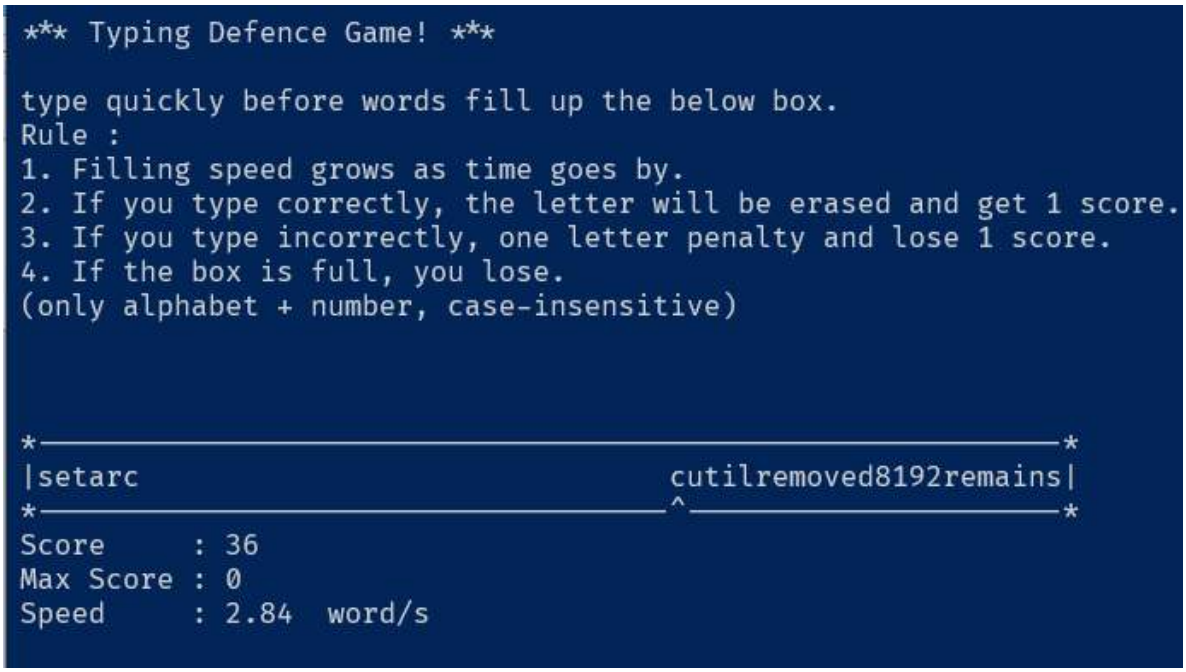
주제 : Typing Defence Game

(1) 주제 소개

Typing Defence Game은 계속 생겨나는 문자열을 빠르게 타이핑하여 없애 나가는 게임입니다.
점차 문자가 생성되는 속도가 빨라져 순식간에 어려워지며 아래 그림 하단에 보이는 상자가 가득 차게 되면 게임오버가 됩니다. 짧은 시간에 한 판을 플레이할 수 있고 타자 실력도 키울 수 있습니다.



<- 상자

(2) 기능 설계

<1> 사전 작업 : 단어 묶음 만들기 (document_picker)
게임을 시작하기 전에 먼저 상자에 자연스러운 단어들이 포함되도록
Linux 명령들(/usr/bin)의 매뉴얼 문서를 단어 단위로 Parsing해서 단어 묶음으로 구성하였습니다.
이 과정에서 특정 단어의 반복을 줄이기 위해 해싱으로 중복 단어는 제외하고 1000개를 선택했습니다.

fork() 함수로
단어 묶음 생성을
멀티프로세스로 처리
	`void game_reset()  //게임 초기화`
	`{`
	`  int pid;`
	`  SYSCALL_CHECK(pid = fork(), -1, "fork");`
	`  //child process : 새로운 단어 목록 만들기 (백그라운드로 미리 실행)*********`
	`  if (pid == 0) {`
	`  exit(write_wordList(wordPath));`
	`  }`
	`  //*************************************************************`
	(typing_defence.c 코드 일부)
	단어 묶음 생성에 시간이 다소 걸릴 수 있어 child process로 분리하여 게임 값 초기화 작업과 병행 실행

write_wordList
살펴보기
(document_picker.c)

write_wordList는 단어 모음을 담은 텍스트 파일 wordList.txt를 생성하는 함수입니다.

