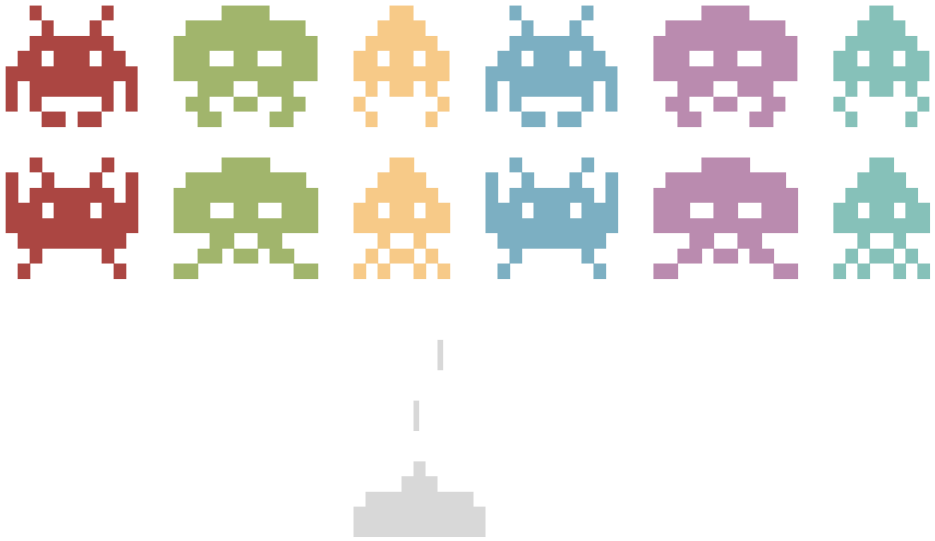


Proyecto: Simulador Espacial

Sistemas Operativos
2018 – 2019

Pedro Burgos Gonzalo
Juan González Domínguez

GRUPO 2272

Introducción

Este proyecto consiste en la realización de un simulador espacial basado en el uso y comunicación del proceso de Linux. Se implementará, por una parte, un programa lance los procesos y ejecute la simulación, y por otra, un programa monitor mediante el uso de la librería ncurses, que muestre un mapa en tiempo real del estado de la simulación.

Resumen

Hemos optado por una implementación dividida en módulos. De esta forma, hemos organizado el proyecto mediante la siguiente estructura de ficheros:

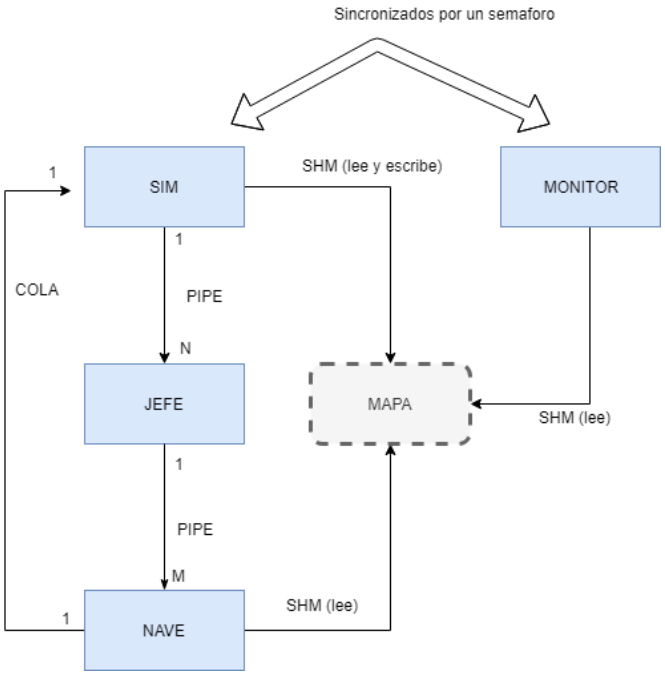
```
→ Proyecto git:(master) ✗
.
├── bin
│   ├── launcher
│   ├── monitor
│   └── style_test
├── Makefile
├── README.md
├── src
│   ├── gamescreen.c
│   ├── gamescreen.h
│   ├── info_nave.c
│   ├── info_nave.h
│   ├── jefe.c
│   ├── jefe.h
│   ├── launcher.c
│   ├── Makefile
│   ├── mapa.c
│   ├── mapa.h
│   ├── monitor.c
│   ├── msg.c
│   ├── msg.h
│   ├── nave.c
│   ├── nave.h
│   ├── simulador.c
│   ├── simulador.h
│   └── types.h
└── tests
    ├── Makefile
    ├── sem_test.c
    └── style_test.c
```

Los ficheros: **nave.c**, **jefe.c** y **sim.c** corresponden a los diferentes procesos. Por otra parte, los ficheros **launcher.c** y **monitor.c** son los que contienen las rutinas principales, (main), de ejecución.

