Matcom-invasion

Abel Ponce González

Mauricio Sunde Jiménez

November 15, 2024

Contents

Descripción

En este proyecto, se desarrollará un juego de estilo arcade en lenguaje C utilizando la biblioteca ncurses, SDL y otras inspirado en el clásico “Alien Invaders”. El jugador controlará una nave espacial con el objetivo de defender la Tierra de una invasión alienígena. Este proyecto tiene como objetivo principal aplicar conceptos fundamentales de la asignatura de Sistemas Operativos, tales como la programación orientada a eventos, la concurrencia, la gestión de memoria y el diseño de estrategias de planificación.

Chapter 1

Memoria

1.1 Block

```c
typedef struct Block
{
    int size;
    int start;
    struct Block *next;
} Block;
```

Definiendo los bloques que serán las particiones en la memoria.

1.2 initialize_memory()

```c
void initialize_memory()
{
    freeList = (Block *)malloc(sizeof(Block));
    freeList->size = MEMORY_SIZE;
    freeList->start = 0;
    freeList->next = NULL;
}
```

Esta función crea un bloque libre que abarca toda la memoria disponible y lo inicializa con los valores adecuados. Este bloque se convierte en el primer y único bloque de la lista de bloques libres al inicio del programa.

1.3 first_fit()

```c
void *first_fit_allocate(int size)
{
    Block *current = freeList;
    Block *previous = NULL;

    while (current != NULL)
    {
        if (current->size >= size)
        {
            // Asignar memoria
            void *allocated_memory = &memory[current->start]; //direccion de memoria

            // Actualizar la lista de bloques libres
            current->start += size;
            current->size -= size;

            if (current->size == 0)
            {
                // Eliminar el bloque si está completamente utilizado
                if (previous == NULL)
                {
                    freeList = current->next;
                }
                else
                {
                    previous->next = current->next;
                }
                free(current);
            }
            return allocated_memory;
        }
        previous = current;
        current = current->next;
    }
    return NULL; // No hay suficiente memoria
}
```

Algoritmo first fit donde se recorre la free list desde el principio hasta encontrar el primer bloque que cumple con las condiciones para ser ocupado. Si el bloque es mas grande se divide en dos partes, una parte se asigna al proceso y la otra permanece en la free list. Si el bloque se utiliza completamente, se elimina de la lista de bloques libres. Retorna NULL si no encuentra un bloque que cumpla la condición de espacio.

1.4 free_memory(void ptr, int size)

```c
void free_memory(void *ptr, int size)
{
    Block *newBlock = (Block *)malloc(sizeof(Block));
    newBlock->size = size;
    newBlock->start = (char *)ptr - memory;
    newBlock->next = freeList;
    freeList = newBlock;
}
```

Esta función crea un nuevo bloque de memoria libre con el tamaño y la posición especificados y lo añade al inicio de la free list. Esto permite que la memoria liberada pueda ser reutilizada.

Chapter 2

Semáforos

2.1 Zonas críticas

Una zona crítica es una sección del código donde varios hilos acceden a un mismo recurso en memoria, lo que puede generar problemas de concurrencia y afectar la correcta implementación del programa. En este caso, las zonas críticas se encuentran en las operaciones de escritura en la consola proporcionada por la biblioteca ncurses. Para solucionar este problema, se ha creado el semáforo enemigo_disparo_bomba. Este semáforo se activa cada vez que se va a realizar un movimiento del cursor y una escritura en la consola, asegurando que solo un hilo pueda ejecutar estas operaciones a la vez. Los demás hilos deben esperar a que el hilo en ejecución termine antes de poder acceder a la zona crítica. De esta manera, se garantiza la integridad de los datos y se evita la interferencia entre hilos.