1. get_cmdList 함수로 /usr/bin 디렉토리를 조회하여 현재 시스템에서 사용할 수 있는 명령어 명단을 받아옵니다.

(디렉토리 관련 함수 다수 사용 : opendir, readdir, stat, closedir)

```
char** get_cmdList(int* cmdCountp)
{
    DIR* commandDir;
    struct dirent *fileEntry;
    struct stat fileStat;
    int commandCount = 0, commandMax = 200;
    char** commandList;

    SYSCALL_CHECK(
    commandList = (char**)malloc(sizeof(char*) * commandMax), NULL, "malloc");

    //1. 디렉토리 열기
    SYSCALL_CHECK(commandDir = opendir("/usr/bin"), NULL, "opendir");

    //2. 파일 엔트리에서 일반 파일만 명령어 목록(commandList)에 추가
    while ((fileEntry = readdir(commandDir)) != NULL)
    {
        char path[1024];
        sprintf(path, "/usr/bin/%s", fileEntry->d_name);
        SYSCALL_CHECK(stat(path, &fileStat), -1, "stat");

        if (S_ISREG(fileStat.st_mode))
        {
            int len = strlen(fileEntry->d_name);
            SYSCALL_CHECK(
```

2. extract_words 함수로, 명령어 명단 중에서 무작위로 하나 선택하여 그 명령의 매뉴얼을 열고, 파싱하면서 단어를 생성합니다.

```
srand(time(NULL)); //Random seed


while (wordCount < WORDCOUNT_MAX)
{
    const char* cmd = cmdList[rand() % cmdCount];
    // man 호출에 10번 연속 실패하면 프로그램 종료
    if ((result = extract_words(cmd, fd, &wordCount)) != 0
        && ++failStreak == 10)
    {
        fprintf(stderr, "Failed to extract words from manual\n");
        exit(1);
    } else {
        failStreak = 0;
    }
}
```

- 명령어에 대한 매뉴얼이 없는 때도 있는데, 이때는 명령어 다시 무작위선택
- 단어가 1000개 생성될 때까지 매뉴얼 문서 파싱
  (부족한 경우 다른 명령어 무작위 선택)

```
문자열을 빠르게 검색하기 위해 해싱하여 저장하는 프로그램
1. 단순하게 문자열 맨 앞, 맨 뒤의 문자로 해싱합니다. (충돌되는 경우 체이닝)
2. 키에 상응하는 값은 없습니다.
3. 테이블은 오직 1개이며, 문자열을 깊은 복사(동적할당)하기 때문에 사용 후 해제해야 누수가 없습니다.
```

- 특정 단어가 많이 나오지 않도록 해싱(simplehash.h, 구현 방식 위 참고)을 사용하여 1000개의 단어는 모두 서로 다름

- 매뉴얼 문서를 얻은 방법은, fork() 실행한 뒤 child process에서 man 명령(execlp)의 표준 출력이 maunal.txt에 쓰여지도록 open() - dup2() - close()를 이용합니다.

```
int extract_words(const char* cmd, int fdout, int* wordCountp)
{
    int pid;
    SYSCALL_CHECK(pid = fork(), -1, "fork");

    if (pid == 0)
    {
        //child process : Load manual to file
        //메뉴얼 내용은 manual.txt에 저장된다.
        int fd, newfd;

        SYSCALL_CHECK(
            fd = open("./manual.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666), -1, "open");
        SYSCALL_CHECK(
            newfd = dup2(fd, 1), -1, "dup2"); //stdout to file
        if (newfd != 1)
        {
            fprintf(stderr, "dup2 error\n");
            exit(1);
        }
        close(2);  //stderr to null

        execlp("man", "man", cmd, NULL);
        perror("execlp");
        exit(1);
    }
```

- 파싱 과정 (extract_words 함수의 일부)

```
char buf[BUFSIZ];
char plainWord[WORDLEN_MAX + 1] = {'\0'};
while (read(fdin, buf, BUFSIZ) > 0)
{
    char* word = strtok(buf, " /-\n\t");  //tokenize
    while (word != NULL)
    {
        //단어에서 알파벳(소문자로), 숫자만 추출
        int plainLen = 0;
        for (int i = 0; plainLen < WORDLEN_MAX && word[i] != '\0'; i++)
        {
            if (isdigit(word[i]))
                plainWord[plainLen++] = word[i];
            else if (isalpha(word[i]))
                plainWord[plainLen++] = tolower(word[i]);
        }
        plainWord[plainLen] = '\0';

        if (WORDLEN_MIN <= plainLen && hashInsert(plainWord, plainLen) == true)
        //길이가 적당하고, 중복되지 않은 단어를 테이블에 추가
        {
            SYSCALL_CHECK(write(fdout, plainWord, plainLen), -1, "write");
            SYSCALL_CHECK(write(fdout, "\n", 1), -1, "write");

            if (++(*wordCountp) == WORDCOUNT_MAX)
            {
                close(fdin);
                return 0;   //단어 목록이 다 찼으면 종료
            }
        }
        word = strtok(NULL, " /-\n\t");
    }
}
```

<2> main 함수 살펴보기.

```c
int main(void)
{
    initscr();
    crmode();
    noecho();
    clear();

    //1. 게임 초기화
    game_reset();

    //2. 게임 시작 전 메인 메뉴
    if (game_init() == 1)
    {
        endwin();
        return 0;
    }


    //3. 게임 진행 창 <-> 게임 오버 창 -> 끝내기
    while (1)
    {
        game_progress();
        if (game_over() == 1)
            break;
        else {
            game_reset(); //초기화하고 다시 시작
        }
    }

    endwin();
    return 0;
}
```

1. **<u>lcurses 라이브러리</u>** 함수들

2. game_reset() : 여기서 <1> 단어 묶음 만들기 그리고 게임 내 변수 초기화를 진행합니다.
3. game_init() : 여기서 게임 설명, 점수, 속도 정보 등 정적인 정보들을 화면에 출력합니다.

4. game_progress() : 게임 시작 키(SPACEBAR)를 눌렀을 때 게임을 동적으로 진행 시킵니다. 게임오버 조건이 되면 함수가 종료됩니다.

5. game_over() : 게임오버 시 게임 결과를 처리하고 게임 다시 하기 여부를 묻습니다. (0 반환 시 다시 하기)

<3> game_reset 함수 살펴보기

1. <1> 단어 묶음 만들기 작업을 child process에서 처리하는 동안 parent process는 게임에 사용되는 변수를 초기화합니다.

```c
//parent process
//1. 게임 값 초기화
front = rear = score = qSize = 0;
speed = TIMER_INTERVAL_MS;
speedUpInterval = SPEEDUP_CHARCNT;
speedUpRemain = (int)speedUpInterval;
memset(circularQueue, ' ', sizeof(circularQueue) - 1);
memset(buf, '\0', sizeof(buf));
memset(wordEntry, '\0', sizeof(wordEntry));
charPtr = NULL;
srand(time(NULL));
move(BOX_ROW, 0);
addstr(BOX);
```

2. 단어 묶음이 들어있는 wordList.txt 파일을 열어서 통째로 buf 문자열 배열에 담은 뒤에, 순차 탐색하면서 char* wordEntry[1000] 배열의 n번째 원소는, buf 문자열에서 n번째 단어의 시작 주소를 담게 합니다.

(예시 : printf("%s", wordEntry[n]) -> n번째 단어 출력)

```c
//2. 단어 목록 가져와서 wordEntry에 보관하기.
int fd;
SYSCALL_CHECK(
    fd = open(wordPath, O_RDONLY), -1, "open");

switch (read(fd, buf, sizeof(buf)))
{
    case -1:
        perror("read");
        exit(1);
    case sizeof(buf) / sizeof(buf[0]):
        fprintf(stderr, "wordList.txt is too big\n");
        exit(1);
};

int idx = 0, entryIndex = 0;
for (int i = 0; buf[i] != '\0'; ++i)
{
    if (buf[i] == '\n')
    {
        buf[i] = '\0';
        wordEntry[entryIndex++] = &buf[idx];
        idx = i + 1;
    }
}
```

<4> game_progress 함수 설명

1. 상자에 문자가 10개 채워진 상태로 시작합니다. (상자 상한 60개)
(상자는 원형 큐로 관리됩니다.)

2. 특정 시간 간격으로 글자가 1개씩 추가되게 하도록 set_ticker() 함수로 timer를 설정하였습니다.

timer 주기 도달 시, **signal()** 함수로 addBox 함수가 호출되도록 하였습니다.
addBox는 addQueue()로 상자에 문자를 1개 추가하고 refreshBox()로 상자의 상태를 갱신합니다.

