0;
        }
        else{
            write(STDOUT_FILENO, "\nThe product key is not correct\n", 32);
            printf("\nYou have %d attempts to try\n", --attemptsDownCount);
            if(attemptsDownCount){
                write(STDOUT_FILENO, "Please, enter the product key:\n", 31);
                printFormattedProductKey(data, outOfEdgeIndex);
                toDigitPosition(currIndex);
            }
            else{
                write(STDOUT_FILENO, "The product key is not entered\n", 31);
                write(STDOUT_FILENO, "For exit press Ctrl + C\n", 24);
            }
        }
    }
    else if (IS_KEY_BACKSPACE(ch0)) {
        if(currIndex){
            --currIndex;
            toDigitPosition(currIndex);
            data[currIndex] = 0;
            write(STDOUT_FILENO, "0", 1);
            toDigitPosition(currIndex);
        }
    }
    else if (IS_KEY_DELETE(ch0, ch1, ch2, ch3)) {
        toDigitPosition(currIndex);
        data[currIndex] = 0;
        write(STDOUT_FILENO, "0", 1);
        toDigitPosition(currIndex);
    }
    else if (IS_KEY_LEFT(ch0, ch1, ch2)) {
        if(currIndex){
            toDigitPosition(--currIndex); // got to 1.5
        }
    }
    else if (IS_KEY_RIGHT(ch0, ch1, ch2)) {
        if(currIndex < outOfEdgeIndex){
            toDigitPosition(++currIndex);
        }
    }
    else if(IS_ESCAPE_KEY(ch0, ch1)){
        // no action
    }

    char chstr_[2] = { 0 };
    char * hexDigitScanfPattern = (char*)"%[0-9abcdefABCDEF]"; // /[0-9A-Fa-f]/g

    if (currIndex < DIGITS_COUNT && ch0 && sscanf((char*)&ch0, hexDigitScanfPattern, chstr_) > 0) {
        data[currIndex] = strtol(chstr_, NULL, 16);
        sprintf(chstr_, "%X", data[currIndex] );
        write(STDOUT_FILENO, chstr_, 1);
        if(outOfEdgeIndex <= currIndex){
            outOfEdgeIndex = currIndex + 1;
        }
        if(currIndex + 1 < DIGITS_COUNT) {
            ++currIndex;
            if (currIndex != DIGITS_COUNT && !(currIndex % 5)) {
                write(STDOUT_FILENO, "-", 1);
            }
        }
        if(currIndex + 1 == DIGITS_COUNT){
            toDigitPosition(currIndex);
        }
    }
}
```

```
void tx(struct pollfd * fdtab){
    int ch[4];
    unsigned char runState = ~0;

    close(fdtab[0].fd); // Close unused read end

    while(runState){
        stdinInputHandler(ch, ch + 1, ch + 2, ch + 3);
        write(fdtab[1].fd, ch, sizeof(int) * 4);
        if(ch[0] == 0x03){
            runState = 0;
        }
    }

    close(fdtab[1].fd); // Reader will see EOF
    wait(NULL); // Wait for child process
    exit(EXIT_SUCCESS);
}

void rx(struct pollfd * fdtab){
    int ch[4] = { 0 };
    unsigned char runState = ~0;
    int ret;

    close(fdtab[1].fd); // Close unused write end

    while(runState) {
        ret = poll(fdtab, 2, 1000000);

        if (!ret) {
            // without
            // no idle action
        }
        else if (ret > 0) {
            // used () for (fdtab[0].revents & POLLIN) for clang unwarning
            if (fdtab[0].revents & POLLIN && read(fdtab[0].fd, ch, 4 * sizeof(int)) > 0) {
                if(ch[0] == ' ' || ch[0] == '\t' ){
                    ch[0] = '0';
                }
                if(IS_KEY_CTRLC(ch[0])){
                    runState = 0;
                }else{
                    inputHandler(ch[0], ch[1], ch[2], ch[3]);
                }
            }
            if (fdtab[1].revents & POLLOUT) {
                // no action
            }
        }
        else {
            /* the poll failed */
            perror("poll failed");
        }
    }

    close(fdtab[0].fd);
    _exit(EXIT_SUCCESS);
}
#endif

int main(void){
#ifdef __linux__
    int pipefd[2];
    pid_t cpid;
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    struct pollfd fdtab[2];
    memset (fdtab, 0, sizeof(fdtab)); // not necessarily

    fdtab[0].fd = pipefd[0]; // read polled for input
    fdtab[0].events = POLLIN;
    fdtab[0].revents = 0;

    fdtab[1].fd = pipefd[1]; // write polled for output
    fdtab[1].events = POLLOUT;
    fdtab[1].revents = 0;

    cpid = fork();
```

```
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (cpid == 0) {
        write(STDOUT_FILENO, "Please, enter the product key:\n", 31);
        rx(fdtab);
    } else {
        tx(fdtab);
    }
#else
    printf("To run this program use Linux!\r\n");
    printf("Press Enter to continue!\r\n");
    (void)getchar();
#endif
}
```

Наведений зразок коду у другому лістингу реалізовує завдання для 5-ти максимально допустимих спроб введення ключа ліцензії за допомогою системного виклику select.

Максимальна кількість спроб для введення ключа ліцензії	5
Оголошення в коді	`#define ATTEMPTS_COUNT 5`

Для коректного виконання коду за допомогою https://repl.it/languages/c віртуальну консоль з правого боку краще трохи розширити перед початком виконання коду, а **у процесі виконання розмір консолі не змінювати**.

Лістинг 2

```
#include <stdio.h>
#ifdef __linux__
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <unistd.h> //
#include <sys/select.h>
#include <sys/wait.h>

#define ATTEMPTS_COUNT 5
int attemptsDownCount = ATTEMPTS_COUNT;

#define GROUPS_DIGITS_COUNT 5
#define GROUP_DIGITS_SIZE 5

const unsigned char PRODUCT_KEY_PART1[] = {
0xF, 0xF, 0xF, 0xF, 0xF,
0xD, 0xD, 0xD, 0xD, 0xD,
0x8, 0x8, 0x8, 0x8, 0x8,
0xB, 0xB, 0xB, 0xB, 0xB,
0xF, 0xF, 0xF, 0xF, 0xF
};

const unsigned char PRODUCT_KEY_PART2[] = {
0xE, 0xE, 0xE, 0xE, 0xE,
0xF, 0xF, 0xF, 0xF, 0xF,
0xB, 0xB, 0xB, 0xB, 0xB,
0xF, 0xF, 0xF, 0xF, 0xF,
0xA, 0xA, 0xA, 0xA, 0xA
};

#define DIGITS_COUNT (GROUPS_DIGITS_COUNT * GROUP_DIGITS_SIZE)

#define TYPER_FULL_RAW_MODE

#define IS_KEY_UP(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'A')
#define IS_KEY_DOWN(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'B')
#define IS_KEY_LEFT(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'D')
#define IS_KEY_RIGHT(CH0, CH1, CH2) (CH0 == 0x1b && CH1 == '[' && CH2 == 'C')
#define ESCAPE_SEQUENSE_INIT(CH0, CH1) { if (!kbhit()) CH1 = 0x1b; else if (CH0 == 0x1b) CH1 = getch(); }
#define IS_ESCAPE_SEQUENSE_PREPARE(CH0, CH1, CH2) {if(CH0 == 0x1b && CH1 == '[') CH2 = getch();}
#define IS_ESCAPE_KEY(CH0, CH1) (CH0 == 0x1b && CH1 == 0x1b)
```

```
#define IS_KEY_DELETE_PREPARE(CH0, CH1, CH2, CH3) {if(CH0 == 0x1b && CH1 == '[' && CH2 == '3') CH3 = getch();}
#define IS_KEY_DELETE(CH0, CH1, CH2, CH3) (CH0 == 0x1b && CH1 == '[' && CH2 == '3') // && CH3 == '^')
#define IS_KEY_BACKSPACE(CH0) (CH0 == 127)
#ifdef TYPER_FULL_RAW_MODE
#define IS_KEY_ENTER(CH0) (CH0 == 13)
#else
#define IS_KEY_ENTER(CH0) (CH0 == 10)
#endif
#define IS_KEY_CTRLC(CH0) (CH0 == 3)

int outOfEdgeIndex = 0;
int currIndex = 0;
unsigned char data[DIGITS_COUNT] = { 0 };

char checkProductKey(unsigned char * productKey){
    unsigned int index;
    for(index = 0; index < DIGITS_COUNT; ++index){
        if(productKey[index] ^ PRODUCT_KEY_PART1[index] ^ PRODUCT_KEY_PART2[index]){
            return 0;
        }
    }

    return ~0;
}

void toDigitPosition(unsigned int currIndex){
    char temp[16];
    int positionAddon = currIndex / GROUP_DIGITS_SIZE;
    positionAddon && positionAddon >= GROUPS_DIGITS_COUNT ? --positionAddon : 0;
    write(STDOUT_FILENO, "\033[64D", 5);
    if(currIndex += positionAddon){
        sprintf(temp, "\033[%dC", currIndex);
        write(STDOUT_FILENO, temp, strlen(temp));
    }
}