2.2 Ejemplos de Zonas críticas

```
void *Enemy_generator()
{
    while (1)
    {
        sem_wait(&sem_enemigo_disparo_bomba);
        Create_enemys(5);
        sem_post(&sem_enemigo_disparo_bomba);
        sleep(3);
    }
}
```

```c
void *MOVER_BOMBAS()
{
    // Mover bombas - Estrategia FCFS
    for (int i = 0; i < Max_BOMBAS; i++)
    {
        if (!bomba[i].activo)
            continue; // Saltar si la bomba no está activa

        sem_wait(&sem_enemigo_disparo_bomba);

        if (bomba[i].y < LINES - 1) // Si la bomba no ha tocado el suelo
        {
            // Borrar la bomba en la posición anterior
            move(bomba[i].y - 1, bomba[i].x);
            addch(' ');

            // Mover la bomba a la nueva posición
            move(bomba[i].y, bomba[i].x);
            addch('o');

            // Incrementar la posición en Y para la próxima iteración
            bomba[i].y++;
        }
        else
        {
            // Si la bomba toca tierra, desactivarla y limpiar su rastro
            bomba[i].activo = 0;
            --Bombas_Actuales;
            move(bomba[i].y - 1, bomba[i].x);
            addch(' ');
        }

        sem_post(&sem_enemigo_disparo_bomba);
    }
}
```

Chapter 3

Nave

3.1 nave

```
struct nave
{
    int x,y;
    int vida;
    char ch;
};
```

Estructura que tiene una posición, nos dice si tiene vida y tiene un símbolo propio.

3.2 VERIFICAR_ENTRADA_MOV_NAVE()

```c
void *VERIFICAR_ENTRADA_MOV_NAVE()
{
    sem_wait(&sem_enemigo_disparo_bomba);

    if (input == KEY_LEFT && nave.x > 4)
    {
        move(nave.y, nave.x);
        addch(' ');
        nave.x--;
    }
    else if (input == KEY_RIGHT && nave.x <= COLS - 5)
    {
        move(nave.y, nave.x);
        addch(' ');
        nave.x++;
    }

    else if (input == KEY_UP && nave.y > 6)
    {
        move(nave.y, nave.x);
        addch(' ');
        nave.y--;
    }
    else if (input == KEY_DOWN && nave.y <= LINES - 3)
    {
        move(nave.y, nave.x);
        addch(' ');
        nave.y++;
    }

    sem_post(&sem_enemigo_disparo_bomba);
}
```

Gestiona el movimiento de la nave en respuesta a las entradas del usuario, asegurando que solo un hilo acceda a la consola a la vez mediante el uso de un semáforo. Esto evita conflictos y garantiza un comportamiento correcto del juego.

Chapter 4

Enemigos

4.1 enemigo

```c
typedef struct enemigo
{
    int x,y;
    int px,py;
    int vivo; // 1 = vivo 0 = muerto
    int direccion; //  0 = izq 1 = derecha
    char ch;
    struct enemigo *next;
}enemigo;
```

Estructura que definimos con sus coordenadas, las coordenadas previas, si se encuentra vivo, su dirección de movimiento y que tipo de alien es.

4.2 Add_enemy()

```c
void Add_enemy2()
{
    enemigo *new_enemy = (enemigo *)first_fit_allocate(sizeof(enemigo));
    int x = rand() % COLS - 5;

    if (new_enemy == NULL)
    {
        return;// no hay espacio en memoria para anadir otro enemigo si eliminas enemigos entonces se vuelven a generar
    }

    new_enemy->direccion = rand() % 2;
    new_enemy->y = 6;
    if(x <= 5) new_enemy->x = 7;
    else new_enemy->x = x;
    new_enemy->vivo = 1;
    new_enemy->ch = Random_type();
    new_enemy->next = NULL;

    // Insertar al final de la lista de enemigos
    if (enemy == NULL)
    {
        enemy = new_enemy;
        lastEnemy = new_enemy;
    }
    else
    {
        lastEnemy->next = new_enemy;
        lastEnemy = new_enemy;
    }
}
```

Esta función agrega un nuevo enemigo a la lista de enemigos, verificando si hay suficiente espacio en memoria.