3. while 내 부분은 사용자의 타이핑 입력을 getch(non-blocking)로 받는 부분입니다.
올바른 입력을 하면 그 글자가 지워지고 1점을 획득하지만,
잘못된 입력을 하면 글자가 1개 늘어나고 1점을 잃게 되어 있습니다.

타이핑 입력 후에는 박스, 점수 정보를 업데이트합니다.

```
void game_progress()    //게임 진행 창
{
    //1. 단어 10개 채우고 시작하기
    for (int i = 0; i < 10; ++i)
        addQueue();
    refreshBox();


    move(KEYINFO_ROW, 0); //키 설명 지우기
    addstr(KEYINFO_CLEAR);

    //2. 게임 진행
    signal(SIGALRM, addBox);    //자동 문자 생성 시그널
    timeout(5);                 //getch Non-blocking 설정
    set_ticker((int)speed);     //타이머 시작

    int ch;

    while (qSize < BOX_SIZE)
    //게임 유지 조건
    {
        ch = getch();

        if (ch != ERR)  //문자 입력이 있는 경우
        {
            //move(21, 0);
            //printw("ch : %c, qSize = %d", ch, qSize);

            if (isalpha(ch))
                ch = tolower(ch);

            if (ch == circularQueue[front] && qSize != 0) //맞는 입력
            {
                score++;
                circularQueue[front] = ' ';
                move(CURSOR_ROW, CURSOR_COL + front);
                addch('-');
                front = (front + 1) % BOX_SIZE;
                move(CURSOR_ROW, CURSOR_COL + front);
                addch('^'); //커서 이동
                qSize--;

            } else {                        //틀린 입력
                score--;
                addQueue();                 //(1개 패널티)
            }

            //박스 갱신
            refreshBox();

            //점수 갱신
            move(SCORE_ROW, 0);
            printw("Score     : %-10d", score);
            move(LINES-1, COLS-1);
        }
    }
```

<5> addQueue() 함수 살펴보기
상자에 문자를 1개 추가하는 함수입니다. (상자는 원형 큐로 관리되고 있음)
문자가 추가되는 시간(타이머)에 도달될 때 혹은 사용자가 입력을 잘못했을 때 호출됩니다.
이 함수는 다음과 같은 기능 추가로 있습니다.
1. 이전 단어를 다 읽었을 때 새로운 단어를 단어 묶음(<1>) 1000개 중에서 무작위로 하나 선택하기.
2. 특정 주기마다 문자 생성 속도를 늘리기 위해 속도(speed) 값을 높이고 타이머를 set_ticker 함수를 호출하여 재설정합니다.

```
bool addQueue()        //문자를 큐에 추가 (+ 큐가 꽉 차면 false 반환)
{
    if (qSize >= BOX_SIZE) {
        return false;
    }
    else {
        //단어 읽기가 끝났거나 처음이면 새로운 단어 무작위 선택
        if (charPtr == NULL || *charPtr == '\0') {
            charPtr = wordEntry[rand() % WORDCOUNT_MAX];
        }
        //큐에 추가
        circularQueue[rear] = *charPtr;
        rear = (rear + 1) % BOX_SIZE;
        charPtr++;
        qSize++;

        //특정 문자 수 마다 속도 증가
        if (--speedUpRemain == 0) {
            speedUpInterval *= (1.0 / SPEEDUP_FACTOR);
            speedUpRemain = (int)speedUpInterval;
            speed *= SPEEDUP_FACTOR;
            set_ticker((int)speed);
            move(SPEED_ROW, 0);
            printw("Speed     : %-5.2f word/s   ", 1000.0 / speed);
        }
        return true;
    }
}
```

(3) 수행 결과
- 시스템 프로그래밍 강의에서 배운 리눅스의 각종 시스템 콜을 골고루 활용할 수 있었습니다.
1. lcurse 라이브러리 : CUI 게임 제작에 필요
2. 디렉토리 관련 함수(stat, opendir, closedir 등) : /usr/bin에 어떤 명령어(프로그램)가 있는 지 조회하기 위해 사용하였습니다.
3. 파이프 활용 : extract_words 함수에서 open-dup2-close 기법으로 man 명령어 실행 결과가 파일에 출력되도록 했다. + execlp()
4. 시그널과 타이머 사용 : 특정 시간마다 문자열 상자에 문자가 1개 추가되도록 SIGALRM 시그널의 핸들러 함수와 타이머를 활용하였습니다.
5. 멀티 프로세스 프로그래밍 : fork()를 2번 사용하여 총 3개의 프로세스를 생성하는 프로그램을 작성하였습니다. (하나는 단어 묶음 생성용, 다른 하나는 man 명령 실행용)

- 간단히 즐길 수 있을 만큼 기능이 갖추어진 게임을 구현할 수 있었습니다.

