Funcionalidades adicionales práctica de ESO (curso 2025-26)

progP

```
int GARLIC_send(unsigned char n, int data)
int GARLIC_receive(unsigned char n)
```

Fase 1

- Implementar un sistema de 8 buzones globales, identificados con números del 0 al 7.
- Cada buzón dispondrá de una cola de datos interna que le permitirá almacenar hasta 16 valores de tipo `int`.
- Inicialmente todos los buzones estarán vacíos.
- Cualquier proceso del sistema operativo podrá invocar a las funciones del API `GARLIC_send()` y `GARLIC_receive()`, indicando con el parámetro `n` el identificador del buzón que quiere utilizar.
- Además, la función `GARLIC_send()` permitirá enviar un valor numérico al buzón correspondiente mediante el parámetro `data`.
- El funcionamiento de los buzones será el siguiente:
    - si un proceso invoca a `GARLIC_send()` sobre un buzón que NO esté lleno (que todavía no contenga 16 datos), el dato enviado quedará registrado en la última posición del buzón y la función retornará un valor diferente de 0;
    - si un proceso invoca a `GARLIC_send()` sobre un buzón que esté lleno, el contenido del buzón no se modificará y la función retornará un 0;
    - si un proceso A invoca a `GARLIC_receive()` sobre un buzón que esté vacío, dicho proceso quedará bloqueado hasta que otro proceso B invoque a `GARLIC_send()` sobre el mismo buzón; cuando el proceso A se desbloquee, la función `GARLIC_receive()` devolverá el dato que el proceso B envió al buzón, sin que este dato quede registrado en dicho buzón;
    - si un proceso invoca a `GARLIC_receive()` sobre un buzón que NO esté vacío, la función extraerá el dato registrado en la primera posición del buzón y lo devolverá como resultado de la llamada, sin bloquear al proceso invocador;
- Por lo tanto, la función `GARLIC_receive()` tiene un comportamiento síncrono (se bloquea mientras el buzón esté vacío), en cambio la función `GARLIC_send()` tiene un comportamiento asíncrono (no se bloquea aunque el buzón esté lleno).
- En la fase 1, la implementación del bloqueo de procesos se puede hacer dentro de la rutina de sistema que gestiona el `GARLIC_receive()`, con un bucle de consulta periódica del estado de la cola de datos del buzón.
- Si existen varios procesos bloqueados sobre un mismo buzón, en la fase 1 de la práctica NO será necesario garantizar el mismo orden de desbloqueo que el orden en que se han bloqueado.

Fase 2

- En la fase 2 se requiere que las llamadas a la función `GARLIC_receive()` que sean bloqueantes provoquen que el proceso invocador salga de la cola de *Ready*, de modo que ya no sea necesario ningún bucle de consulta periódica; la llamada pertinente a la función `GARLIC_send()` pondrá al proceso a desbloquear (si lo hay) otra vez en la cola de *Ready*.
- Para gestionar el bloqueo de procesos se puede usar la misma técnica que para retardar procesos, creando una cola de procesos bloqueados en cada uno de los buzones.
- En esta segunda fase, el orden de desbloqueo de los procesos sí tiene que ser el mismo que el orden de bloqueo (FIFO).

progM

```
void * GARLIC_malloc(unsigned short size)
int GARLIC_free(void * addr)
```

Fase 1

- Implementar las funciones `GARLIC_malloc()` y `GARLIC_free()`, para permitir que los procesos de usuario puedan reservar y liberar trozos de memoria dinámica durante su ejecución.
- Cada proceso de usuario podrá tener reservados simultáneamente hasta 4 trozos de memoria dinámica.
- La función `GARLIC_malloc()` admite como parámetro el número de bytes a reservar de un trozo de memoria dinámica; si hay suficiente memoria dinámica consecutiva y no se ha llegado al máximo de trozos reservados por el proceso invocador, la función devolverá la dirección de memoria inicial del nuevo trozo reservado; en caso contrario, la función retornará un 0.
- La función `GARLIC_free()` admite como parámetro la dirección inicial de memoria del trozo a liberar; si dicha dirección coincide con la dirección inicial de alguno de los trozos que el proceso invocador tiene reservados actualmente, se liberará dicho trozo y la función retornará un valor diferente de 0; en caso contrario, la función retornará un 0.
- En la fase 1, estas funciones pueden llamar internamente a las funciones `malloc()` y `free()` de la librería ***libc***, de modo que la memoria dinámica asignada a cada trozo será gestionada directamente por dichas funciones de librería.

