SISTEMAS OPERATIVOS
PROYECTO FINAL

realizado por
Miguel González Bustamante
Grupo 2261, pareja 13

08/05/2019

Introducción

El contenido de este proyecto se compone de una carpeta llamada ‘src’ en la que se encuentran los ficheros .c y .h, el makefile para su compilación y este documento que explica la funcionalidad desarrollada.

Resumen

Se ha implementado un proceso simulador que genere un ‘combate’ entre diferentes procesos nave de diferentes procesos equipo.

La comunicación entre el simulador y los equipos se ha desarrollado mediante tuberías así como la de los equipos con su grupo de naves. Sin embargo, los procesos nave se comunican con el simulador por una cola de mensajes.

Existe una alarma que indica el turno a los equipos para que comuniquen a las naves las acciones que estas deben tomar. Se permiten hasta dos acciones por turno. Las acciones implementadas son las de movimiento aleatorio, movimiento mediante rastreo, ataque y finalización.

El proceso puede finalizar mediante interrupción por teclado (control + C) o cuando solo quede un equipo en el mapa. El mapa se encuentra en un segmento de memoria compartida para que tanto el proceso simulador como el proceso monitor pueda acceder a él.

Proceso Simulador

Inicialización de recursos

En primer lugar genera los recursos necesarios para la comunicación entre procesos y para almacenar los datos que se generen. Estos recursos son:

- Cola de mensajes: se crea con funcionalidad para poder escribir y leer de ella, ya que será heredada por los procesos naves cuando sean generados. De esta forma, las naves escribirán en ella y el simulador recibirá sus mensajes.
- Segmento de memoria compartida: ha sido desarrollado en el método *shm_create().* Esta función crea el segmento, la redimensiona para que ocupe el tamaño de la estructura *tipo_mapa* y finalmente lo mapea a la variable global *mapa*.
- Semáforo de control: ha sido creado para gestionar el acceso al segmento de memoria compartida por parte del proceso monitor. Esto nos permite ejecutar el proceso monitor el cual esperará a que ejecutemos el simulador para continuar con su ejecución. Si no existiera este semáforo, al intentar ejecutar antes el monitor intentaría abrir un segmento inexistente y se produciría un error.
- Tuberías simulador a jefes: se trata de una matriz de *N_EQUIPOS* tuberías. Con ella el simulador es capaz de mantener la comunicación con los equipos.
- Manejador de la señal SIGINT: se crea después de la generación de procesos nave y controla que cuando el usuario introduzca la señal SIGINT (control + C), se liberen todos los recursos y se espere a la finalización de los procesos hijos.
- Manejador de la señal SIGALRM: controla la señal de alarma que se produce cada *TURNO_SECS* segundos.

Una vez creados estos recursos, se vacían todas las posiciones del mapa a modo de inicialización para su posterior visualización en el monitor.

Generación de procesos jefe

A continuación, el simulador procede a la generación de sus procesos hijo. Estos han sido denominados ‘jefe’ o ‘equipo’ ya que son los que gestionan los procesos nave.

La creación de estos procesos se realiza en un bucle for de 0 a *N_EQUIPOS* mediante la función fork().

No se ha realizado ningún control mediante señales que compruebe la creación de estos procesos para que comiencen su funcionalidad cuando lo indique el simulador, aunque esto sería una mejora para el proyecto ya que en el estado actual el proceso que sea creado en primer lugar comenzará antes su ejecución. Esto es una ventaja frente a otros procesos pues probablemente, aunque no con total seguridad, escriban primero en la cola de mensajes y se tramite su acción primero. De todas maneras, el componente de aleatoriedad que brindan los movimientos ayuda a que no sea una funcionalidad indispensable y por ello no a sido realizada.

Funcionalidad

Una vez se han creado los recursos y procesos jefe, el simulador establece una alarma de *TURNO_SECS* segundos. Después duerme este tiempo más un segundo para que les dé tiempo a los procesos nave a escribir en la cola de mensajes, si no se realiza este *sleep(),* la función *mq_receive()* devuelve un error. No es lo normal, ya que debería esperar hasta que recibiera un mensaje pero es un arreglo poco óptimo que ha funcionado.

En este momento, el simulador entra en un bucle del que sólo sale si recibe la señal SIGINT terminando el proceso o cuando un equipo gane la batalla y se liberen todos los recursos.