4.3 Create_enemys()

```c
void *Create_enemys(int number_enemys)
{
    int n = number_enemys;
    for (int i = 0; i < n; i++)
    {
        Add_enemy2();
    }
}
```

Función que se basa en Add_enemy() e incrementa un contador de enemigos.

4.4 Enemy_generator()

```
void *Enemy_generator()
{
    while (1)
    {
        sem_wait(&sem_enemigo_disparo_bomba);
        Create_enemys(5);
        sem_post(&sem_enemigo_disparo_bomba);
        sleep(3);
    }
}
```

Esta función se encuentra en un hilo que estará generando cada cierto tiempo enemigos hasta que se llene la memoria.

4.5 Delete_enemy(enemigo delete)

```
void Delete_enemy(enemigo *delete)
{
    enemigo *current = enemy;
    enemigo *prev = current;

    if (delete == enemy)
    {
        enemy = delete->next;
        free_memory(delete, sizeof(enemigo)); // Liberar memoria usando free_memory()
    }
    else
    {
        while (current->next != delete)
        {
            current = current->next;
        }
        prev = current;
        current = current->next;

        if (current == lastEnemy)
        {
            lastEnemy = prev;
        }

        prev->next = current->next;
        free_memory(current, sizeof(enemigo)); // Liberar memoria usando free_memory()
    }
}
```

Esta función elimina un enemigo especifico de la lista de enemigos, actualiza los punteros de la lista según sea necesario y libera la memoria asociada al enemigo eliminado.

4.6 free_enemy()

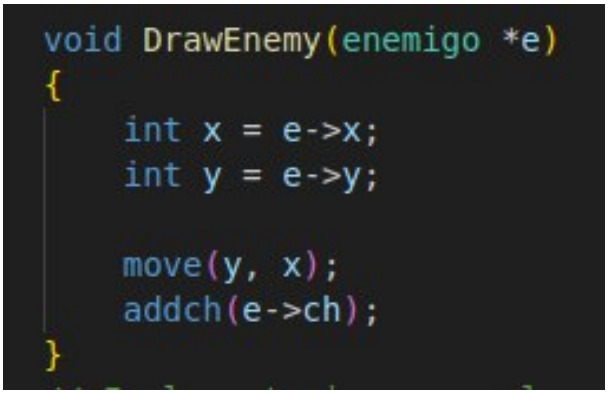
```c
void free_enemys()
{
    enemigo *current = enemy;
    enemigo *next;

    while (current != NULL)
    {
        next = current->next;
        free_memory(current, sizeof(enemigo)); // Liberar memoria usando free_memory()
        current = next;
    }

    enemy = NULL;
    lastEnemy = NULL;
}
```

Recorre la lista de enemigos liberando la memoria de cada enemigo y finalmente restablece la lista a un estado vacío.

4.7 DrawEnemy(enemigo e)



Dibuja los enemigos en consola.