Fase 2

- En la fase 2 se deberán usar franjas de la memoria de programas de usuario para gestionar los trozos de memoria dinámica, de modo que la reserva de dichos trozos afectará al espacio de memoria disponible para cargar los segmentos de código y datos de nuevos procesos de usuario. Por lo tanto, la gestión de memoria dinámica deberá interactuar con la estructura de datos `gm_zocmem[]`, descrita en el apartado 3.3 del manual de la fase 2 de la práctica.
- Otra consecuencia será que los trozos reservados deberán tener asignados un número entero de franjas (de 32 bytes), de modo que seguramente se generará fragmentación interna en cada trozo.
- En el apartado 3.4 del manual de la fase 2 se indica cómo llamar a la rutina `_gs_pintarFranjas()` (ya implementada en el fichero `garlic_itcm_sys.s`) para obtener una visualización gráfica de la ocupación de memoria de los procesos de usuario. Sin embargo, esta rutina no está preparada para representar las franjas ocupadas por la memoria dinámica.
- Como parte de las funcionalidades adicionales se pide implementar una nueva rutina `_gm_pintarFranjas()` (en el fichero `garlic_itcm_mem.s`) que permita distinguir visualmente qué franjas están ocupadas por memoria dinámica. Para ello se sugiere la siguiente representación gráfica:
    - para mostrar los trozos de memoria asignados a segmentos de código o a segmentos de datos, llamar a la rutina `_gs_pintarFranjas()`, la cual permite visualizar una banda de color sólido para un segmento de código o un banda con un patrón ajedrezado para un segmento de datos, con un color específico para cada proceso de usuario; estas bandas se ubican dentro de tres áreas delimitadas por dos líneas blancas que dejan un espacio de 4 píxeles de altura, y donde cada franja de memoria ocupada se corresponde con un píxel de anchura de la banda correspondiente;
    - para mostrar los trozos de memoria dinámica, la nueva rutina `_gm_pintarFranjas()` deberá modificar los 4 píxeles de cada franja

asignada a cada trozo, marcando los píxeles superior e inferior de la banda en color gris y los dos píxeles centrales de la banda en el color específico del proceso de usuario que ha reservado ese trozo;

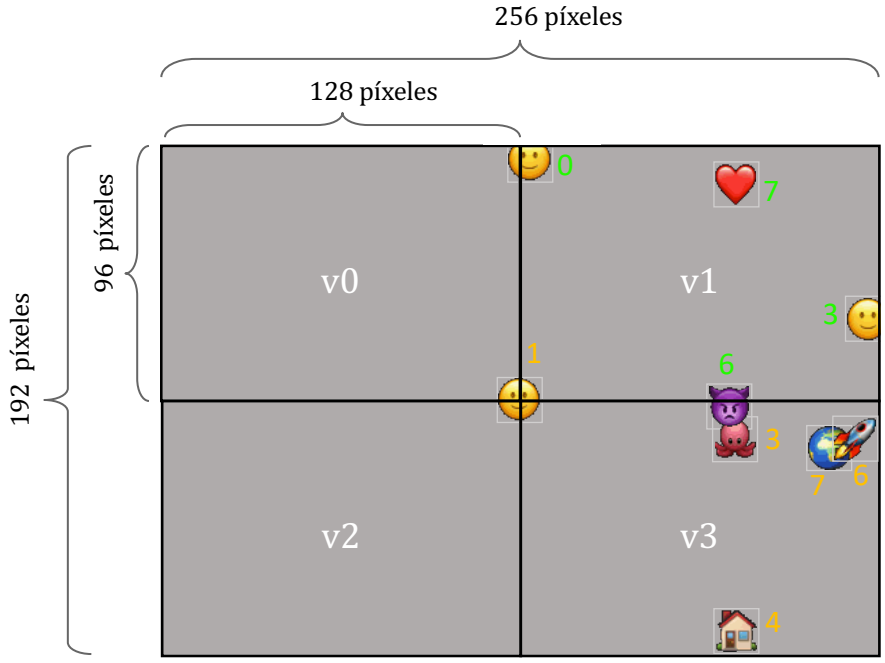
- además, para poder distinguir trozos de memoria dinámica que se hayan asignado de manera consecutiva al mismo proceso de usuario, la primera franja de cada trozo se deberá pintar con el patrón de colores invertido, es decir, los píxeles superior e inferior de la banda en el color específico del proceso de usuario y los dos píxeles centrales en color gris;
- a continuación se muestra un ejemplo de representación gráfica, donde se pueden observar 3 procesos de usuario (rojo, verde y amarillo), con un segmento de código para cada proceso, un segmento de datos para 2 de los procesos (verde y amarillo), 4 trozos de memoria dinámica para el proceso rojo y 2 trozos de memoria dinámica para el proceso verde:

- en el ejemplo anterior, la primera banda de memoria dinámica del proceso rojo que se muestra en el área central es significativamente grande (56 franjas de longitud, 1792 bytes de capacidad), pero las otras 3 bandas (área inferior) son muy pequeñas, ya que corresponden a trozos de 5, 3 y 1 franja, respectivamente, pero que se han asignado consecutivamente; el ejemplo está diseñado expresamente para mostrar la capacidad de distinguir bandas de memoria dinámica consecutivas asignadas al mismo proceso;
- además, la primera de las dos bandas de memoria dinámica asignada al proceso verde empieza en el área central (justo después del segmento de datos del proceso amarillo) pero continua al principio del área inferior, puesto que las tres áreas en realidad representan un espacio contiguo de memoria; el hecho de que, al inicio del área inferior, el patrón de la primera franja sea gris-verde-gris y no verde-gris-verde es una indicación clara de que se trata de la continuación de la banda que empieza en el área central; por lo tanto, podemos deducir que el proceso verde en este momento solo tiene 2 trozos de memoria dinámica reservados.

```
void GARLIC_spriteSet(unsigned char n, unsigned char icon)
void GARLIC_spriteMove(unsigned char n, short px, short py)
void GARLIC_spriteShow(unsigned char n)
void GARLIC_spriteHide(unsigned char n)
```

Fase 1

- Implementar una nueva función del API de GarlicOS, `GARLIC_spriteSet()`, que permitirá a los procesos definir *sprites* de 32x32 píxeles:
  - cada proceso podrá definir su propio conjunto de 8 *sprites*,
  - el parámetro `n` indicará el número de *sprite*, entre 0 y 7,
  - el parámetro `icon` será un número de icono entre 0 y 63, que corresponderá al dibujo que debe tener asociado ese *sprite* (ver los iconos disponibles más adelante).
- Implementar las nuevas funciones del API de GarlicOS, `GARLIC_spriteMove()`, `GARLIC_spriteShow()`, `GARLIC_spriteHide()`, que permitirán a los procesos mover, mostrar y ocultar el *sprite* indicado con el parámetro `n`.
- Inicialmente todos los *sprites* estarán ocultos.
- Los *sprites* se podrán posicionar mediante los parámetros `px` y `py` de la función `GARLIC_spriteMove()`, que indicarán las coordenadas del píxel superior-izquierdo del *sprite* a mover, de manera relativa a la posición de la ventana donde se ejecuta el proceso invocador; los rangos permitidos serán los siguientes:
  - `px`: [-32..256]
  - `py`: [-32..192]
- Las coordenadas relativas de ventana se deberán convertir a coordenadas absolutas de pantalla, de modo similar a la ubicación de caracteres escritos en cada ventana (ver apartado 6.3 del manual de la fase 1), aunque los cálculos serán diferentes.
- A continuación se muestra un ejemplo de visualización de algunos *sprites* creados en las ventanas v1 y v3, para la configuración de pantalla de la fase 1 (solo 4 ventanas); hay que tener en cuenta que se han usado diversos elementos gráficos con fines explicativos, pero que en la visualización final no serán visibles, como los marcos y los números de los *sprites*, las dimensiones y los identificadores de las ventanas:



- Las siguientes tablas comentan las diferentes situaciones de los *sprites* del ejemplo anterior, para dos ventanas:

  - proceso en ventana v1 (en el esquema, los números de *sprite* están en verde):

Sprite	Coord.	Observaciones
0	(-10, -10)	una parte superior del *sprite* queda recortada por la pantalla, pero una parte izquierda invade el espacio de la ventana v0: idealmente, esta parte que sale por la izquierda también se debería recortar, pero este requisito complicaría enormemente la implementación de la visualización de los *sprites*, por lo tanto, se permitirá la "invasión" del espacio de ventanas colindantes
3	(240, 135)	una parte derecha del *sprite* queda recortada por el límite de la pantalla
6	(110, 182)	una parte inferior del *sprite* invade el espacio de la ventana v3
7	(150, 20)	está íntegramente dentro del espacio de la ventana v1

  - proceso en ventana v3 (en el esquema, los números de *sprite* están en marrón):

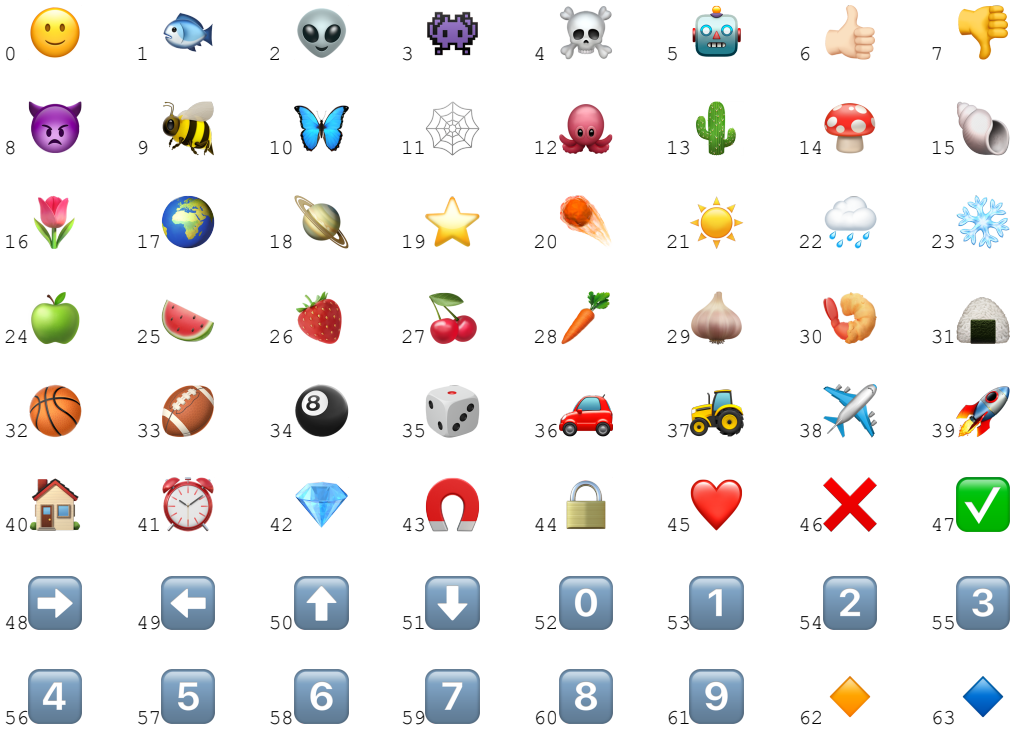
Sprite	Coord.	Observaciones
1	(-16, -16)	está posicionado en coordenadas negativas, de manera que está ocupando espacio de las cuatro ventanas
3	(150, 20)	está posicionado en las mismas coordenadas que el *sprite* 7 del proceso de la ventana v1, pero obviamente sus posiciones en pantalla son diferentes; además, hay un poco de colisión con el *sprite* 6 del proceso de la ventana v1, pero esta situación se considera normal (no hay que hacer nada para evitarlo)
4	(150, 159)	se encuentra en el límite vertical para no salir por debajo del espacio de la ventana
6	(223, 16)	se encuentra en el límite horizontal para no salir por la derecha del espacio de la ventana
7	(218, 20)	se solapa con el *sprite* 6, pero esta situación se considera normal.

- Los *sprites* se deberán visualizar encima del texto (baldosas del fondo 2) pero debajo de los bordes de las ventanas (baldosas del fondo 3).
- En la fase 1, los *sprites* se deberán visualizar a una escala del 50%, puesto que las ventanas están reducidas en esta proporción: la transformación de coordenadas relativas a coordenadas absolutas de pantalla deberá tener en cuenta esta reducción de tamaño.

Fase 2

- En la fase 2, la posición y escalado de las 16 ventanas se puede variar mediante los controles de interfaz de usuario; esto significa que,
  - para cada cambio de escalado de la pantalla será necesario cambiar el escalado de los *sprites*,
  - para cada cambio de posición de las ventanas será necesario recalcular la posición en pantalla de todos los *sprites*, ocultando aquellos *sprites* que queden totalmente fuera de las coordenadas visibles de pantalla.
- El ajuste de posición y escalado de la pantalla ser realiza de forma dinámica desde la rutina `_gi_controlInterfaz()` (ya implementada en `garlic_itcm_ui.s`); para simplificar la implementación de la funcionalidad adicional, NO se exige que el ajuste de escalado y posición de los *sprites* se realice de forma síncrona con este proceso dinámico, sino que simplemente se deberán ocultar todos los *sprites* antes del ajuste, y visualizar de nuevo los *sprites* pertinentes en su nueva posición y escalado, una vez haya terminado el ajuste.

En la sección de prácticas del espacio Moodle de la asignatura se proporciona un fichero llamado `Sprite_icons.gif`, el cual incluye los siguientes dibujos (selección de emoticonos del sistema MacOS®):



Para convertir el fichero de gráficos en ficheros de código fuente, desde un intérprete de comandos se puede invocar al programa `grit` con los siguientes argumentos:

```
$ grit Sprite_icons.gif -gt -gzl -gB8 -Mh4 -Mw4 -pn 256 -pT 3 -oicons
```

Si la ejecución funciona correctamente, se generarán dos ficheros, `icons.h` y `icons.s`, con las definiciones numéricas de las baldosas gráficas `iconsTiles[]` y la paleta correspondiente `iconsPal[]`. Estos ficheros se deberán incorporar a los directorios `include` y `source` del proyecto GARLIC_OS.

También es muy recomendable adjuntar los ficheros proporcionados en la práctica de Computadores `Sprites_sopo.h` y `Sprites_sopo.s` , que permiten gestionar los *sprites* mediante una serie de rutinas ya implementadas, aunque el programador `progG` deberá adaptar las llamadas para conseguir *sprites* locales a cada proceso y a su ventana asociada.

progT

Redirección de entrada desde buzón

Fase 1

- Añadir una nueva funcionalidad al procedimiento de carga de procesos de GarlicOS que permita redireccionar la entrada de texto de un proceso de usuario:
    - después de preguntar el programa a cargar y su argumento, si el programa seleccionado por el usuario contiene llamadas a `GARLIC_getstring()` se presentará un tercer menú para que el usuario indique uno de los 8 buzones globales, identificados por un número del 0 al 7, así como la opción "`keyboard`" que permitirá introducir el texto por teclado según el procedimiento básico descrito en los manuales de prácticas,
    - los nombres de buzón empezarán por la palabra "`mb`" seguida de un dígito decimal: "`mb0`", "`mb1`", "`mb2`", …, "`mb7`",
    - en caso de seleccionar uno de los buzones, se registrará la redirección para que la función `GARLIC_getstring()` proceda según dicha selección,
    - cuando un proceso de usuario llame a `GARLIC_getstring()` sobre un buzón, esta función deberá leer un *string* del buzón seleccionado:
        - el *string* se compone de una serie de caracteres ASCII (valores de tipo `int`) enviados uno a uno al buzón, que se acaban con el carácter de salto de línea ('`\n`') o con un carácter centinela de final de *string* (`0`),
        - la función `GARLIC_getstring()` irá extrayendo los caracteres del buzón uno a uno;
        - si en el proceso de extracción de caracteres del *string* el buzón se queda vacío (temporalmente), la función `GARLIC_getstring()` se quedará bloqueada a la espera de que otro proceso envíe más caracteres sobre el buzón correspondiente,
        - el proceso de lectura del *string* termina cuando se recibe un carácter de salto de línea o un carácter centinela,
        - cuando el *string* termina con un carácter de salto de línea, la rutina de sistema que gestiona la llamada a `GARLIC_getstring()` añadirá un carácter centinela a continuación,
        - los caracteres leídos se devolverán por referencia sobre el parámetro `string` de la función `GARLIC_getstring()`, y el resultado de la función será el número total de caracteres leídos finalmente (excluido el centinela).
    - hay que tener en cuenta que la longitud del mensaje enviado por otro proceso no tiene por qué coincidir con la longitud del mensaje esperado: se leerán caracteres hasta el final del *string* (salto de línea o centinela) o hasta que se llegue al número máximo de caracteres indicado por el parámetro `max_char` de la función `GARLIC_getstring()`,
    - cuando la función `GARLIC_getstring()` esté leyendo información de un buzón, NO será necesario mostrar ninguna interfaz de teclado, ni realizar ningún tipo de sincronización, puesto que la llamada será síncrona,
    - la implementación del sistema de buzones en la fase 1 será la que implemente el programador progP en esta fase, es decir, usando un bucle de encuesta dentro de la implementación del mecanismo de recepción de mensajes.

Fase 2

- Adaptar la implementación de la fase 1 de `GARLIC_getstring()` para usar la nueva implementación del sistema de buzones de la fase 2 que realice progP en esta fase, es decir, usando las colas de bloqueo de procesos para cada buzón.

- Además, cuando un proceso de usuario esté esperando un mensaje de un buzón, en la columna de la tabla de control que indica el estado de cada proceso se deberá mostrar el número identificador de dicho buzón, en vez de la letra 'K' que se utilizará cuando el proceso esté esperando un mensaje desde la interfaz de teclado.