El fichero **msg.c** contiene la implementación de un sistema de mensajes de log por terminal basado en etiquetas. (Las declaraciones se encuentran en **msg.h**).

Además, **info_nave.c**, que corresponde a la estructura para almacenar la información de las naves en el mapa (hemos modificado la estructura **tipo_mapa**).

Jerarquía de procesos



El anterior diagrama es un esquema de la organización básica de los procesos. Cabe destacar, que, además de eso, nosotros implementamos un semáforo naves_ready para avisar al simulador del momento en que las naves han completado su inicialización.

En dicho diagrama se puede observar que, a partir de un proceso simulador, se lanzarán varios procesos hijo correspondientes a los jefes de cada uno de los N_EQUIPOS. Por otra parte, cada uno de estos procesos jefe, lanzará a su vez M procesos nave (N_NAVES).

De forma paralela, e independiente al orden de ejecución, cuando el simulador finalice su inicialización, (semáforo SIM – MONITOR), se ejecutará un monitor en otra terminal que mostrará en tiempo real el desarrollo de la simulación.

Acerca de types.h y las estructuras de proceso

En el fichero types.h se han incluido definiciones de forma pública, por razones de comodidad, de todas las estructuras empleadas en la práctica.

Cabe destacar, sobre todo, la estructura "info_nave", que almacena los atributos de posición, daño, vida, etc, de las naves en el mapa, esta nueva estructura será la que se encuentre almacenada en tipo_mapa en la memoria compartida:

```
// Contiene los datos de una nave (ver tipo_mapa)
typedef struct {
    int vida;       // Vida que le queda a la nave
    int posx;       // Columna en el mapa
    int posy;       // Fila en el mapa
    int equipo;
    int num;
    int dmg;        // El daño que inflinge
    int alcance;
} info_nave;
```

En el caso de las estructuras tipo_nave, tipo_jefe y tipo_sim, contienen los recursos empleados por el proceso, en el ámbito de la comunicación interproceso.

```
/*** PROCESO NAVE ***/
// Información del PROCESO nave
typedef struct {

    int equipo;    // Equipo de la nave
    int num;
    int * pipe_jefe;
    char tag[TAG_MAX];
    mqd_t cola_sim;
    tipo_mapa * mapa;

} tipo_nave;
```

```
/*** PROCESO JEFE ***/
// Información del PROCESO jefe
typedef struct {
    int equipo;     // equipo del jefe (id)

    int pipes_naves[N_NAVES][2];
    int * pipe_sim;
    char tag[TAG_MAX];
    int pid_naves[N_NAVES];
    int naves_res;  // naves restantes
    bool naves_vivas[N_NAVES];
} tipo_jefe;
```

```c
/*** PROCESO SIM ***/
// Información del PROCESO simulador
typedef struct {
    int pipes_jefes[N_EQUIPOS][2];
    char tag[TAG_MAX];
    bool equipos_vivos[N_EQUIPOS];
    int equipos_res;
    tipo_mapa *mapa;
    mqd_t cola_msg_naves;
    sem_t *sem_naves_ready;
    sem_t *sem_sim;     // semaforo para avisar al monitor
    //int * readers_count;

} tipo_sim;
```

Además, se han incluido definiciones de diferentes cadenas de texto y enumeraciones para su empleo en el envió y la recepción de mensajes.

Sobre la estructura de los procesos

Siguiendo la división en modulos del proyecto, se ha tratado de abstraer la mayor parte de la funcionalidad de cada tipo de proceso. De esta forma todos los procesos constan de una función **launch()**, que servirá como rutina principal, y que a su vez, será la encargada de llamar a la secuencia de acciones del proceso.

Asimismo, cada proceso contiene su funcionalidad dividida en diferentes métodos, que, pese a ser de uso interno, se han declarado en el fichero de headers para mayor claridad. En particular, cabe destacar las siguientes funciones, que se encuentran presente en todos los tipos de procesos del simulador, y que son invocadas por **launch().**

create(): Crea la estructura que contendrá los recursos empleados en la comunicación de procesos.

init(): Inicializa los recursos empleados en la comunicación de procesos.

run(): Método que contiene la parte “principal” del código del proceso, es decir, el bucle de ejecución.

end(): Se encarga de la liberación de recursos, y espera a procesos hijos si los hubiera.

destroy(): Libera la memoria necesaria, y finaliza la ejecución. En el caso del simulador, también elimina los recursos del SO.