void printProductKey(unsigned char * productKey, unsigned int outOfEdgeIndex){
    unsigned int index;
    unsigned char value;
    for(index = 0; index < DIGITS_COUNT && index < outOfEdgeIndex; ++index){
        value = productKey[index];
        value > 9 ? (value += 'A' - 10) : (value += '0') ;
        write(STDOUT_FILENO, &value, 1);
    }
}

void printFormattedProductKey(unsigned char * productKey, unsigned int outOfEdgeIndex){
    unsigned int index;
    unsigned char value;
    for(index = 0; index < DIGITS_COUNT && index < outOfEdgeIndex; ++index){
        value = productKey[index];
        value > 9 ? (value += 'A' - 10) : (value += '0') ;
        write(STDOUT_FILENO, &value, 1);
        if(!((index + 1) % GROUP_DIGITS_SIZE) && (index + 1) < DIGITS_COUNT){
            write(STDOUT_FILENO, "-", 1);
        }
    }
}

static struct termios term, oterm;

static int getch(void){
    int c = 0;

    tcgetattr(0, &oterm);
    memcpy(&term, &oterm, sizeof(term));
#ifdef TYPER_FULL_RAW_MODE
    term.c_iflag |= IGNBRK;
    term.c_iflag &= ~(INLCR | ICRNL | IXON | IXOFF);
    term.c_lflag &= ~(ICANON | ECHO | ECHOK | ECHOE | ECHONL | ISIG | IEXTEN);
#else
      term.c_lflag &= ~(ICANON | ECHO);
#endif
    term.c_cc[VMIN] = 1;
    term.c_cc[VTIME] = 0;
    tcsetattr(0, TCSANOW, &term);
    c = getchar();
    tcsetattr(0, TCSANOW, &oterm);
    return c;
}
```

```
static int kbhit(void){
    int c = 0;

    tcgetattr(0, &oterm);
    memcpy(&term, &oterm, sizeof(term));
#ifndef TYPER_FULL_RAW_MODE
    term.c_iflag |= IGNBRK;
    term.c_iflag &= ~(INLCR | ICRNL | IXON | IXOFF);
    term.c_lflag &= ~(ICANON | ECHO | ECHOK | ECHOE | ECHONL | ISIG | IEXTEN);
#else
      term.c_lflag &= ~(ICANON | ECHO);
#endif
    term.c_cc[VMIN] = 0;
    term.c_cc[VTIME] = 1;
    tcsetattr(0, TCSANOW, &term);
    c = getchar();
    tcsetattr(0, TCSANOW, &oterm);
    if (c != -1) ungetc(c, stdin);
    return ((c != -1) ? 1 : 0);
}

void stdinInputHandler(int * ch0, int * ch1, int * ch2, int * ch3) {
    *ch0 = getch();
    *ch1 = 0; //
    ESCAPE_SEQUENSE_INIT(*ch0, *ch1);
    IS_ESCAPE_SEQUENSE_PREPARE(*ch0, *ch1, *ch2);
    IS_KEY_DELETE_PREPARE(*ch0, *ch1, *ch2, *ch3);
    if ( IS_KEY_CTRLC(*ch0) ) {
        // no action
    }
    else if ( IS_KEY_ENTER(*ch0) ) {
        // no action
    }
    else if (IS_KEY_BACKSPACE(*ch0)) {
        // no action
    }
    else if (IS_KEY_DELETE(*ch0, *ch1, *ch2, *ch3)) {
        // no action
    }
    else if (IS_KEY_LEFT(*ch0, *ch1, *ch2)) {
        // no action
    }
    else if (IS_KEY_RIGHT(*ch0, *ch1, *ch2)) {
        // no action
    }
        else if(IS_ESCAPE_KEY(*ch0, *ch1)){
        while (kbhit()) {
            getch();
        }
    }
}
void inputHandler(int ch0, int ch1, int ch2, int ch3){
    if(!attemptsDownCount){
      return;
    }
    if ( IS_KEY_ENTER(ch0) ) {
        if (checkProductKey(data) ) {
            write(STDOUT_FILENO, "\nThe product key is correct\n\n", 29);
            printProductKey(data, outOfEdgeIndex);
            write(STDOUT_FILENO, " (COMPLETE)", 11);
            write(STDOUT_FILENO, "\nFor exit press Ctrl + C\n", 25);
            attemptsDownCount = 0;
        }
        else{
            write(STDOUT_FILENO, "\nThe product key is not correct\n", 32);
            printf("\nYou have %d attempts to try\n", --attemptsDownCount);
            if(attemptsDownCount){
                write(STDOUT_FILENO, "Please, enter the product key:\n", 31);
                printFormattedProductKey(data, outOfEdgeIndex);
                toDigitPosition(currIndex);
            }
            else{
                write(STDOUT_FILENO, "The product key is not entered\n", 31);
                write(STDOUT_FILENO, "For exit press Ctrl + C\n", 24);
            }
        }
    }
    else if (IS_KEY_BACKSPACE(ch0)) {
        if(currIndex){
            --currIndex;
            toDigitPosition(currIndex);
```

```
            data[currIndex] = 0;
            write(STDOUT_FILENO, "0", 1);
            toDigitPosition(currIndex);
        }
    }
    else if (IS_KEY_DELETE(ch0, ch1, ch2, ch3)) {
        toDigitPosition(currIndex);
        data[currIndex] = 0;
        write(STDOUT_FILENO, "0", 1);
        toDigitPosition(currIndex);
    }
    else if (IS_KEY_LEFT(ch0, ch1, ch2)) {
        if(currIndex){
            toDigitPosition(--currIndex); // got to 1.5
        }
    }
    else if (IS_KEY_RIGHT(ch0, ch1, ch2)) {
        if(currIndex < outOfEdgeIndex){
            toDigitPosition(++currIndex);
        }
    }
    else if(IS_ESCAPE_KEY(ch0, ch1)){
        // no action
    }

    char chstr_[2] = { 0 };
    char * hexDigitScanfPattern = (char*)"%[0-9abcdefABCDEF]"; // /[0-9A-Fa-f]/g

    if (currIndex < DIGITS_COUNT && ch0 && sscanf((char*)&ch0, hexDigitScanfPattern, chstr_) > 0) {
        data[currIndex] = strtol(chstr_, NULL, 16);
        sprintf(chstr_, "%X", data[currIndex] );
        write(STDOUT_FILENO, chstr_, 1);
        if(outOfEdgeIndex <= currIndex){
            outOfEdgeIndex = currIndex + 1;
        }
        if(currIndex + 1 < DIGITS_COUNT) {
            ++currIndex;
            if (currIndex != DIGITS_COUNT && !(currIndex % 5)) {
                write(STDOUT_FILENO, "-", 1);
            }
        }
        if(currIndex + 1 == DIGITS_COUNT){
            toDigitPosition(currIndex);
        }
    }
}

void tx(int * pipefd){
    int ch[4];
    unsigned char runState = ~0;

    close(pipefd[0]); // Close unused read end

    while(runState){
        stdinInputHandler(ch, ch + 1, ch + 2, ch + 3);
        write(pipefd[1], ch, sizeof(int) * 4);
        if(ch[0] == 0x03){
            runState = 0;
        }
    }

    close(pipefd[1]); // Reader will see EOF
    wait(NULL); // Wait for child process
    exit(EXIT_SUCCESS);
}

void rx(int * pipefd){
    int ch[4] = { 0 };
    unsigned char runState = ~0;
    struct timeval timeout;
    fd_set readfds;
    int ret;

    close(pipefd[1]); // Close unused write end

    while(runState) {
        FD_ZERO(&readfds);
        FD_SET(pipefd[0], &readfds);

        timeout.tv_sec = 1000;
        timeout.tv_usec = 0;
```

```
        ret = select(FD_SETSIZE/*pipefd[0] + 1*/, &readfds, NULL, NULL, &timeout);

        if(!ret){
            // without
            // no idle action
        }
        else if (ret > 0){
            if (read(pipefd[0], ch, 4 * sizeof(int)) > 0){
                if(ch[0] == ' ' || ch[0] == '\t' ){
                    ch[0] = '0';
                }
                if(IS_KEY_CTRLC(ch[0])){
                    runState = 0;
                }else {
                    inputHandler(ch[0], ch[1], ch[2], ch[3]);
                }
            }
        }
        else{
            write(STDOUT_FILENO, "select failed\n", 14);
            //write(STDERR_FILENO, "select failed\n", 14);
        }
    }

    close(pipefd[0]);
    _exit(EXIT_SUCCESS);
}
#endif

int main(void){
#ifdef __linux__
    int pipefd[2];
    pid_t cpid;
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (cpid == 0) {
        write(STDOUT_FILENO, "Please, enter the product key:\n", 31);
        rx(pipefd);
    } else {
        tx(pipefd);
    }
#else
    printf("To run this program use Linux!\r\n");
    printf("Press Enter to continue!\r\n");
    (void)getchar();
#endif
}
```