초기 화면

```
** Typing Defence Game! **

type quickly before words fill up the below box.
Rule :
1. Filling speed grows as time goes by.
2. If you type correctly, the letter will be erased and get 1 score.
3. If you type incorrectly, one letter penalty and lose 1 score.
4. If the box is full, you lose.
(only alphabet + number, case-insensitive)

Press SPACEBAR key to (re)start the game.
Press @(Shift+2) key to quit the game.
*------------------------------------------------------------*
|                                                            |
*^-----------------------------------------------------------*
Score     : 0
Max Score : 0
Speed     : 1.46  word/s
```

플레이 화면

```
** Typing Defence Game! **

type quickly before words fill up the below box.
Rule :
1. Filling speed grows as time goes by.
2. If you type correctly, the letter will be erased and get 1 score.
3. If you type incorrectly, one letter penalty and lose 1 score.
4. If the box is full, you lose.
(only alphabet + number, case-insensitive)



*------------------------------------------------------------*
|                 significantcontradictstructureclearerser  |
*-----------------^------------------------------------------*
Score     : 78
Max Score : 0
Speed     : 3.71  word/s
```

플레이 결과 화면

```
** Typing Defence Game! **

type quickly before words fill up the below box.
Rule :
1. Filling speed grows as time goes by.
2. If you type correctly, the letter will be erased and get 1 score.
3. If you type incorrectly, one letter penalty and lose 1 score.
4. If the box is full, you lose.
(only alphabet + number, case-insensitive)

Press SPACEBAR key to (re)start the game.
Press @(Shift+2) key to quit the game.
*------------------------------------------------------------*
|  GAME OVER..  New Record! : 78                             |
*------------------------------------------------------------*
Score     : 78
Max Score : 78
Speed     : 4.24  word/s
```

(단어 배치는 매 게임마다 완전히 무작위입니다. /usr/bin에 있는 명령은 수백 개나 되며, 그 중에서 추출된 단어들 1000개 내에서도 무작위 배치됩니다.)

(4) 이슈 사항 및 해결

1. IPC (Inter-process Communication) 문제
이 프로그램은 fork()로 생성한 parent / child process 간에 데이터 통신이 필요하였습니다.
(명령어 매뉴얼 문서 문자열 이동, 단어 묶음을 담은 문자열 이동)
조사한 바에 따르면 공유 메모리 방식이 가장 느린 보조 저장장치를 사용할 필요가 없어 가장 쾌적한 방법이라고 생각했지만, 구현 방법이 상당히 복잡하고 처음 보는 내용이라 적용하는 데 어려움이 있었습니다.
최근에 배운 내용인 pipe 방식을 고려해보기도 하였는데, pipe 방식은 용량이 큰 명령어 매뉴얼 문서 내용(execlp(man, ...) 출력)을 모두 옮기기에는 버퍼 용량이 많이 부족해 pipe 입력 쪽에서 블록될 위험이 크다고 판단했고 비동기로 처리 시 구현 및 디버깅이 어려울 것이라 생각해 사용하지 않기로 했습니다.
그래서 저는 파일 입출력을 이용하여 구현하였습니다. 이 방식은 컴퓨터에서 가장 느린 보조 저장장치에 상주하는 파일을 이용하기 때문에 처리가 느리다는 단점이 있지만, 프로그램 구현 과정에서 중간 결과를 확인하기 쉽고 제게 익숙한 방식이었기 때문에 채용했습니다.

2. 자식 프로세스 디버깅 문제
저는 시스템프로그래밍 강의 실습 시에 Visual Studio Code 리눅스 버전에서 프로그램 작성하는 것을 선호하는데, 평소에 구현에 어려움이 있을 때마다 디버거를 많이 활용해왔습니다. 그런데 이번 강의에서 멀티프로세스 방식을 처음 접했고, 이 프로젝트에서 멀티프로세스를 본격적으로 적용하였는데 segment fault 등으로 구현 과정에 막힘이 있을 때 fork() 지점에서 프로세스가 나누어질 때 디버거 프로그램 조작을 어떻게 할지 몰라 헤맸습니다.
stackoverflow 등을 참고하여 launch.json을 수정해 멀티프로세스 디버깅을 시도하는 데는 성공했으나 흐름이 복잡해져서 크게 도움은 못 받았고 막상 코드에서 잘못된 부분도 자식 프로세스 구현과 아무 관련 없는 곳에 있었습니다. 이번 프로젝트에선 크게 도움은 안 되었으나 미래에 복잡한 프로젝트를 할 때는 크게 도움이 되겠다고 생각했습니다.

(5) 매뉴얼 (사용 방법)

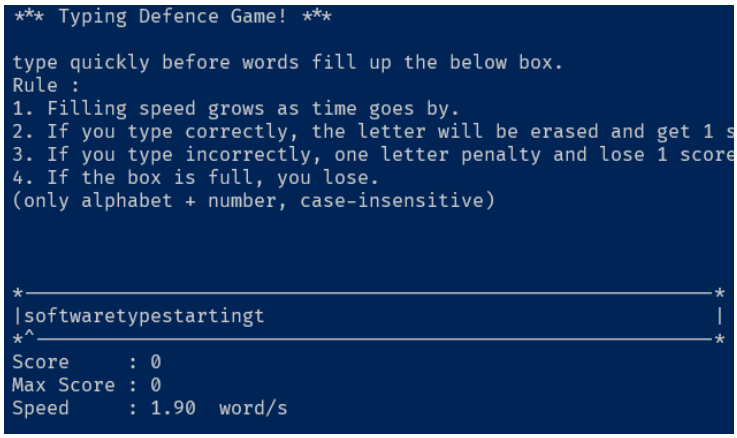
- 프로그램 실행 방법

make로 쉽게 실행 가능합니다.
1. 프로젝트가 있는 디렉토리로 이동합니다.
2. make를 입력하면 컴파일/링킹하여 실행 가능한 파일을 생성합니다.
3. make run을 입력하면 프로그램(게임)을 실행합니다. (프로그램이 없는 경우 컴파일/링킹 후 실행)
4. make clean을 입력하면 소스코드, 레포트, make 파일을 제외한 모든 파일을 삭제합니다.
(실행 파일, 프로그램 컴파일/링킹/실행으로 생겨난 부수 파일들)
5. 2번 make로 만든 실행 가능 파일의 명칭은 ./typing_defence입니다.

- 게임 플레이 방법

1. SPACEBAR 키를 눌러 시작, @(Shift + 2)키를 눌러 종료합니다.

2. 게임을 시작하면,
오른쪽 그림 하단의 상자에서 글자가 하나 하나씩 계속 생겨납니다.
상자에 글자가 꽉 차면 게임오버이며, 플레이어는 글자가 꽉 차지 않게 생겨나는 속도보다 빠르게 타이핑하여 최대한 버텨야 합니다.



3.
^로 가리킨 곳의 글자를 올바르게 누르면 그 문자가 사라지고 점수가 1점 더해지며, ^는 한 칸 오른쪽으로 이동합니다.
(대소문자 신경 쓰지 않으며, 알파벳 또는 숫자만 출현합니다.)
글자를 잘못 누르면 문자는 사라지지 않고 점수가 1점 감점되고 즉시 상자에 문자가 1개 늘어납니다.

4. 오른쪽 끝까지 문자가 도달하면 이어서 왼쪽 처음부터 글자가 나옵니다. 이는 타이핑할 때도 마찬가지입니다.

5. 일정 간격(약 5초)마다 점점 생성속도가 빨라집니다. 현재 생성속도는 하단에 표시됩니다. (위 그림에서 1초당 1.9글자) 언제까지 살아남을 수 있을까요?

6. 게임오버가 되면 상자 안에 GAME OVER라고 표시되고, 점수 신기록을 달성할 경우 이를 알려줍니다. 이 상태에서 SPACEBAR를 누르면 새 게임, @를 누르면 게임이 종료됩니다.
(최대 점수가 따로 저장되지는 않습니다.)