Dentro de dicho bucle, lo primero que se realiza es la recepción de mensajes por la cola de mensajes. El procesamiento del mensaje se ha desarrollado en una función aparte llamada *simulador_update()* para no colapsar de líneas de código la función *main()*. En este método se comprueba en primer lugar que la nave que envía el mensaje siga viva en el momento que se procesa la acción ya que puede ocurrir que para cuando esta se envíe esté viva pero cuando se procese ya haya muerto. En caso de que esté viva, el siguiente paso a seguir es comprobar la acción que desea realizar. Solo existen dos tipos, mover o atacar, aunque cada una de ellas puede procesarse de forma diferente respecto al estado del mapa en ese momento.

Si se trata de la acción 'mover', el simulador comprobará si la posición a la que se desea desplazar está vacía. Aquí se ha detectado un fallo en la función *mapa_is_casilla_vacia()*, ya que en ocasiones cuando procesos del mismo equipo tratan de desplazarse a una casilla vacía en un mismo turno es posible que acaben ambos en la misma posición. No ocurre el 100% de las veces pero si con frecuencia. En cualquier caso, he optado por sacar la casilla y comprobar si tiene un equipo para solucionar este error. Por lo que si la casilla está vacía, se ordena al mapa vaciar la posición actual en la que se encuentre la nave y después se cambia su posición a la que se quiere desplazar. Si estuviera ocupada no se realizaría ningún cambio en el mapa.

Por el contrario, si la acción es 'ataque', el primero paso es enviar el misil mediante la función *mapa_send_misil()* que se encarga de mostrar la animación por el monitor. En caso de que la casilla a la que se envía no esté ocupada por una nave, se rellenará con el símbolo de agua (w) y en caso contrario se comprobará la vida que tiene dicha nave. Si ésta, tras la reducción del daño del impacto, es menor que cero se procederá a destruir dicha nave. Para ello se cambia su estado de viva a *false,* se muestra el símbolo de nave destruida y se envía al equipo de la nave la instrucción 'DESTRUIR <nº de nave>'. Si la vida de la nave es mayor que cero tras la reducción, se muestra el símbolo de nave tocada.

Por último y tras salir de la función, se realiza un sleep de un tiempo mayor que el de refresco de pantalla y de la animación de *mapa_send_misil()* para que sea posible apreciar por el monitor las acciones que se van mostrando. Aunque podría ralentizarse si se incrementa el número de naves, el número de equipos o se reduce el tiempo por turno.

Proceso Jefe

Inicialización de recursos

Este proceso es creado por el simulador y se encarga principalmente de establecer la comunicación con los procesos naves que crea. Para ello crea los siguientes recursos:

- Tuberías jefe a naves: se trata de una matriz de *N_NAVES* tuberías. Con ella el jefe es capaz de mantener la comunicación con las naves.
- Array pid_naves: lista que almacena los identificadores de las naves de ese equipo para poder destruirlos mandandoles la señal SIGTERM.

Tras crear estos recursos indica mediante la función *mapa_set_num_naves()* el número de naves del equipo.

Generación de procesos nave

Mediante un bucle for de 0 a *N_NAVES* crea los procesos naves de cada equipo. En el código del padre (pid devuelto por *fork()* > 0) se guardan los identificadores de los procesos hijos para su posterior destrucción.

Una vez más, una mejora que implementara señales para controlar la creación de las naves hubiera sido más acertada.

Funcionalidad

Tras crear los procesos naves, el proceso entra en un bucle del que solo saldrá una vez finalice la partida o reciba una señal SIGINT. Dentro de este bucle se capturan los mensajes enviados por el simulador y se envían las acciones necesarias a su grupo de naves.

En primer lugar se recibe un mensaje por la tubería común con el simulador. Las acciones posibles que puede mandar el simulador indican el comienzo de un nuevo turno, la finalización de la ejecución o la destrucción de alguna de sus naves.

Si la acción indica la llegada de un nuevo turno se ha optado por enviar siempre el mismo mensaje a las naves, atacar. En el apartado de funcionalidad de los procesos nave se explica el porqué de esta decisión. La acción FIN sirve para finalizar la ejecución de este proceso pero antes de ello manda la señal SIGTERM a sus naves para que finalicen primero. Para esto se guardaron los identificadores de los procesos nave en su creación. Después de mandarla espera a que terminen su ejecución y finaliza la suya. Por último, la acción DESTRUIR <nº de nave> manda un mensaje a la nave en cuestión para que no realice más acciones pues ha sido destruida por otra nave. Esta función no finaliza su ejecución simplemente en adelante no se procesarán las acciones que le mande el proceso jefe.

Proceso Nave

Inicialización de recursos

- Manejador de la señal SIGTERM: necesario para capturar la señal SIGTERM enviada por el proceso jefe de esta nave después de que reciba la instrucción finalizar.
- tipo_nave: estructura con los datos de una nave.
- tipo_acción: estructura con los datos de una acción realizada por una nave.

Funcionalidad

La primera acción que toma este proceso es la de crear la estructura de la nave. Para ello se ha implementado una función *nave _create()* que inicializa sus parámetros y la coloca en el mapa en función del número de equipos que se hayan definido. Si el número de equipos es menor o igual que cuatro, las naves se situarán en función de su equipo en una esquina del mapa, de la forma que se muestra en la siguiente imagen (*figura 1.a*):

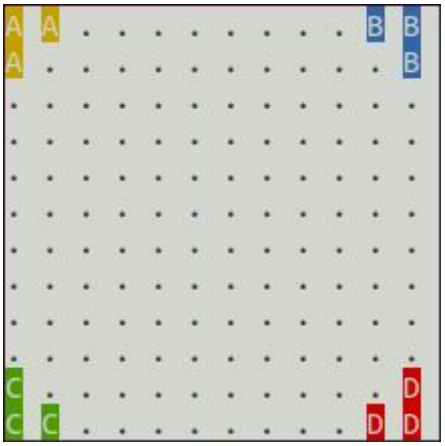

figura 1.a: Inicialización estándar de naves.

Si por el contrario, el número de equipos es mayor que cuatro, la disposición en el mapa será la siguiente (*figura 1.b*):

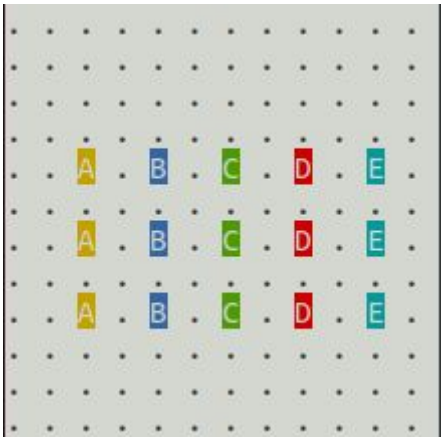

figura 1.b: Inicialización secundaria de naves.

Tras crear y posicionar las naves se procede a la captura de mensajes por parte del proceso jefe. Las acciones que pueden recibirse solamente son dos: atacar o finalizar. He decidido no incluir una acción de movimiento por agilizar la simulación, ya que, si el ataque no se puede realizar se cambia a hacer un movimiento. De esta forma, siempre se sigue el mismo esquema atacar/mover->mover, siendo el segundo movimiento obligatorio. En un principio, opté por realizar aleatoriamente la elección de la acción (siendo posible moverse o atacar), pero la simulación tardaba mucho en arrancar ya que lo normal es que las naves no puedan dispararse desde la posición inicial, con lo cual perdían alrededor de un 50% de las acciones por turno. Por ello, al final decidí que la elección de la acción fuese estática y que si el ataque inicial fallaba, se pasase a realizar una acción de movimiento.

El ataque en sí se calcula en una función llamada *nave_atacar()* cuya funcionalidad es detectar si la nave más cercana está dentro del rango de alcance definido por *ATAQUE_ALCANCE.* Si es así se enviaría la acción de atacar. La detección de la nave más cercana se realiza en la función *nave_rastrear()* la cual itera sobre todas las naves de los equipos enemigos para detectar cual es la que se encuentra más cerca. Esta función también se utiliza en la siguiente acción.

El movimiento de las naves también ha sufrido una modificación por agilizar la ejecución de la simulación. Existen dos tipos de movimiento, uno aleatorio y otro direccionado a la nave más próxima. Es este último el que también hace uso de la función *nave_rastrear().* Una vez detecta que nave está más cerca se mueve en la dirección que esta se encuentre, con esta medida los movimiento pierden cierta aleatoriedad pero ganan en rendimiento. Ya que en ocasiones al final de la partida, cuando quedan un par de naves, si estas están muy separadas en el tablero, tardan mucho en juntarse. Este movimiento se puede ver en la siguiente imagen (*figura 2.a*):

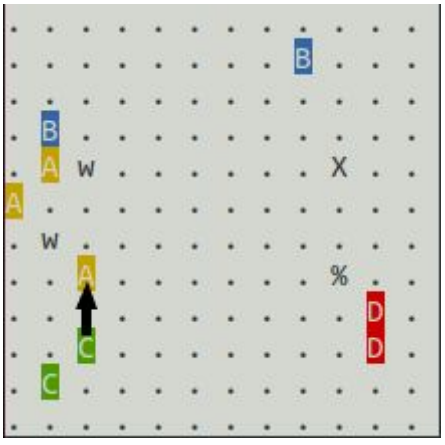

figura 2.a: movimiento direccionado.

Para conservar los movimientos aleatorios, se ha mantenido que uno de las dos acciones que se realicen sea la de un movimiento aleatorio. No importa si la primera es un ataque o un movimiento direccionado, la segunda acción siempre va a ser un movimiento aleatorio. Este se calcula mediante las dos funciones *accion_moverAleatorioY()* y *accion_moverAleatorioX()*, las cuales calculan una posición aleatoria entre la de la nave más su alcance de movimiento y la de la nave menos su alcance de movimiento. Siempre que no se rebasen los límites del mapa. Este movimiento se puede ver en la siguiente imagen (*figura 2.b*):

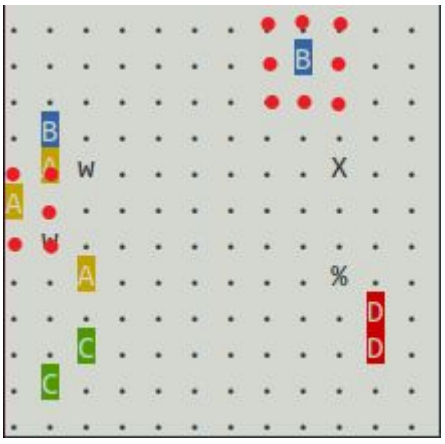

figura 2.b: movimiento aleatorio.

Finalmente la acción FIN solo llega cuando el proceso nave ha sido destruido por otra nave, de tal forma que activa un flag para que no se realicen más acciones, pero no termina su ejecución todavía.

Cabe destacar que algunas acciones que se realizan en tiempo de ejecución del proceso nave pueden no ser realizables al llegar al proceso simulador. Este tipo de acciones son descartadas y la nave no cambiaría su estado. Por ejemplo podemos ver en la *figura 2.b* como el proceso A podría desplazarse aleatoriamente a la posición de su compañero de equipo, pero no lo hace porque el simulador descarta tal situación.

Proceso Monitor

De este proceso sólo cabe destacar que contiene un semáforo que impide que se ejecute si el simulador no ha creado antes el segmento de memoria compartida. Y la ampliación de la información mostrada por pantalla mediante la librería ncurses. El estado de una partida por el monitor se ve como se muestra a continuación (*figura 3.a*):

figura 3.a: salida de ejecución por monitor.

y cuando finaliza su ejecución porque un equipo ha ganado (*figura 3.b*):

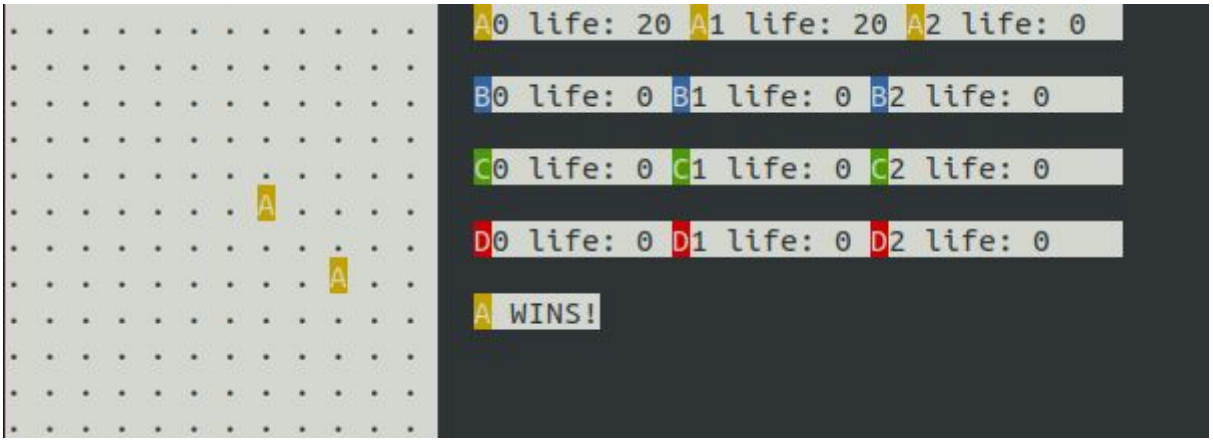

figura 3.b: salida de ejecución por monitor, tras finalizar.

Esquema del proyecto

A continuación se incluye un esquema gráfico del proyecto con alguno de sus componentes principales (*figura 4*):

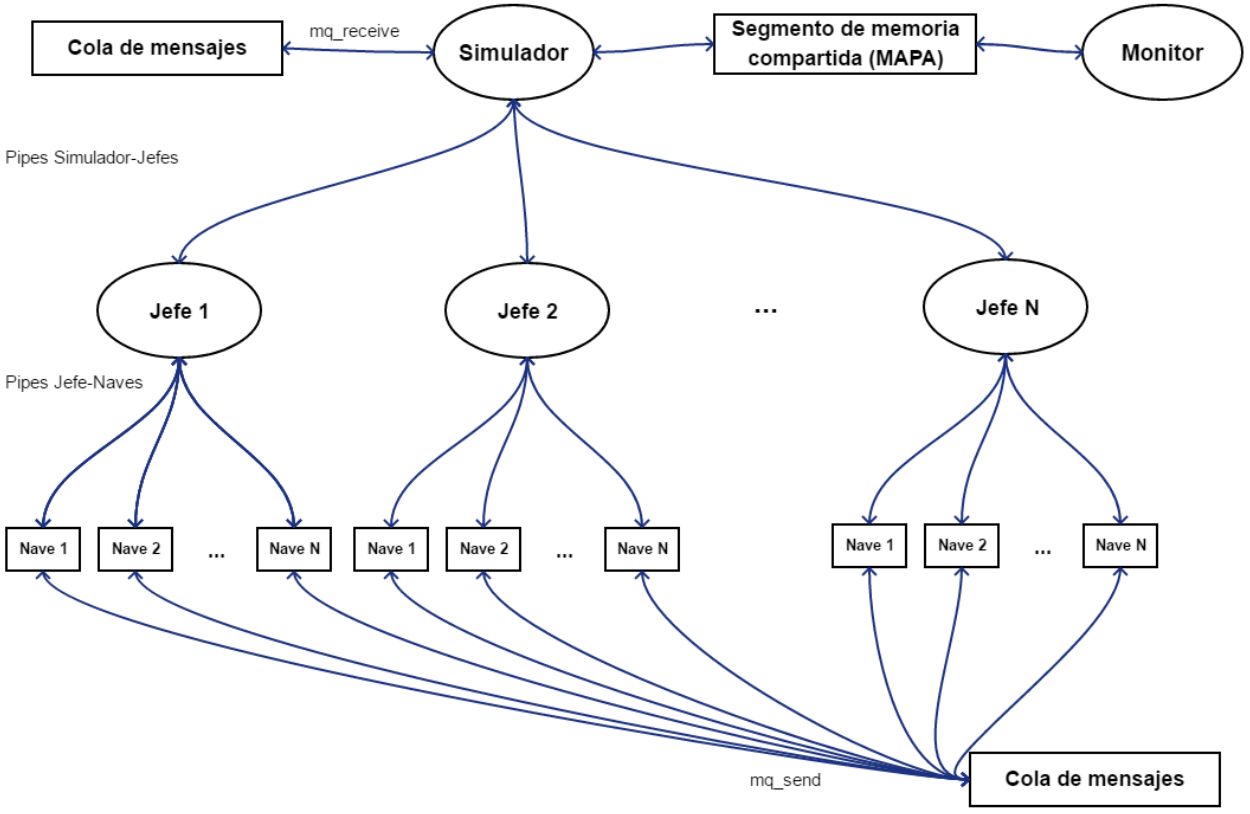

figura 4: esquema del proyecto.

*La cola de mensajes se representa partida pero es una unidad.