Además, se encuentran presentes otras funciones como **init_X_RECURSO(),** que inicializarán un recurso en particular, o **actua()**, encargada de reaccionar ante los mensajes obtenidos.

Explicación de la ejecución principal de los procesos

Proceso simulador

En primer lugar, se crea e inicializa una estructura de tipo tipo_sim, así como todos los recursos necesarios: pipes a jefes, cola de naves, memoria compartida para el mapa, semáforos, y manejadores de señal.

Después, ejecuta los jefes, y se pone a la espera del semáforo naves_ready.

Una vez este semáforo se levanta, comienza su ejecución, en primer lugar, establece una alarma periódica, tras esto, entra en su bucle principal de ejecución, en el que espera a la recepción de mensajes de las naves, los evalúa y actúa en consecuencia hasta que hay un equipo ganador.

Una vez se finaliza el bucle, se inhabilitan las alarmas, y se comienza con la fase de finalización.

Primero, el proceso espera a que los procesos jefes finalicen, y tras esto libera todos los recursos, y destruye en la estructura tipo_sim.

Determinadas acciones recibidas por parte de las naves, MOVER y ATACAR requerirán hacer uso de la memoria del mapa, que será creada como memoria compartida con permisos de escritura y lectura.

Proceso jefe

Al igual que el simulador, comienza por crear una estructura de tipo tipo_jefe, e inicializar los recursos pertinentes, es decir, crear una pipe de salida a cada nave, y abrir la pipe de entrada del simulador.

Una vez hecho esto, se inicia el bucle de ejecución, en el que se espera a la llegada de mensajes de nave, se procesan, y se envían mensajes a las naves.

Cuando recibe el mensaje FIN del simulador, se envía una señal SIGTERM a cada nave, y se espera a que estas finalicen su ejecución, para, posteriormente liberar los recursos almacenados, y, el también, terminar su ejecución.

Proceso nave

El proceso nave es similar al proceso jefe, en el sentido en que recibe mensajes de su padre (jefe), los procesa, y envía una respuesta al simulador. Sin embargo, a diferencia de los anteriores, que realizaban el envió mediante pipes, el proceso nave emplea una cola bloqueante.

Una particularidad es que, a la hora de procesar ciertas acciones, como ATACAR, será necesario conocer la posición del resto de naves en el mapa, por ello, se cargará esta estructura en memoria compartida, pero solo con permisos de lectura.

Sobre el manejo de excepciones

Existen 3 tipos de excepciones a manejar:

SIGINT, por el proceso simulador, al recibirla, deberá enviar un mensaje “FIN” a los procesos jefe vivos, y estos a su vez, enviarán SIGTERM a las naves vivas.

SIGTERM, por los procesos nave, al recibirla liberarán sus recursos y finalizarán.

SIGALRM, por el proceso simulador, al recibirla periódicamente, se entenderá como la llegada de un nuevo turno, y se mandará el mensaje “TURNO” a los jefes, para que estos permitan actuar, a su vez, a las naves.

Cabe destacar, de cara a los manejadores de señales, que a menudo, interrumpen cualquier tipo de espera (pipes, colas, semáforos…), sin embargo, mediante el uso de la flag SA_RESTART, podemos hacer que dichas esperas se reanuden. En nuestra implementación esto solo es necesario para el caso de SIGALRM, y, además, resta mencionar que dicha flag no funciona con las operaciones de sem_wait(). Sin embargo, dado que los semáforos que empleamos se utilizan solo de forma previa a la recepción de dicha señal, esto no es relevante.

Sobre las cadenas de mensaje

A continuación, se detallan las cadenas de mensaje recibidas y enviadas por cada proceso:

Simulador:

Envía:

1. TURNO
2. DESTRUIR TAG_NAVE
3. FIN

Recibe:

1. ACCION ATACAR TAG_NAVE_ATACANTE COORDENADAS_ATAQUE
2. ACCCION MOVER TAG_NAVE

Jefe:

Envía:

1. ATACAR
2. MOVER_ALEATORIO
3. DESTRUIR

Recibe:

1. TURNO
2. DESTRUIR TAG_NAVE
3. FIN

Nave:

Envía:

1. ACCION ATACAR TAG_NAVE_ATACANTE COORDENADAS_ATAQUE
2. ACCCION MOVER TAG_NAVE

Recibe:

1. ATACAR
2. MOVER_ALEATORIO

Donde **<TAG_NAVE>** tiene el formato: “NAVE E:_/N:_” y **<COORDENADA>:** X:__/Y:__.