4.8 MOVER_ENEMIGOS()

```c
void *MOVER_ENEMIGOS()
{
    enemigo *current = enemy;
    int i = 0;
    while (current != NULL)
    {
        sem_wait(&sem_enemigo_disparo_bomba);
        if (current->vivo == 1)
        {
            // Borrar la posición anterior del enemigo
            move(current->py, current->px);
            addch(' ');
            refresh();
            // Mover el cursor a la nueva posición
            move(current->y, current->x);
            // Cambiar el color del enemigo basado en el índice `i`
            if (i % 6 == 0) attron(COLOR_PAIR(ALIEN_COLOR_1));
            else if (i % 6 == 1) attron(COLOR_PAIR(ALIEN_COLOR_2));
            else if (i % 6 == 2) attron(COLOR_PAIR(ALIEN_COLOR_3));
            else if (i % 6 == 3) attron(COLOR_PAIR(ALIEN_COLOR_4));
            else if (i % 6 == 4) attron(COLOR_PAIR(ALIEN_COLOR_5));
            else if (i % 6 == 5) attron(COLOR_PAIR(ALIEN_COLOR_6));
            else attron(COLOR_PAIR(NAVE_COLOR));
            // Dibujar el enemigo en la nueva posición
            addch(current->ch);
            // Restablecer el color al predeterminado
            attron(COLOR_PAIR(NAVE_COLOR));
            // Actualizar las posiciones anteriores del enemigo
            current->py = current->y;
            current->px = current->x;
            // Verificar si el enemigo tiene que soltar una bomba
            int mi_random = 1 + (rand() % 100);
            if ((options.chanceBomba - mi_random) >= 0 && Bombas_Actuales < Max_BOMBAS)
            {
                for (int j = 0; j < Max_BOMBAS; j++)
                {
                    if (bomba[j].activo == 0)
                    {
                        // Activar una nueva bomba en la posición del enemigo
                        bomba[j].activo = 1;
                        Bombas_Actuales++;
                        bomba[j].y = current->y + 1;
                        bomba[j].x = current->x;
                        break;
                    }
                }
            }
            // Establecer la nueva posición del enemigo
            if (current->direccion == 0)
                current->x--;
            else if (current->direccion == 1)
                current->x++;
            // Verificar si el enemigo ha llegado a los límites de la pantalla
            if (current->x == COLS - 5)
            {
                current->y++;
                current->direccion = 0;
            }
            else if (current->x == 4)
            {
                current->y++;
                current->direccion = 1;
            }
        }
        sem_post(&sem_enemigo_disparo_bomba);
        current = current->next;
        i++;
    }
    refresh();
}
```

Mueve los enemigos, actualiza sus posiciones en la pantalla, cambia los colores y gestiona el lanzamiento de bombas.

Chapter 5

Disparo

5.1 disparo

```
struct disparo
{
    int x,y;
    int activo;
    char ch;
};
```

Estructura que tiene una posición, nos dice si esta activa y tiene un símbolo propio.

5.2 VERIFICAR_ENTRADA_DISPARAR()

```
void *VERIFICAR_ENTRADA_DISPARAR()
{
    if (input == ' ' && Disparos_Actuales < 3)
    {
        playSoud_Shoot();
        for (int i = 0; i < 3; i++)
        {
            // Si el tienes disparos disponibles
            if (disparo[i].activo == 0)
            {
                disparo[i].activo = 1;
                Disparos_Actuales++;
                --score;
                // posicionar disparo sobre la nave
                disparo[i].y = nave.y - 1;
                disparo[i].x = nave.x;
                break;
            }
        }
    }
}
```

Gestiona la entrada del usuario para disparar proyectiles, asegurando que no haya mas de 3 disparos activos al mismo tiempo, reproduce un sonido de disparo y posiciona el disparo justo encima de la nave.

5.3 MOVER_DISPAROS()

```
void *MOVER_DISPAROS()
{
    for (int i = 0; i < 3; i++)
    {
        if (!disparo[i].activo)
            continue;

        sem_wait(&sem_enemigo_disparo_bomba);
        // Borrar el disparo en la posición anterior
        if (disparo[i].y < LINES - 2)
        {
            move(disparo[i].y + 1, disparo[i].x);
            addch(' ');
        }
        // Verificar colisiones con enemigos
        enemigo *current = enemy;
        while (current != NULL)
        {
            if (current->vivo && current->y == disparo[i].y && current->x == disparo[i].x)
            {
                // Actualizar estado después de la colisión
                playSoud_Impact();
                score += 20;
                current->vivo = 0;
                disparo[i].activo = 0;
                --Disparos_Actuales;
                --Enemigos_Actuales;
                move(current->py, current->px);
                addch(' ');
                refresh();
                Delete_enemy(current);
                break;
            }
            current = current->next;
        }
        // Si el disparo sigue activo después de la verificación de colisiones
        if (disparo[i].activo)
        {
            // Mover el disparo a la nueva posición
            move(disparo[i].y, disparo[i].x);
            addch(disparo[i].ch);
            disparo[i].y--;
        }
        else
        {
            // Si el disparo ya salió de la pantalla, eliminarlo
            move(disparo[i].y + 1, disparo[i].x);
            addch(' ');
        }
        // Si el disparo ya ha salido de la pantalla
        if (disparo[i].y <= 5)
        {
            disparo[i].activo = 0;
            --Disparos_Actuales;
            move(disparo[i].y + 1, disparo[i].x);
            addch(' ');
        }
        sem_post(&sem_enemigo_disparo_bomba);
    }
}
```

Gestiona el movimiento de los disparos, verifica colisiones con los enemigos, actualiza la pantalla y asegura que los disparos se desactiven cuando salen de la pantalla o colisionan con un enemigo.

Chapter 6

Bombas

6.1 bomba

```
struct bomba
{
    int x,y;
    int activo;
    char ch;
};
```

Estructura que tiene una posición, nos dice si esta activa y tiene un símbolo propio.

6.2 MOVER_BOMBAS()

```c
void *MOVER_BOMBAS()
{
    // Mover bombas - Estrategia FCFS
    for (int i = 0; i < Max_BOMBAS; i++)
    {
        if (!bomba[i].activo)
            continue; // Saltar si la bomba no está activa

        sem_wait(&sem_enemigo_disparo_bomba);

        if (bomba[i].y < LINES - 1) // Si la bomba no ha tocado el suelo
        {
            // Borrar la bomba en la posición anterior
            move(bomba[i].y - 1, bomba[i].x);
            addch(' ');

            // Mover la bomba a la nueva posición
            move(bomba[i].y, bomba[i].x);
            addch('o');

            // Incrementar la posición en Y para la próxima iteración
            bomba[i].y++;
        }
        else
        {
            // Si la bomba toca tierra, desactivarla y limpiar su rastro
            bomba[i].activo = 0;
            --Bombas_Actuales;
            move(bomba[i].y - 1, bomba[i].x);
            addch(' ');
        }

        sem_post(&sem_enemigo_disparo_bomba);
    }
}
```

Gestiona el movimiento de las bombas en el juego, asegurando que se muevan hacia abajo en la pantalla, verificando si han alcanzado el suelo y actualizando la pantalla en consecuencia.

Chapter 7

Sonido

7.1 initAudio()

```
void initAudio()
{
    if (SDL_Init(SDL_INIT_AUDIO) < 0)
    {
        fprintf(stderr, "Error inicializando SDL: %s\n", SDL_GetError());
        exit(1);
    }
    if (Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 2048) < 0)
    {
        fprintf(stderr, "Error inicializando SDL_mixer: %s\n", Mix_GetError());
        exit(1);
    }

    sonido_disparo = Mix_LoadWAV("disparo.wav");
    sonido_colision = Mix_LoadWAV("colision.wav");
    sonido_Fondo = Mix_LoadMUS("laserTag2.wav");
    sonido_Win = Mix_LoadWAV("arcade-game-winner.mp3");
    sonido_gameOver = Mix_LoadWAV("game-over.wav");

    if (!sonido_gameOver || !sonido_disparo || !sonido_colision || !sonido_Fondo || !sonido_Win)
    {
        fprintf(stderr, "Error cargando sonido: %s \n", Mix_GetError());
        exit(1);
    }
}
```

Utilizando la biblioteca SDL2 y SDL2- mixer inicializa el sistema de audio y carga varios archivos de sonido necesarios para el juego, asegurándose de que todos los recursos de audio estén disponibles y listos para su uso.

7.2 clouseAudio()

```
void clouseAudio()
{
    Mix_FreeChunk(sonido_disparo);
    Mix_FreeChunk(sonido_colision);
    Mix_FreeMusic(sonido_Fondo);
    Mix_FreeChunk(sonido_gameOver);
    Mix_FreeChunk(sonido_Win);
    Mix_CloseAudio();
    SDL_Quit();
}
```

Detiene la reproducción de música y efectos de sonido, libera la memoria utilizada por los recursos de audio, cierra el sistema de audio y finaliza SDL, asegurando que todos los recursos se liberen correctamente.