Los mensajes son divididos en fragmentos mediante “dividir_accion()” y “dividir_msg()”. Asimismo, a la hora de extraer valores de tags o coordenadas, se emplearán las funciones “extractv_nave_tag()” y “extractv_coordenadas()”.

Acerca de la inicialización de las posiciones de las naves en el tablero:

Mediante la función de mapa.c “generate_pos_nave()”, se generan posiciones semialeatorias de inicio para cada nave, esta función divide el mapa en 4 cuadrantes, correspondientes a, hasta un máximo de 4 equipos (MAX_EQUIPOS), y a su vez, cada uno de esos cuadrantes, es dividido de nuevo en otros 4, correspondientes, cada uno a una de las naves del equipo. De esta forma se asegura que no haya naves superpuestas.

Sobre launcher.c, argumentos y mensajes de terminal

Hemos creado el fichero launcher.c que recoge la funcionalidad de lectura de argumentos de ejecución mediante la librería **getopt**, y el formato de salida.

Los argumentos de ejecución son almacenados en una variable global de tipo “tipo_argumentos”, y, en el caso de haber un fichero de log, este se abre, también de forma global en un puntero “FILE * fpo”.

Para esto último, se ha creado el módulo **msg.c** que agrupa la funcionalidad del sistema de colores y tags. En **msg.h** se han definido una serie de macros de cara al formato de salida. En función de los argumentos de ejecución, la estructura global “tipo_estiloMSG estiloMSG” se rellena con tags “a color” o tags “sin color”.

(ver funciones: launcher.c: “set_default_params()”, “leer_argumentos()” y msg.c “estiloMSG_set_default()”, “estiloMSG_set_colorful()”).

```c
typedef struct {
    // messages
    char std_msg[STYLE_STRING_L];
    char status_msg[STYLE_STRING_L];
    // status
    char ok_status[STYLE_STRING_L];
    char err_status[STYLE_STRING_L];
    // tags
    char turno_tag[TAG_MAX];
    char nave_tag[TAG_MAX];
    char jefe_tag[TAG_MAX];
    char sim_tag[TAG_MAX];
    int tag_offset;
} tipo_estiloMSG;
```

Consideraciones adicionales de la implementación:

1. Dado que toda la memoria es clonada al realizar una llamada a fork, se deben liberar la memoria reservada nada más comenzar la ejecución del proceso hijo. De este mismo modo, es conveniente restaurar los manejadores de señal con la funcion “signal()”
2. Mientras se realiza la liberación de memoria final, el manejador de SIGINT ha de ser desactivado para evitar errores.
3. Se han modificado, para agilizar la ejecución, los valores los predefinidos de duración del turno, así como la tasa de refresco del monitor y la espera tras cada acción, no obstante, estos valores pueden ser revertidos si fuera necesario.
4. Se consideró implementar un algoritmo de lectores-escritores con prioridad a escritores de cara al acceso a la memoria compartida del mapa, sin embargo, por falta de tiempo no se llegó a implementar. (No es mucho problema dado que solo hay un escritor, y “nmap” es capaz de avisar de actualizaciones en la memoria al resto de procesos según su inicialización).
5. Se ha incluido un breve comentario del objetivo de cada método en los ficheros de cabeceras.
6. **Finalmente**, cabe añadir, que por algún motivo desconocido, la función “mapa_send_misil()” provoca errores en la ejecución (las naves no finalizan correctamente), por lo que se ha omitido su uso. Se incluye comentada en la línea 425 de simulador.c).

Ejecución:

La compilación del proyecto se realizará mediante “make -B”, “make all”, o simplemente “make”.

Se adjunta un test del estilo de mensajes en el directorio “tests”, que se ejecutará mediante: **“./tests/style_test”.**

Para la ejecución del proyecto se empelarán los comandos:

“./bin/monitor”

“./bin/launcher”, con la posibilidad de añadir las flags -f <fichero_log> y -c *(color).* Se recomienda no mezclar la ejecución en fichero con la ejecución con color.

Ejemplos:
./bin/launcher -c

./bin/launcher -f log.out

Se recomienda la herramienta grep para comprobar la correcta finalización de los hijos y similares. De esta forma, se podrá emplear, por ejemplo: “… | grep -i -e destr”

Nota: En caso de haber algún problema con los colores de la terminal, ejecutar sin la flag -c.

Se ha comprobado mediante la herramienta Valgrind, la ausencia de errores de memoria, a excepción de unos pocos allocs no liberados en los procesos nave.

Ejemplo de ejecución: