Manual de Prácticas de

Estructura de Sistemas Operativos (ESO)

Fase 2

GarlicOS (versión 2.0)

Sistema Operativo para NDS

Departament d'Enginyeria Informàtica i Matemàtiques

Universitat Rovira i Virgili

santiago.romani@urv.cat

Índice de contenido

1 Introducción a la fase 2

1.1 Preparación de las ramas de trabajo

Recordemos que, al finalizar la fase 1, los últimos *commits* de las ramas de trabajo de los programadores se deben fusionar con la rama `master` para conseguir la versión definitiva **GarlicOS 1.0**.

El esquema que se proponía en el manual de la fase 1 de la práctica como ejemplo de evolución de *commits* finales era el siguiente:

En el espacio *Moodle* de la asignatura se proporciona un fichero comprimido de nombre **`Garlic_OS_2.git.zip`**, que contiene el código fuente de soporte para la realización de esta segunda fase de la práctica. **Es un repositorio git** que contiene un *commit* inicial para la rama `master` más un *commit* específico para cada rol (excepto `progT`), de modo que será necesario utilizar el comando `git checkout` para acceder al contenido específico de cada rama.

En la rama `master` de `Garlic_OS_2.git` se encuentran todos los ficheros comunes para la fase 2. Por lo tanto, habrá que sustituir la versión antigua de estos ficheros comunes sobre el último *commit* de la fase 1, y crear un nuevo *commit* de partida para todos los roles. Este proceso **solo se tiene que realizar una vez sobre la rama `master`**, y después subir el nuevo *commit* en el servidor para que todos los programadores del

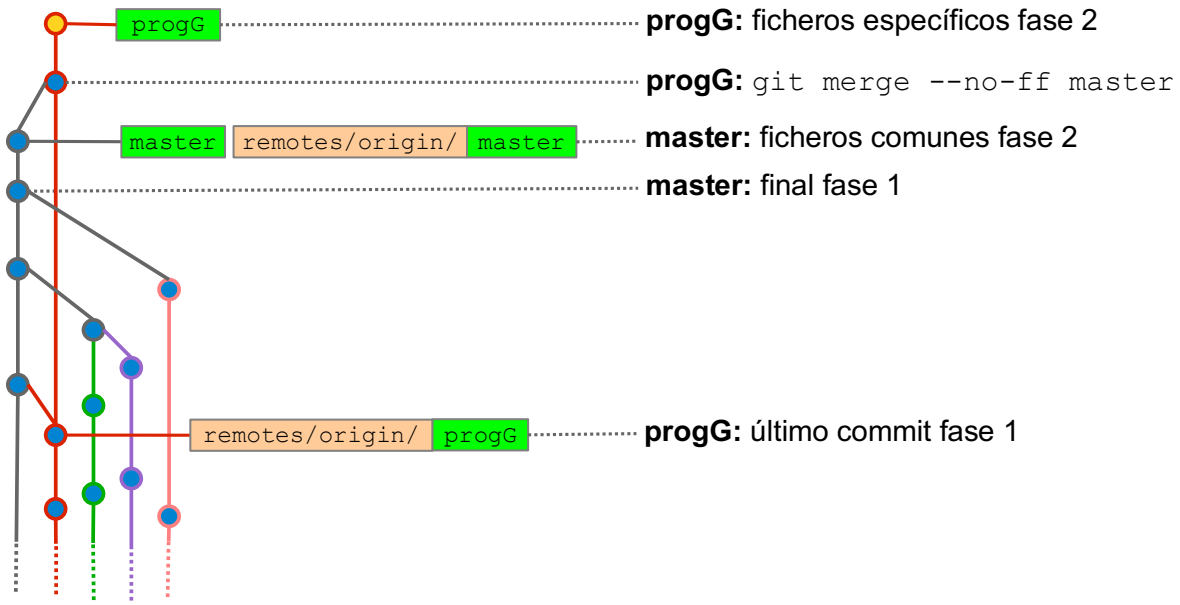
grupo de prácticas se lo puedan bajar a la rama `master` de sus repositorios locales con los siguientes comandos (ejemplo para el programador `progG`):

```
(progG)$ git checkout master
(master)$ git pull origin master
```

Posteriormente, hay que avanzar la rama de trabajo de cada programador a partir de este nuevo *commit* del `master`, con los siguientes comandos:

```
(master)$ git checkout progG
(progG)$ git merge --no-ff master
```

A continuación, cada programador debe modificar los ficheros específicos de su rama según el contenido de su *commit* específico en `Garlic_OS_2.git` y crear un nuevo *commit* con dichos cambios. La evolución de los nuevos *commits* para empezar la fase 2 se puede ejemplificar con el siguiente gráfico:

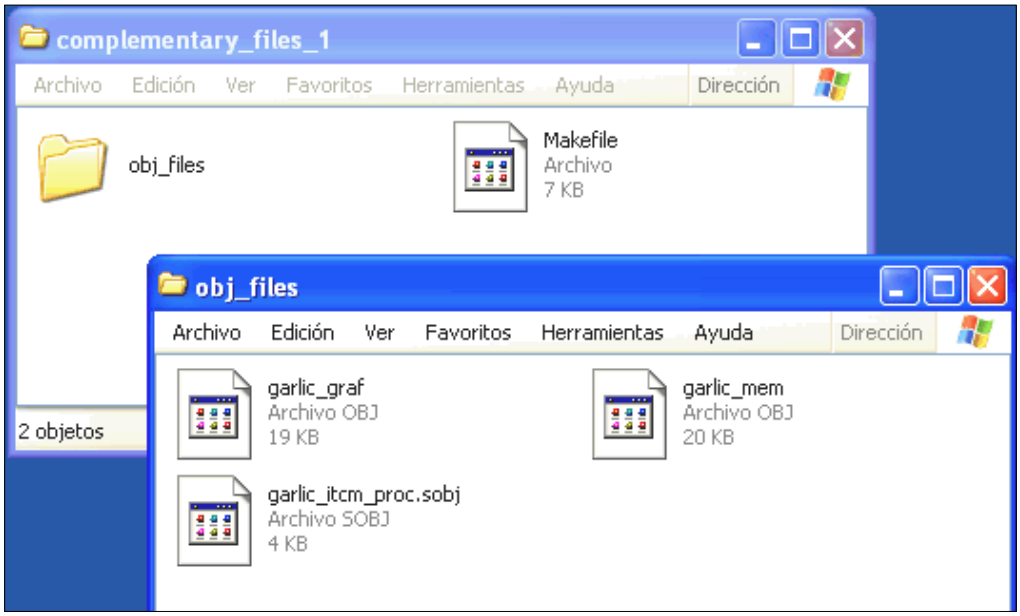


Nota: en este gráfico, los *commits* nuevos de la rama `progG` se representan sobre la misma línea vertical que los *commits* antiguos de dicha rama, para ofrecer una distinción más clara de la evolución de cada rama, pero hay que tener en cuenta que el **gitk** no aplicará este mismo criterio en la visualización de los distintos *commits* y sus líneas de conexión.

Por último, advertir que dentro del repositorio `Garlic_OS_2.git` hay algunos ficheros nuevos y otros antiguos actualizados. Hay que tener cuidado con algunos ficheros que contienen código fuente que se tiene que añadir al código realizado de los ficheros de la fase 1. Por ejemplo, el fichero `garlic_itcm_proc.s` contiene código nuevo para la rutina `_gp_terminarProc()`, o sea, que habrá que sustituir el código anterior de dicha rutina, pero **no hay que sustituir todo el fichero**, puesto que las rutinas como `_gp_crearProc()` están en blanco.

1.2 Ficheros objeto complementarios de la fase 1

En el espacio *Moodle* de la asignatura se proporciona otro fichero comprimido de nombre `complementary_files_1.zip`, que contiene una nueva versión del fichero `Makefile` junto con un directorio con los **ficheros objeto** correspondientes al código máquina de las rutinas que tienen que implementar los programadores `progP`, `progM` y `progG` para la fase 1 de la práctica:



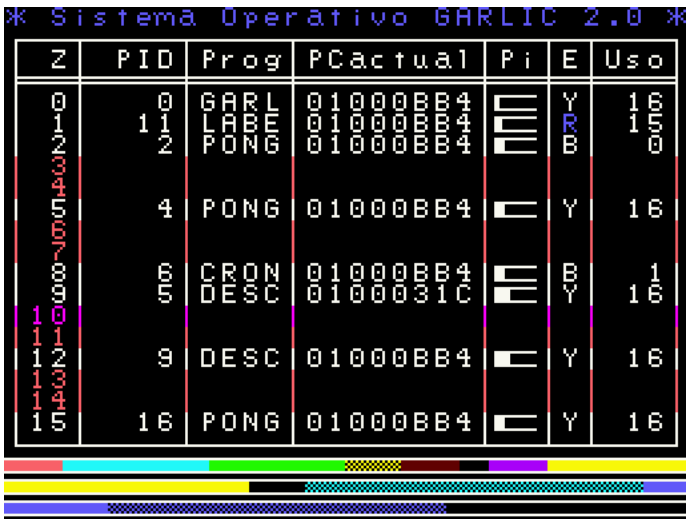
El propósito de estos ficheros es complementar los proyectos GarlicOS de los grupos de prácticas que no han presentado alguno de estos roles en la fase 1. Por lo tanto, estos grupos deberán actualizar el fichero `Makefile` de GarlicOS y añadir el directorio `obj_files` con **solo los ficheros objeto de los roles que les falten** en su grupo de prácticas. Por ejemplo, si un grupo de 2 alumnos realiza los roles `progM` y `progG`, su directorio `obj_files` deberá contener solo el fichero `garlic_itcm_proc.sobj`.

1.3 Refinamiento de las especificaciones de la práctica

A continuación se refinan algunos de los requisitos generales de la práctica proporcionados en el manual de la fase 1:

- **Ventanas de texto**: en esta segunda fase habrá 16 ventanas, aunque los programadores `progP` y `progM` todavía trabajarán con 4 ventanas en su desarrollo individual, ya que utilizarán el código de `progG` para la fase 1.

- **Pantalla de control:** en la pantalla inferior de la NDS se visualizará una tabla con información de los procesos en ejecución:



Las columnas de la tabla significan lo siguiente:

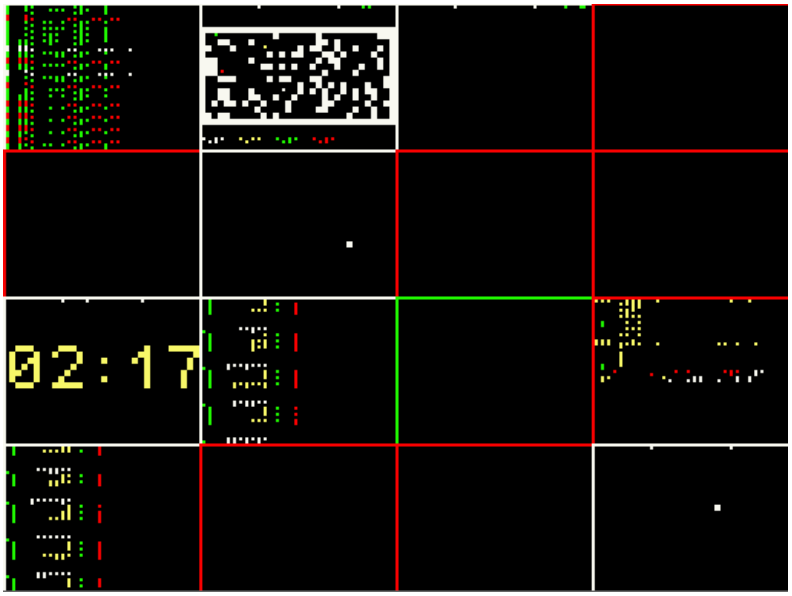
- **Z**: número de zócalo,
- **PID**: identificador del proceso (*Process IDentifier*),
- **Prog**: nombre en clave (*Key Name*) del programa que está ejecutando el proceso,
- **PCactual**: dirección de ejecución actual del proceso,
- **Pi**: gráfico de ocupación de la pila del proceso,
- **E**: estado del proceso (R: *Run*, Y: *readY*, B: *Blocked*),
- **Uso**: tanto por ciento del uso de la CPU por parte del proceso.

Además, se usarán cuatro colores en el dibujo de la tabla para aportar la siguiente información:

- **Blanco**: el zócalo está cargado con un proceso,
- **Salmón**: el zócalo está libre,
- **Magenta**: el zócalo está en el foco de la interfaz de usuario (ver el apartado de las teclas de control),
- **Azul marino**: solo se utiliza para el título de la ventana y para marcar la '**R**' del proceso que está en *Run* en cada momento.

Además, debajo de la tabla se muestra un gráfico de ocupación de los 24 KBytes de memoria disponible para los procesos de usuario, con un color distintivo de cada zócalo y un patrón diferente según se trate de segmentos de código o datos.

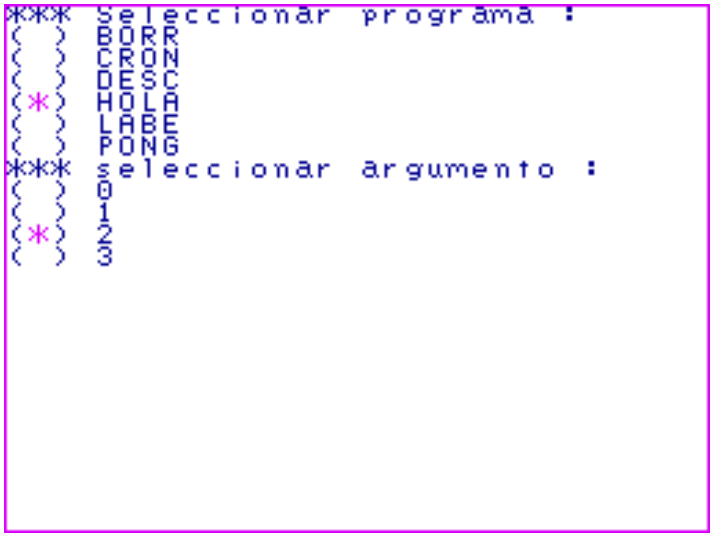
- **Teclas de control**: para controlar todo el sistema se propone una interfaz de usuario que se basará en la selección de una ventana con las teclas de dirección. La ventana seleccionada se enmarcará con un borde de color verde; en la imagen siguiente se observa un ejemplo de selección de la ventana correspondiente al zócalo 10:



La ventana seleccionada se dice que está "**en foco**". Para cambiar la ventana que está en foco y para otras manipulaciones de la interfaz de usuario se utilizarán los siguientes botones:

- **UP**, **DOWN**, **LEFT**, **RIGHT**: moverán el foco a la ventana de arriba, abajo, izquierda o derecha, respectivamente, sin permitir sobrepasar los límites de las ventanas,
- **Alt-L** y **Alt-R**: los botones alternativos de dirección de izquierda y derecha se utilizarán para aumentar y disminuir el zoom de visualización de las ventanas, permitiendo mostrar 16, 4 o 1 ventanas en cada momento,
- **START**: permitirá iniciar un nuevo proceso en el zócalo de la ventana en foco, provocando la finalización forzosa del proceso anterior en el caso de que existiera alguno en ese zócalo; para la ventana del zócalo 0 solo provocará la ampliación de la visualización de su contenido, ya que nunca se podrá finalizar el proceso de control del sistema operativo.

- **Selección de programas**: el procedimiento de selección del programa a ejecutar por un nuevo proceso provocará la ampliación de la visualización de la ventana (zoom 1-1) y mostrará una interfaz de selección del programa y de su argumento, que se podrá manipular con los botones **UP**, **DOWN** y **START** para mover y marcar una de las posibles opciones de cada apartado:



1.4 Ampliación del API para los programas de GARLIC

El API básico de la fase 1 se amplía con cuatro nuevas funciones:

```
     /* GARLIC_pid: devuelve el identificador del proceso actual */
extern int GARLIC_pid();

     /* GARLIC_random: devuelve un número aleatorio de 32 bits */
extern int GARLIC_random();

     /* GARLIC_divmod: calcula la división num / den (numerador /
denominador) */
extern int GARLIC_divmod(unsigned int num, unsigned int den,
                         unsigned int * quo, unsigned int * mod);

     /* GARLIC_divmodL: calcula la división larga num / den (numerador
/ denominador); el numerador y el cociente son de 64 bits */
extern int GARLIC_divmodL(long long * num, unsigned int * den,
                         long long * quo, unsigned int * mod);

     /* GARLIC_printf: escribe string en la ventana del proceso actual
*/
extern void GARLIC_printf(char * format, ...);



     /* GARLIC_printchar: escribe un carácter (c), especificado como
código de baldosa, en la posición (vx, vy) de la ventana del proceso
actual, con el color especificado por parámetro */
extern void GARLIC_printchar(int vx, int vy, char c, int color);

     /* GARLIC_printmat: escribe una matriz de caracteres (m) en la
posición (vx, vy) de la ventana del proceso actual, con el color
especificado por parámetro */
extern void GARLIC_printmat(int vx, int vy, char m[][8], int color);

     /* GARLIC_delay: retarda la ejecución del proceso actual el
número de segundos que se especifica por parámetro */
extern void GARLIC_delay(unsigned int nsec);

     /* GARLIC_clear: borra todo el contenido de la ventana del
proceso que invoca esta función */
extern void GARLIC_clear();
```

La implementación de estas funciones se encuentra en el fichero `GARLIC_API.s`, utilizando nuevos desplazamientos al vector de direcciones de las rutinas del API del sistema operativo, que ahora quedará definido en el fichero `GARLIC_OS/source/garlic_vectors.s` como sigue:

```
.section .vectors,"a",%note

APIVector:              @; Vector de direcciones de rutinas del API
      .word _ga_pid     @; (código de rutinas en "garlic_itcm_api.s")
      .word _ga_random
      .word _ga_divmod
      .word _ga_divmodL
      .word _ga_printf
      .word _ga_printchar
      .word _ga_printmat
      .word _ga_delay
      .word _ga_clear
```

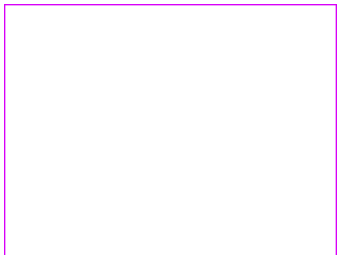
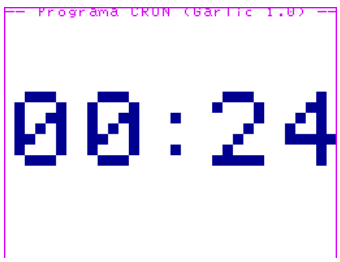
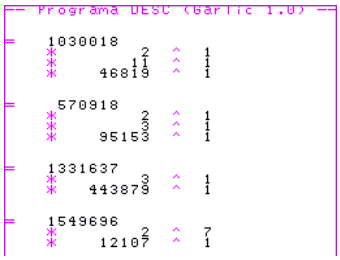
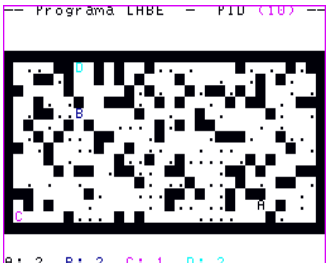
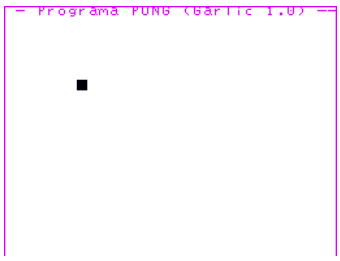
Los programadores `progT` deberán añadir todas las rutinas API específicas de teclado **al final del vector**, incluida la rutina `_ga_getstring()` definida en la fase 1, con el fin de mantener la máxima compatibilidad entre todos los programas de GarlicOS.

Dentro del repositorio `Garlic_OS_2.git` se proporciona un fichero específico de la implementación de las funciones internas `garlic_itcm_api.s` para las ramas `progP`, `progM` y `progG`, todos ellos diferentes de la versión del proyecto fusionado, almacenada en la rama `master`.

Todas las actualizaciones de los ficheros relativos a la API del sistema (`GARLIC_API.h`, `GARLIC_API.s`, `garlic_vectors.s` y `garlic_itcm_api.s`) deberán sustituir a sus respectivas versiones de la fase 1.

1.5 Nuevos programas para GarlicOS 2.0

En el directorio `GARLIC_Progs` se han creado nuevos programas para poder probar las nuevas funcionalidades de la fase 2:

Programa	*Descripción*
	**BORR** Borra el contenido de la ventana. Invoca `GARLIC_clear()` Argumento: indiferente.
	**CRON** Cronómetro de segundos y minutos. Invoca `GARLIC_printmat()`, `GARLIC_delay()` Argumento: 0 → rápido; 1 → cambia cada segundo (tiempo real); 2 → cambia cada 2 segundos; 3 → cambia cada 3 segundos.
	**DESC** Descomposición en factores primos. Invoca `GARLIC_printf()`, `GARLIC_random()`, `GARLIC_divmod()` Argumento: 0 → números del 0 a 100; 1, 2, 3 → números aleatorios con 21, 25 y 29 bits de longitud, respectivamente.
	**LABE** Recorrido de un laberinto, realizado por unos caracteres que "comen" puntos. Invoca `GARLIC_random()`, `GARLIC_printchar()` Argumento: 0,1,2,3 → indica el número de caracteres (de 1 a 4) y el tamaño del laberinto (8, 16, 24, 32 columnas).
	**PONG** Pelota que rebota. Invoca `GARLIC_printchar()`, `GARLIC_delay()` Argumento: 0 → avance rápido de la pelota; 1, 2, 3 → avance de la pelota cada 1, 2 o 3 segundos.

Los programas `HOLA.elf` y `PRNT.elf`, compilados para GarlicOS 1.0, deben funcionar sobre GarlicOS 2.0 sin recompilarlos con la nueva API (compatibilidad a nivel de ficheros binarios).

1.6 Estructuras de datos de GarlicOS 2.0

Para la fase 2 se han modificado las estructuras de datos globales del sistema operativo, definidas en el fichero `garlic_dtcm.s`:

```
.section .dtcm,"wa",%progbits

	.global _gd_pidz	@; Identificador de proceso + zócalo actual
_gd_pidz:	.word 0

	.global _gd_pidCount		@; Contador global de PIDs
_gd_pidCount:	.word 0

	.global _gd_tickCount		@; Contador global de tics
_gd_tickCount:	.word 0

	.global _gd_sincMain	@; Sincronismos con programa principal
_gd_sincMain:	.word 0

	.global _gd_seed		@; Semilla para números aleatorios
_gd_seed:	.word 0xFFFFFFFF

	.global _gd_nReady	@; Número de procesos en cola de READY
_gd_nReady:		.word 0

	.global _gd_qReady	@; Cola de READY (procesos preparados)
_gd_qReady:		.space 16

	.global _gd_nDelay	@; Número de procesos en la cola de DELAY
_gd_nDelay:		.word 0

	.global _gd_qDelay	@; Cola de DELAY (procesos retardados)
_gd_qDelay:		.space 16 * 4

	.global _gd_pcbs		@; Vector de PCBs de los procesos activos
_gd_pcbs:	.space 16 * 6 * 4

	.global _gd_wbfs			@; Vector de WBUFs de las ventanas
_gd_wbfs:	.space 16 * (4 + 64)

	.global _gd_stacks		@; Pilas de los procesos de usuario
_gd_stacks:		.space 15 * 128 * 4
```

Los cambios respecto a la fase 1 son los siguientes:

- `_gd_sincMain`: nueva variable que permite sincronizar el programa de control del sistema operativo con la RSI del *timer* 0 para el cálculo del porcentaje de uso de la CPU y con la rutina `_gp_terminarProc()` para detectar la finalización de los procesos de usuario,

- `_gd_nDelay` y `_gd_qDelay[]`: contador y cola de procesos retardados; son similares a las `_gd_nReady` y `_gd_qReady[]`, pero en la cola de procesos retardados se almacena, junto con el número de zócalo (8 bits altos), el número de tics (*VBlanks*) que faltan para que el proceso salga de su bloqueo temporal,

- `_gd_wbfs[]`: se modificada el número de ventanas y la estructura del buffer de caracteres, que ahora es de un *short* por cada posición para poder codificar el color de las baldosas correspondientes, tal como se describe en el fichero `garlic_system.h`:

```
//-----------------------------------------------------------------
//    Variables globales del sistema (garlic_dtcm.s)
//-----------------------------------------------------------------

typedef struct              // Estructura del buffer de una ventana
{                           // (WBUF: Window BUFfer)
      int pControl;         // control de escritura en ventana
                            // 4 bits altos: cód. color actual (0..3)
                            // 12 bits medios: número de línea (0..23)
                            // 16 bits bajos: car. pendientes (0..32)
      short pChars[32];     // vector 32 car. pendientes de escritura
                            // 16 bits por entrada, indicando núm. de
                            // baldosa corresp. al carácter+color
} PACKED garlicWBUF;

extern garlicWBUF _gd_wbfs[16];  // vector con buffers de 16 ventanas
```

1.7 Estructura de funciones de GARLIC

En el fichero `garlic_system.h` también se han incluido los prototipos (cabeceras) de las nuevas funciones relativas a las tareas de la segunda fase. A continuación se muestran las funciones y rutinas adicionales y modificadas de la versión de GarlicOS 2.0 para cada rol, excepto para el `progT`:

```
//---------------------------------------------------------------------
//   Rutinas de gestión de procesos (garlic_itcm_proc.s)
//---------------------------------------------------------------------

/* _gp_retardarProc:  retarda la ejecución del proceso */
extern void _gp_retardarProc(int nsec);

/* _gp_matarProc:     elimina un proceso del sistema */
extern void _gp_matarProc(int zocalo);

/* _gp_rsiTIMER0:     escribe periódicamente el % de uso de la CPU en
la tabla de procesos */
extern void _gp_rsiTIMER0();


//---------------------------------------------------------------------
//   Funciones de gestión de memoria (garlic_mem.c)
//---------------------------------------------------------------------

/* _gm_listaProgs: devuelve la lista de programas disponibles */
extern int _gm_listaProgs(char *progs[]);

/* _gm_cargarPrograma: busca un fichero de nombre "(keyName).elf" y
lo carga en memoria de programas, asignando dicha memoria al zócalo
indicado */
extern intFunc _gm_cargarPrograma(int zocalo, char *keyName);

//---------------------------------------------------------------------
//   Rutinas de soporte a la gestión de memoria (garlic_itcm_mem.s)
//---------------------------------------------------------------------

/* _gm_reubicar: reubica las referencias absolutas a la memoria del
programa, analizando 1 o 2 segmentos del programa (código y datos) */
extern void _gm_reubicar(char *fileBuf,
                unsigned int pAddr_code, unsigned int *dest_code,
                unsigned int pAddr_data, unsigned int *dest_data);
```

```
/* _gm_reservarMem: reserva una zona de memoria para el zócalo z */
extern void * _gm_reservarMem(int z, int tam, unsigned char tipo_seg);

/* _gm_liberarMem: libera toda la memoria asignada al zócalo z */
extern void * _gm_liberarMem(int z);

/* _gm_rsiTIMER1:     escribe periódicamente el uso de la pila y el
estado de cada proceso en la tabla de procesos */
extern void _gm_rsiTIMER1();


//--------------------------------------------------------------------
//    Funciones de gestión de gráficos (garlic_graf.c)
//--------------------------------------------------------------------

/* _gg_iniGraf: inicializa el procesador gráfico A para GARLIC 2.0 */
extern void _gg_iniGrafA();

/* _gg_generarMarco: dibuja el marco de la ventana v con el color
indicado */
extern void _gg_generarMarco(int v, int color);

/* _gg_escribir: escribe una cadena de caracteres en el zócalo
indicado, atendiendo a los códigos de formato (%c, %d, %x, %s) para
los valores pasados por parámetro (val1, val2) además de nuevos códigos
para indicar el color actual de los caracteres (%0, %1, %2, %3) */
extern void _gg_escribir(char *formato, unsigned int val1,
                                 unsigned int val2, int ventana);

//--------------------------------------------------------------------
//    Rutinas de soporte a la gestión de gráficos (garlic_itcm_graf.s)
//--------------------------------------------------------------------

/* _gg_escribirLinea: transferir el contenido del buffer de baldosas
de una ventana a las posiciones correspondientes en el mapa de baldosas
*/
extern void _gg_escribirLinea(int v, int f, int n);

/* _gg_escribirCar: escribe un carácter (baldosa) en una posición de
la ventana */
extern void _gg_escribirCar(int vx, int vy, char c, int color,
                                                    int ventana);
```

```
/* _gg_escribirMat: escribe una matriz de 8x8 caracteres en una
posición de la ventana */
extern void _gg_escribirMat(int vx, int vy, char m[][8], int color,
                                                    int ventana);

/* _gg_escribirLineaTabla: escribe la información básica de una linea
de la tabla de procesos, con el color especificado */
extern void _gg_escribirLineTabla(int z, int color);

/* _gg_rsiTIMER2:     escribe periódicamente el valor actual del PC
de cada proceso en la tabla de procesos */
extern void _gg_rsiTIMER2();
```

Atención: el fichero `garlic_system.h` es diferente en las ramas para cada programador, puesto que incluye la definición de la fase 1 para las funciones de los otros programadores. Solo la rama `master` contiene la versión para el proyecto fusionado y finalizado.

Todas las versiones del fichero `garlic_system.h` incluyen la definición de nuevas funciones de soporte, que ya se proporcionan implementadas en el fichero `garlic_itcm_sys.s`:

```
//------------------------------------------------------------------
//    Rutinas de soporte al sistema (garlic_itcm_sys.s)
//------------------------------------------------------------------

/* _gs_borrarVentana: borra la ventana del zócalo indicado */
extern void _gs_borrarVentana(int zocalo, int modo);

/* _gs_iniGrafB: inicializa el procesador gráfico B para GARLIC 2.0
*/
extern void _gs_iniGrafB();

/*  _gs_escribirStringSub:  escribe  un  string  en  una  determinada
posición de la pantalla del procesador gráfico secundario */
extern void _gs_escribirStringSub(char *string, int fil, int col,
                                                    int color);

/* _gs_dibujarTabla:  dibuja la tabla de procesos */
extern void _gs_dibujarTabla();
```

```
/* _gs_pintarFranjas: pinta las franjas de memoria asignadas a un
 segmento (de código o datos) del zócalo indicado por parámetro */
extern void _gs_pintarFranjas(unsigned char zocalo,
                              unsigned short index_ini,
                              unsigned short num_franjas,
                              unsigned char tipo_seg);

/* _gs_representarPilas: representa gráficamente la ocupación de las
 pilas de los procesos de usuario, sobre la tabla de control de
procesos */
extern void _gs_representarPilas();
```

2 Tareas de gestión del procesador (progP)

2.1 Retardo de procesos

La nueva función del API `GARLIC_delay()` permitirá a los procesos retardar su ejecución un cierto número de segundos:

```
      /* GARLIC_delay: retarda la ejecución del proceso actual el
            número de segundos que se especifica por el parámetro
            (nsec); el rango permitido será de [0..600] (máx. 10
            minutos); el valor 0 provocará que el proceso se desbanque
            y pase a READY, sin determinar un retardo concreto (el
            que resulte de volver a restaurar el proceso); */
extern void GARLIC_delay(unsigned int nsec);
```

A través del vector de direcciones, las llamadas a la función del API invocarán la rutina interna `_ga_delay()`, que se proporciona en el fichero `garlic_itcm_api.s`:

```
      .global _ga_delay
      @;Parámetros
      @; R0: int nsec
_ga_delay:
      push {r2-r3, lr}
      ldr r3, =_gd_pidz      @; R3 = dirección _gd_pidz
      ldr r2, [r3]
      and r2, #0xf           @; R2 = zócalo actual
      cmp r0, #0
      bhi .Ldelay1
      bl _gp_WaitForVBlank   @; si nsec = 0, solo desbanca el proceso
      b .Ldelay2             @; y salta al final de la rutina
.Ldelay1:
      cmp r0, #600
      movhi r0, #600         @; limitar el número de segundos a 600
      bl _gp_retardarProc
.Ldelay2:
      pop {r2-r3, pc}
```

Después de aplicar algunos filtros al parámetro `nsec`, `_ga_delay()` llama a la rutina `_gp_retardarProc()`, que se implementará en el fichero `garlic_itcm_proc.s`:

```
	.global _gp_retardarProc
	@; retarda la ejecución de un proceso durante cierto número de
	@; segundos, colocándolo en la cola de DELAY
	@;Parámetros
	@; R0: int nsec
_gp_retardarProc:
	push {lr}

	pop {pc}
```

Esta rutina pone el proceso que la llama en la nueva cola de DELAY. El funcionamiento de esta cola es similar a la cola de READY: almacena una lista de identificadores de zócalos de los procesos que se encuentran en un cierto estado. Sin embargo, los procesos que se encuentran en esta cola están **bloqueados**, de modo que la rutina de cambio de contexto **no** los restaurará mientras no vuelvan a añadirse a la cola de procesos preparados.

Por lo tanto, ahora tendremos tres estados de los procesos:

- ***RUN***: el proceso que se está ejecutando en cada instante,
- ***READY***: procesos que están preparados para pasar a RUN, cuando les toque el turno,
- ***BLOCKED***: procesos bloqueados a la espera de que ocurra un evento que los desbloquee, en este caso, el paso de un cierto tiempo.

Otra diferencia importante es que la cola de procesos retardados incorpora también el número de retrocesos verticales (VBLs) que debe esperar el proceso en la cola de DELAY. Por este motivo, las posiciones de esta cola son *words* (32 bits), ya que se incluirá información sobre el número de tics (VBLs) restantes:

```
extern int _gd_nDelay;		// Número de procesos en cola de DELAY

extern int _gd_qDelay[16]; // Cola de DELAY (procesos retardados) :
	// vector con _gd_nDelay entradas, conteniendo los
	// identificadores de los zócalos (8 bits altos) más el número
	// de tics restantes (16 bits bajos) para desbloquear el proceso
```

Por tanto, las tareas que debe realizar la rutina `_gp_retardarProc()` son las siguientes:

- calcular cuantos tics (VBLs) corresponden al número de segundos a retardar el proceso,
- construir un *word* con los 8 bits altos igual al zócalo del proceso actual y los 16 bits bajos con el número de tics a retardar, e incluir este valor en el vector `_gd_qDelay[]` (incrementar `_gd_nDelay`),
- fijar el bit más alto de la variable global `_gd_pidz` a 1, para que la rutina de salvar contexto sepa que no debe añadir el zócalo a la cola de procesos preparados,
- forzar la cesión de la CPU invocando a la rutina `_gp_WaitForVBlank()`.

El proceso retardado se tiene que desbancar con el mecanismo habitual, es decir, se tiene que salvar su contexto en su pila y en el vector `_gd_pcbs[]`. Sin embargo, la rutina `_gp_salvarProc()` no debe poner el zócalo del proceso en la cola de READY, atendiendo a la marca de aviso en el bit de más peso de la variable `_gd_pidz`.

Por último, para gestionar el final del bloqueo de los procesos retardados será necesario modificar la rutina `_gp_rsiVBL()`, de modo que a cada interrupción de retroceso vertical se decremente el contador de tics de todos los procesos retardados. Cuando el contador de tics de un proceso retardado llegue a 0, será necesario sacar al proceso de la cola de DELAY y ponerlo en la cola de READY.

Para realizar estas tareas, en el fichero `garlic_itcm_proc.s` adicional para la fase 2 se ha dispuesto la cabecera de una rutina local que puede ser llamada desde la rutina `_gp_rsiVBL()`:

```
	@; Rutina para actualizar la cola de procesos retardados,
	@; poniendo en cola de READY aquellos cuyo número de tics
	@; de retardo sea 0
_gp_actualizarDelay:
	push {lr}

	pop {pc}
```

2.2 Matar procesos

Otra de las tareas de `progP` consiste en implementar la rutina `_gp_matarProc()`, que permitirá eliminar un proceso antes de que termine su ejecución:

```
	.global _gp_matarProc
	@; Rutina para destruir un proceso de usuario:
	@; borra el PID del PCB del zócalo referenciado por parámetro,
	@; para indicar que esa entrada del vector _gd_pcbs[] está libre;
	@; elimina el índice de zócalo de la cola de READY o de la cola
	@; de DELAY, esté donde esté;
	@; Parámetros:
	@;   R0:   zócalo del proceso a matar (entre 1 y 15).
_gp_matarProc:
	push {lr}

	pop {pc}
```

Se trata de hacer lo siguiente:

- poner el campo `_gd_pcbs[z].PID` a cero, para indicar que el zócalo `z` está libre, lo cual permitirá cargar otro proceso en ese mismo zócalo,
- buscar el valor de `z` en la cola de READY y eliminarlo en caso de haberlo encontrado,
- sino se ha encontrado en la cola de READY, buscar el valor de `z` en la cola de DELAY y eliminarlo en caso de haberlo encontrado.

Solo el proceso de control del sistema operativo llamará a la rutina `_gp_matarProc()` para eliminar cualquier proceso de usuario.

2.3 Generación del porcentaje de uso de la CPU

Para realizar el cálculo del porcentaje de uso de la CPU de cada proceso, el programador `progP` deberá implementar una RSI para el *timer* 0, de nombre `_gp_rsiTIMER0()` (dentro del fichero `garlic_itcm_proc.s`), que se ejecutará una vez por segundo y deberá realizar las siguientes operaciones:

- sumar los tics de trabajo (`workTicks`) de todos los procesos activos,
- calcular el porcentaje de los tics de trabajo de cada proceso respecto al total,
- poner a cero los tics de trabajo de cada proceso y guarda el porcentaje calculado en los 8 bits altos del campo `_gd_pcbs[z].workTicks` de cada proceso,
- escribir el porcentaje calculado en la tabla de procesos de la pantalla inferior de la NDS,
- poner a uno el bit 0 de la variable global `_gd_sincMain`, para que el programa principal pueda detectar que ya dispone de esa información.

Por lo tanto, si se instala la RSI del *timer* 0 en el vector de interrupciones y se configura convenientemente el controlador del *timer* 0, el cálculo del porcentaje se realizará periódicamente sin ninguna intervención de ningún proceso de usuario ni del proceso de control del sistema operativo.

Sin embargo, para que la rutina `_gp_rsiTIMER0()` pueda realizar los cálculos correctamente será necesario incrementar el contador de tics de trabajo en el campo `_gd_pcbs[z].workTicks` de cada proceso activo.

Hay que tener en cuenta que el campo `_gd_pcbs[z].workTicks` contiene los tics de trabajo en los 24 bits bajos, ya que los 8 bits altos se utilizan para almacenar temporalmente el porcentaje. Sin embargo, no habrá ningún conflicto con los bits, si el número de tics de trabajo se "refresca" cada segundo (máximo, 60 tics).

El programador `progP` deberá decidir dónde es mejor incrementar los tics de trabajo.

2.4 Secciones críticas

En esta segunda fase de la práctica se tendrán que tomar precauciones para evitar problemas debidos a la concurrencia del sistema.

Esto significa que, mientras se están manipulando variables globales relacionadas con la gestión de procesos, podría llegar una interrupción que provocara una modificación de dichas variables, lo cual podría generar inconsistencias graves en el funcionamiento de la multiplexación.

Por ejemplo, supongamos que un proceso termina su ejecución y, después de poner su campo `_gd_pcbs[z].PID` a cero, llega una interrupción y provoca un cambio de contexto antes de que la rutina `_gp_terminarProc()` pueda poner el valor del PID de la variable global `_gd_pidz` a cero, para indicar a la rutina `_gp_rsiVBL()` que no debe salvar el contexto.

En este supuesto, tendríamos un proceso terminado según el campo PID de su entrada en el vector PCB, pero con su número de zócalo todavía almacenado en la cola de READY, lo cual supondría una **inconsistencia** en el sistema que podría provocar múltiples problemas, como cargar otro programa en el zócalo aparentemente libre.

Por este motivo, la nueva versión del fichero `garlic_itcm_proc.s` incorpora una versión de la rutina `_gp_terminarProc()` que realiza llamadas a dos nuevas rutinas `_gp_inhibirIRQs()` y `_gp_desinhibirIRQs()`, que deberán evitar que se interrumpa la secuencia de instrucciones para la terminación de un proceso. Es decir, se establece una sección crítica en la ejecución del código, a nivel de activación y desactivación de las interrupciones del sistema.

Será necesario tomar las mismas precauciones en otras rutinas que requieran secciones críticas.

El programador `progP` será el encargado de programar las rutinas `_gp_inhibirIRQs()` y `_gp_desinhibirIRQs()`, cuya definición se encuentra reseñadas dentro del fichero `garlic_itcm_proc.s` proporcionado para iniciar la segunda fase de la práctica.

2.5 Programa principal para progP

Dentro de la rama `progP` del respositorio `Garlic_OS_2.git` se proporciona un programa principal `main.c` específicamente preparado para poder probar las rutinas relativas a la gestión de procesos sin tener que depender del trabajo de los otros programadores. Sin embargo, este programa principal requiere disponer de una versión operativa de las funciones y rutinas de los roles `progM` y `progG` para la fase 1.

```
/*------------------------------------------------------------------
     "main.c" : fase 2 / progP
------------------------------------------------------------------*/
#include <nds.h>
#include <stdlib.h>
#include "garlic_system.h"

const short divFreq0 = -33513982/1024;// frecuencia TIMER0 = 1 Hz

//------------------------------------------------------------------
void inicializarSistema() {
//------------------------------------------------------------------
     _gg_iniGrafA();       // inicializar procesadores gráficos
     _gg_iniGrafB();
     _gs_dibujarTabla();
     ...
     _gd_pcbs[0].keyName = 0x4C524147;     // "GARL"

     if (!_gm_InitFS()) {
          _gg_escribir("ERROR: ¡no se puede inicializar el sistema
de ficheros!", 0, 0, 0);
          exit(0);
     }

     irqInitHandler(_gp_IntrMain);         // inicializar IRQs
     irqSet(IRQ_VBLANK, _gp_rsiVBL);
     irqEnable(IRQ_VBLANK);

     irqSet(IRQ_TIMER0, _gp_rsiTIMER0);
     irqEnable(IRQ_TIMER0);     // instalar la RSI para el TIMER0
     TIMER0_DATA = divFreq0;
     TIMER0_CR = 0xC3;          // Timer Start | Timer IRQ Enabled

     REG_IME = IME_ENABLE;      // activar las interr. en general
}
```

En las inicializaciones se observa cómo se activa el entorno gráfico y el sistema de ficheros, además de inicializar las interrupciones e instalar las RSIs para el *Vertical Blank* y el *timer* 0.

```
//--------------------------------------------------------------
int main(int argc, char **argv) {
//--------------------------------------------------------------
      intFunc start;
      ...
      inicializarSistema();
      ...
      _gg_escribir("*** Inicio fase 2_P\n", 0, 0, 0);

      _gg_escribir("*** Carga de programa HOLA.elf\n", 0, 0, 0);
      start = _gm_cargarPrograma("HOLA");
      if (start)
      {     _gp_crearProc(start, 1, "HOLA", 3);
            _gp_crearProc(start, 2, "HOLA", 3);
            _gp_crearProc(start, 3, "HOLA", 3);

            while (_gd_tickCount < 240)      // esperar 4 segundos
            {     _gp_WaitForVBlank();
                  porcentajeUso();
            }
            _gp_matarProc(1);                // matar proceso 1
            _gg_escribir("Proceso 1 eliminado\n", 0, 0, 0);
            _gs_dibujarTabla();

            while (_gd_tickCount < 480)      // esperar 4 segundos más
            {     _gp_WaitForVBlank();
                  porcentajeUso();
            }
            _gp_matarProc(3);                // matar proceso 3
            _gg_escribir("Proceso 3 eliminado\n", 0, 0, 0);
            _gs_dibujarTabla();

            while (_gp_numProc() > 1) // esperar a que proceso 2 acabe
            {     _gp_WaitForVBlank();
                  porcentajeUso();
            }
            _gg_escribir("Proceso 2 terminado\n", 0, 0, 0);
      } else
            _gg_escribir("*** Programa NO cargado\n", 0, 0, 0);
```

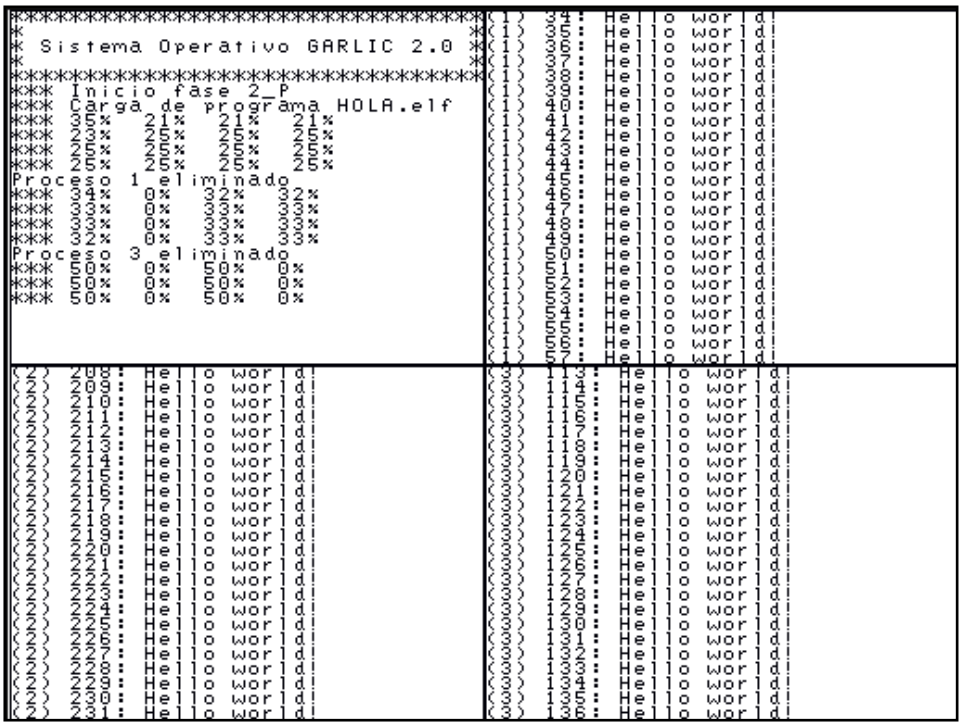
```
	_gg_escribir("*** Carga de programa PONG.elf\n", 0, 0, 0);
	start = _gm_cargarPrograma("PONG");
	if (start)
	{	_gp_crearProc(start, 1, "PONG", 1);
		_gp_crearProc(start, 2, "PONG", 2);
		_gp_crearProc(start, 3, "PONG", 3);

		mtics = _gd_tickCount + 960;
		while (_gd_tickCount < mtics)	// esperar 16 segundos más
		{	_gp_WaitForVBlank();
			porcentajeUso();
		}
		_gp_matarProc(1);	// matar los 3 procesos a la vez
		_gp_matarProc(2);
		_gp_matarProc(3);
		_gg_escribir("Procesos 1, 2 y 3 eliminados\n", 0, 0, 0);

		while (_gp_numProc() > 1)	// esperar a que todos los
		{	_gp_WaitForVBlank();	// procesos acaben
			porcentajeUso();
		}
	} else
		_gg_escribir("*** Programa NO cargado\n", 0, 0, 0);

	_gg_escribir("*** Final fase 2_P\n", 0, 0, 0);
	_gs_dibujarTabla();

	while(1) {
		_gp_WaitForVBlank();
	}		// parar el procesador en un bucle infinito
	return 0;
}
```

En la rutina principal `main()` se distinguen dos partes.

La parte de los procesos HOLA permite verificar el procedimiento para matar procesos. En este caso utilizamos el contador de tics general (variable `_gd_tickCount`) para esperar un cierto tiempo antes de eliminar los procesos 1 y 3. En el caso del proceso 2 simplemente se espera a que acabe, con el fin de verificar el correcto funcionamiento de la nueva versión de la rutina `_gp_terminarProc()`.

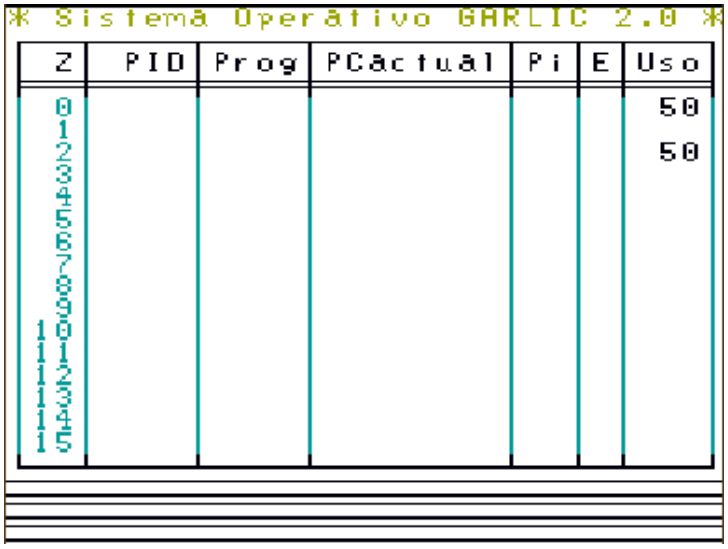
La parte de los procesos PONG permite verificar el procedimiento para retardar procesos. Además, también comprueba la eliminación de varios procesos a la vez.

El resultado final que se observará en pantalla de la NDS será similar al siguiente:



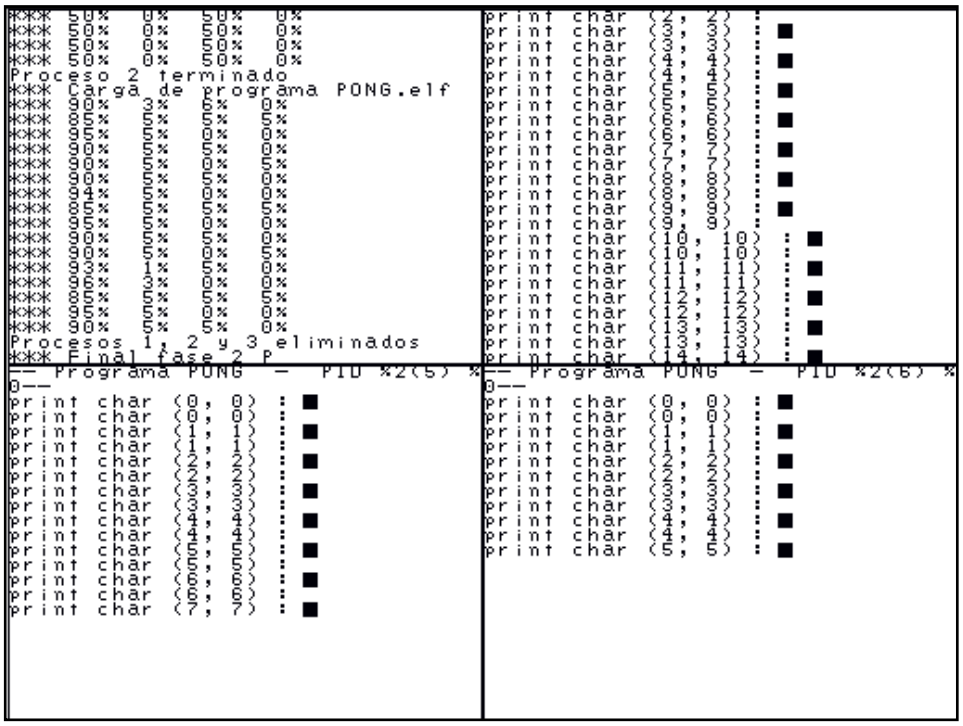
Aunque no se pueda observar en esta imagen estática, solo la ventana del proceso 2 estaba imprimiendo mensajes en el momento de capturar la pantalla. Esto será un indicio de que la rutina de matar procesos funciona para procesos que están en READY.

Además, la función `porcentajeUso()` del fichero `main.c` visualiza los porcentajes de uso de los procesos del 0 al 3 en la ventana del proceso de control del sistema operativo, lo cual también permitirá demostrar que los procesos se van eliminando de la cola de READY. Dichos porcentajes también se visualizan en la tabla de procesos, desde la rutina `_gp_rsiTIMER0()`:



En la segunda parte del programa se inician tres procesos con el programa `PONG.elf`, el cual está preparado para invocar a las funciones del API `GARLIC_delay()` y `GARLIC_printchar()`, con el fin de probar la invocación de las rutinas `_gp_retardarProc()` y `_gg_escribirCar()`, respectivamente.

Sin embargo, cuando se está programando las rutinas de la rama `progP` puede que todavía no se disponga de la rutina `_gg_escribirCar()`, ya que pertenece a las nuevas tareas de `progG`. Para salvar este inconveniente el fichero `garlic_itcm_api.s` que se proporciona en la rama `progP` (y también en la `progM`) contienen una variación en la implementación de `_ga_printchar()` que permite imprimir los parámetros de `GARLIC_printchar()` con llamadas a la función `_gg_escribir()` (de la fase 1). De este modo podremos observar si funciona o no el código del programa `PONG.elf`:



En la ventana del proceso de control del sistema operativo se muestra cómo se han eliminado correctamente los tres procesos de usuario. Aunque en la imagen estática no se puede observar el movimiento, los mensajes que genera el programa `PONG.elf` se retardaban uno, dos y tres segundos en cada ventana de los procesos de usuario, lo cual indicará que la rutina `_gp_retardarProc()` funciona correctamente (en este rango de retardos). Por supuesto, estas pruebas también contribuirán a corroborar el correcto funcionamiento del resto de rutinas del fichero `garlic_itcm_proc.s`, puesto que los procesos se deberán multiplexar correctamente a través de las colas de READY y DELAY, además de contabilizar la correspondiente carga de CPU de cada proceso.

3 Tareas de gestión de la memoria (progM)

3.1 Programas con varios segmentos

Aunque la mayoría de los programas de ejemplo proporcionados para GarlicOS 2.0 presentan un único segmento de código, el programa `DESC.elf` presenta dos segmentos, uno de código y otro de datos:

```
$ arm-none-eabi-readelf -l GARLIC_OS/nitrofiles/Programas/DESC.elf

Elf file type is EXEC (Executable file)
Entry point 0x8274
There are 2 program headers, starting at offset 52

Program Headers:
  Type  Offset   VirtAddr   PhysAddr  FileSiz MemSiz  Flg Align
  LOAD 0x000400 0x00008000 0x00008000 0x005f7 0x005f7 R E 0x400
  LOAD 0x0009f8 0x000089f8 0x000089f8 0x00fa0 0x00fd8 RW  0x400

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata
   01     .data .bss
```

Los dos segmentos son de tipo LOAD, es decir, los dos se tienen que cargar en memoria.

Se han resaltado en rojo el valor de los flags para hacer notar que un segmento es de solo lectura y ejecución (R E), mientras que el otro es de lectura y escritura (RW). Esto está relacionado con la asignación de secciones a los segmentos: el primer segmento incluye código ejecutable más datos constantes (`.text` `.rodata`), mientras que el segundo segmento contiene variables globales inicializadas y no inicializadas (`.data` `.bss`).

También se ha resaltado en rojo los dos tamaños en fichero y en memoria del segundo segmento, para hacer notar que son diferentes, ya que la sección de variables globales no inicializadas (`.bss`) no se incluye dentro del fichero, pero sí se debe reservar espacio en memoria para alojar las variables de dicha sección.

3.2 Reubicación de direcciones con varios segmentos

El problema de la reubicación de direcciones se complica para el caso de tener varios segmentos a cargar, ya que se debe realizar en base a las posiciones de origen y de destino de cada segmento.

Para poder distinguir a qué segmento pertenece cada dirección a reubicar, se debería consultar la sección de tabla de símbolos, que está referenciada en el campo `sh_link` de la sección de reubicadores.

Para evitar esta tarea vamos a asumir que cualquier programa para GarlicOS 2.0 **solo podrá presentar uno o dos segmentos**:

- Si hay un único segmento podemos suponer que es el de código,
- Si hay dos segmentos podemos suponer que el primero (posiciones de memoria inferiores) es el de código y el segundo (posiciones de memoria superiores) es el de datos.

Para verificar el tipo de información que contiene cada segmento podemos utilizar el valor del campo `p_flags` de cada entrada de la tabla de segmentos, que será 5 (R-E) para el segmento de código y 6 (RW-) para el segmento de datos.

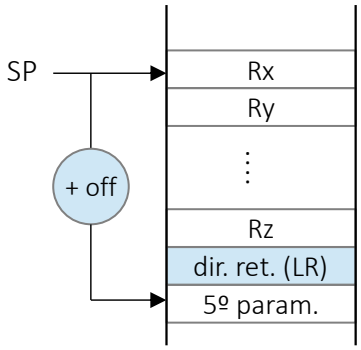
Para tener en consideración estos nuevos requisitos de reubicación respecto a la fase 1 de la práctica, se deberá modificar la rutina `_gm_reubicar()` para que pueda operar con la información de los dos posibles segmentos:

```
/* _gm_reubicar: rutina de soporte a _gm_cargarPrograma(),
                 que interpreta los 'relocs' de un fichero ELF,
                 contenido en un buffer *fileBuf, y ajusta las
                 direcciones de memoria correspondientes a las
                 referencias de tipo R_ARM_ABS32, a partir de las
                 direcciones de memoria destino de código (dest_code)
                 y datos (dest_data), y según el valor de las
                 direcciones de las referencias a reubicar y de las
                 direcciones de inicio de los segmentos de código
                 (pAddr_code) y datos (pAddr_data); */
void _gm_reubicar(char *fileBuf,
                  unsigned int pAddr_code, unsigned int *dest_code,
                  unsigned int pAddr_data, unsigned int *dest_data)
```

El algoritmo que se propone en esta fase es una modificación del algoritmo de la fase 1:

- para cada reubicador de tipo R_ARM_ABS32:
    - obtener la dirección de memoria a reubicar en base al segmento de código, ya que las referencias a memoria de variables o constantes (punteros) siempre se encuentran entre las rutinas del programa (direccionamiento relativo a PC),
    - obtener el contenido de la dirección de memoria a reubicar y aplicarle la reubicación respecto al inicio del segmento de código o al inicio del segmento de datos, en función de la posición de la dirección a reubicar respecto al rango de direcciones de los segmentos de código y datos;

Hay que tener en cuenta que la nueva versión de la rutina tendrá ahora 5 parámetros, lo cual implica que los cuatro primeros se pasan en los registros R0-R3, mientras que el quinto parámetro se pasa por la pila. Por lo tanto, para obtener su valor desde dentro de la rutina habrá que acceder a la pila utilizando un desplazamiento adecuado respecto al registro `sp`, según los registros apilados al entrar en la rutina:




```
_gm_reubicar:
     push {Rx, Ry, ..., Rz, LR}
     :
     :
     :
     pop {Rx, Ry, ..., Rz, PC}
```

3.3 Gestión de memoria

En la primera fase de la práctica los segmentos de código se cargaban consecutivamente a partir de la dirección inicial `0x01002000`. En esta segunda fase, los segmentos de cada proceso se tendrán que cargar en zonas de memoria libre dentro de los 24 Kbytes que siguen a la misma dirección inicial `0x01002000` (memoria ITCM de la NDS).

Para gestionar las zonas de memoria que ocupa cada segmento se propone utilizar el vector `_gm_zocMem[]`, definido en el fichero `garlic_itcm_mem.s`, dentro de la memoria DTCM de la NDS:

```
NUM_FRANJAS = 768
INI_MEM_PROC = 0x01002000

.section .dtcm,"wa",%progbits
      .align 2
      .global _gm_zocMem
_gm_zocMem:      .space NUM_FRANJAS  @; vector ocupación franjas mem.
```

El vector no está declarado en el fichero `garlic_system.h` porque no se necesita acceder desde ninguna otra parte del sistema operativo escrito en lenguaje C. Si fuese necesario, su definición sería similar a la siguiente:

```
/*    _gm_zocMem: vector que almacenará el índice del zócalo que está
                  ocupando cada franja de memoria de 32 bytes */
extern unsigned char _gm_zocmem[768];
```

Se trata de un vector donde cada posición contendrá un número de zócalo, que indicará que un trozo o franja de 32 bytes de memoria está asignada al proceso que se ejecuta en ese zócalo.

La siguiente tabla muestra un ejemplo de asignación de las primeras 16 franjas de memoria, algunas asignadas a los procesos de los zócalos 1 y 3, y otras libres (contenido igual a 0):

Entradas del vector	*Contenido*	*Posiciones de la franja*
`_gm_zocMem[0]`	`0`	`0x01002000 .. 0x0100201F`
`_gm_zocMem[1]`	`0`	`0x01002020 .. 0x0100203F`
`_gm_zocMem[2]`	`1`	`0x01002040 .. 0x0100205F`
`_gm_zocMem[3]`	`1`	`0x01002060 .. 0x0100207F`
`_gm_zocMem[4]`	`1`	`0x01002080 .. 0x0100209F`
`_gm_zocMem[5]`	`0`	`0x010020A0 .. 0x010020BF`
`_gm_zocMem[6]`	`0`	`0x010020C0 .. 0x010020DF`
`_gm_zocMem[7]`	`3`	`0x010020E0 .. 0x010020FF`
`_gm_zocMem[8]`	`3`	`0x01002100 .. 0x0100211F`
`_gm_zocMem[9]`	`3`	`0x01002120 .. 0x0100213F`
`_gm_zocMem[10]`	`3`	`0x01002140 .. 0x0100215F`
`_gm_zocMem[11]`	`3`	`0x01002160 .. 0x0100217F`
`_gm_zocMem[12]`	`3`	`0x01002180 .. 0x0100219F`
`_gm_zocMem[13]`	`1`	`0x010021A0 .. 0x010021BF`
`_gm_zocMem[14]`	`1`	`0x010021C0 .. 0x010021DF`
`_gm_zocMem[15]`	`0`	`0x010021E0 .. 0x010021FF`
...	...	...

En el ejemplo anterior se observa que el proceso del zócalo 1 tiene asignadas tres franjas consecutivas de memoria, en el rango `0x01002040 .. 0x0100209F` (96 bytes), y después otras dos franjas consecutivas, en el rango `0x010021A0 .. 0x010021DF` (64 bytes). Por otro lado, el proceso del zócalo 3 tiene asignadas seis franjas consecutivas de memoria, en el rango `0x010020E0 .. 0x0100219F` (192 bytes).

Las rutinas que hay que implementar para realizar la gestión de la memoria disponible serán las siguientes (en el fichero `garlic_itcm_mem.s`):

```
/* _gm_reservarMem: rutina para reservar un conjunto de franjas de
                    memoria libres consecutivas que proporcionen un
                    espacio suficiente para albergar el tamaño de un
                    segmento de código o datos del proceso (según indique
                    tipo_seg), asignado al número de zócalo que se pasa
                    por parámetro;
                    la rutina devuelve la primera dirección del espacio
                    reservado; en el caso de que no quede un espacio de
                    memoria consecutivo del tamaño requerido, devuelve
                    cero */
void * _gm_reservarMem(int z, int tam, unsigned char tipo_seg)
```

Esta rutina será invocada por la función `_gm_cargarPrograma()`, ya que es la que sabe cuánto espacio de memoria necesitan los segmentos a cargar. Es por este motivo que a la función `_gm_cargarPrograma()` se le ha añadido el parámetro de número de zócalo, que no estaba en su definición para la fase 1. Por lo tanto, la rutina `_gm_reservarMem()` reservará un conjunto consecutivo de franjas libres de 32 bytes, que proporcionen suficiente cantidad de memoria para alojar el contenido de cada segmento del programa.

Hay que tener precaución porque se puede dar el caso de que **exista memoria para el primer segmento pero no para el segundo**, en cuyo caso se tendría que liberar la memoria asignada al primer segmento al no poder cargar todo el programa íntegro.

Para liberar toda la memoria asignada a un zócalo se debe implementar la siguiente rutina:

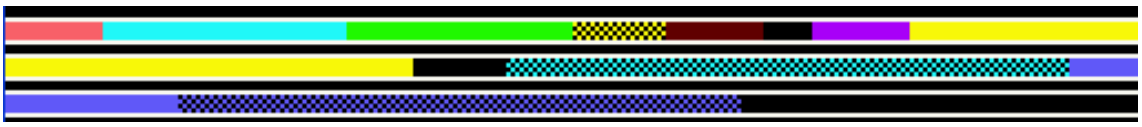
```
/* _gm_liberarMem: rutina para liberar todas las franjas de memoria
                   asignadas al proceso del zócalo indicado por
                   parámetro */
void _gm_liberarMem(int z)
```

Esta rutina de liberar memoria se llamará desde la función de cargar el programa, en el caso de que no exista espacio suficiente para alguno de los segmentos del programa, o cuando un proceso termina o se mata al proceso.

Hay que tener en cuenta que los conjuntos de franjas asignadas a los dos posibles segmentos de un programa no tienen por qué ser consecutivos, aunque sí lo deben ser todas las franjas asignadas a cada segmento de programa.

3.4 Representación gráfica de la ocupación de memoria

Para poder visualizar la ocupación de la memoria de los procesos de usuario, se propone gestionar un gráfico en la parte inferior de la pantalla secundaria:



Se generarán 96 baldosas nuevas, inicialmente iguales a la baldosa 119  . Estas nuevas baldosas se colocarán en la memoria gráfica para píxeles de baldosas, a partir del *offset* 512*64. Los índices de estas baldosas se guardarán en las posiciones apropiadas del mapa del fondo 0 del procesador gráfico secundario, de manera que se visualizarán en las 3 últimas filas de la pantalla.

Estas inicializaciones gráficas se realizan invocando la siguiente rutina, ya implementada en el fichero `garlic_itcm_sys.s`:

```
/* _gs_iniGrafB: inicializa el procesador gráfico B para GARLIC 2.0
*/
extern void _gs_iniGrafB();
```

En cada baldosa se pueden pintar hasta 8 líneas verticales de 4 píxeles de altura, lo cual da un total de 768 líneas, una por cada franja de memoria de 32 bytes, generando así la representación de la ocupación de los 24 Kbytes de memoria disponible para programas de usuario.

Para distinguir visualmente los segmentos de memoria de los diferentes procesos se utilizará un color diferente para cada zócalo. En el fichero `garlic_itcm_sys.s` se encuentra definido un vector de nombre `_gs_colZoc[]` que contiene 16 índices de paleta, con el fin de asignar un color específico a cada zócalo (la posición 0 del vector no se utilizará).

Además, se propone utilizar dos patrones de píxeles diferentes para distinguir el tipo de segmento reservado, un patrón sólido para los segmentos de código (todos los píxeles del mismo color) y un patrón ajedrezado para los segmentos de datos (un píxel de color y otro en negro).

Para conseguir pintar las franjas consecutivas correspondientes a cada segmento, las rutinas `_gm_reservarMem()` y `_gm_liberarMem()` invocarán a otra rutina de nombre `_gs_pintarFranjas()`, que **ya se encuentra implementada** dentro del fichero `garlic_itcm_sys.s`:

```
/* _gs_pintarFranjas: rutina para pintar las líneas verticales
                correspondientes a un conjunto de franjas
                consecutivas de memoria asignadas a un segmento
                (de código o datos) del zócalo indicado por
                parámetro.
     Parámetros:
          zocalo  -> el zócalo que reserva la memoria (0 para borrar)
          index_ini   ->  el índice inicial de las franjas
          num_franjas ->  el número de franjas a pintar
          tipo_seg    ->  el tipo de segmento reservado
                          (0 -> código, 1 -> datos)
*/
extern void _gs_pintarFranjas(unsigned char zocalo,
                              unsigned short index_ini,
                              unsigned short num_franjas,
                              unsigned char tipo_seg);
```

3.5 Lista de programas

Para que el programa principal pueda disponer de la lista de los programas que se encuentran disponibles en el directorio `/Programas/` del sistema de ficheros ***Nitrofiles***, se tiene que implementar la siguiente función dentro del fichero `garlic_mem.c`:

```
/* _gm_listaProgs: devuelve una lista con los nombres en clave de
          todos los programas que encuentra en el directorio
          "Programas".
          Se considera que un fichero es un programa si su nombre
          tiene 8 caracteres y termina con ".elf"; se devuelven solo
          los 4 primeros caracteres (nombre en clave).
          El resultado es un vector de strings (paso por referencia)
          y el número de programas detectados */
extern int _gm_listaProgs(char* progs[]);
```

Hay que observar que los nombres de los programas se devuelven **por referencia**, es decir, llenando las posiciones del vector `progs[]` con punteros a *strings* de caracteres que contendrán el nombre en clave de cada programa de usuario.

3.6 Representación de la pila y el estado de los procesos

Para completar la representación de información de la tabla de procesos, el programador `progM` deberá programar la rutina `_gm_rsiTIMER1()` dentro del fichero `garlic_itcm_mem.s`, la cual se invocará por interrupciones periódicas para actualizar el contenido de las columnas `Pi` y `E`, que indicarán la cantidad de posiciones ocupadas de la pila y el estado de cada proceso para todos los zócalos activos:

```
	.global _gm_rsiTIMER1
	@; Rutina de Servicio de Interrupción (RSI) para actualizar la
	@; representación de la pila y el estado de los procesos activos.
_gm_rsiTIMER1:
	push {lr}

	pop {pc}
```

Para representar la ocupación de la pila hay que consultar el campo `SP` correspondiente del vector `_gd_pcbs[]` y calcular cuantos *words* están ocupados en cada momento respecto a la dirección de inicio de la pila. Con este número habrá que codificar un gráfico con dos caracteres consecutivos del siguiente subconjunto de baldosas:

Con dichos caracteres gráficos se pueden generar hasta 17 niveles de ocupación, o sea, que habrá que representar el número de posiciones ocupadas proporcionalmente respecto al máximo (128 *words*).

Para representar la ocupación de todas las pilas se puede invocar a la rutina `_gs_representarPilas()`, que **ya se encuentra implementada** dentro del fichero `garlic_itcm_sys.s`:

```
/* _gs_representarPilas: rutina para representar gráficamente la
                         ocupación de las pilas de los procesos de
                         usuario, además de la pila del proceso de
                         control del sistema operativo, sobre la tabla
                         de control de procesos. */
extern void _gs_representarPilas();
```

Por último, el estado del proceso se representará con las letras '**R**' para *Run*, '**Y**' para *readY* y '**B**' para *Blocked* (retardado). Las letras se escribirán en color blanco excepto la '**R**', que se debe escribir en color azul (ver baldosas de colores en memoria de vídeo para fondos de procesador secundario).

Si se implementan las tareas de teclado `progT`, los procesos que estén bloqueados esperando un *string* se deberán marcar con una '**K**' (*Keyboarding*), y los procesos retardados se pueden representar con una '**D**' (*Delayed*).

3.7 Programa principal para progM

Dentro de la rama `progM` del respositorio `Garlic_OS_2.git` se proporciona un programa principal `main.c` específicamente preparado para poder probar las rutinas relativas a la gestión de memoria sin tener que depender del trabajo de los otros programadores. Sin embargo, este programa principal requiere disponer de una versión operativa de las funciones y rutinas de los roles `progP` y `progG` para la fase 1.

```
/*------------------------------------------------------------------
      "main.c" : fase 2 / progM
------------------------------------------------------------------*/
#include <nds.h>
#include "garlic_system.h"
…
                                        // frecuencia TIMER1 = 7 Hz
const short divFreq1 = -33513982/(1024*7);

/* Función para gestionar los sincronismos */
void gestionSincronismos()
{
   int i, mask;
```

```
   if (_gd_sincMain & 0xFFFE)    // si hay algun sincronismo pendiente
   {
       mask = 2;
       for (i = 1; i <= 15; i++)
       {
            if (_gd_sincMain & mask)
            {     // liberar la memoria del proceso terminado
                  _gm_liberarMem(i);
                  _gg_escribir("*proceso %d terminado\n", i, 0, 0);
                  _gs_dibujarTabla();
                  _gd_sincMain &= ~mask;      // poner bit a cero
            }
            mask <<= 1;
       }
   }
}
```

La función de gestión de sincronismos permite detectar qué procesos han terminado su ejecución, a través del bit correspondiente de la variable global `_gd_sincMain`, que se activa en la nueva versión de la rutina `_gp_terminarProc()` que se proporciona en el fichero `garlic_itcm_proc.s` de la rama `progP` del repositorio `Garlic_OS_2.git`.

ATENCIÓN: para que funcione la detección del sincronismo anterior es imprescindible actualizar la rutina `_gp_terminarProc()` de la fase 1; en el caso de usar el fichero `garlic_itcm_proc.sobj` de los `complementary_files_1` **no** será posible realizar esta actualización, por lo que la liberación de la memoria de un programa que termina se tendrá que gestionar de otro modo (desde el programa de control).

```
//------------------------------------------------------------------
void inicializarSistema() {
//------------------------------------------------------------------
      _gg_iniGrafA();       // inicializar procesadores gráficos
      _gs_iniGrafB();
      _gs_dibujarTabla();
      ...
      if (!_gm_initFS()) {
            _gg_escribir("\nERROR: ¡no se puede inicializar el sistema
de ficheros!\n", 0, 0, 0);
            exit(0);
      }
```

```
	irqInitHandler(_gp_IntrMain);		// inicializar IRQs
	irqSet(IRQ_VBLANK, _gp_rsiVBL);
	irqEnable(IRQ_VBLANK);

	irqSet(IRQ_TIMER1, _gm_rsiTIMER1);
	irqEnable(IRQ_TIMER1);		// instalar la RSI para el TIMER1
	TIMER1_DATA = divFreq1;
	TIMER1_CR = 0xC3;		// Timer Start | Timer IRQ Enabled

	REG_IME = IME_ENABLE;		// activar las interr. en general
}
```

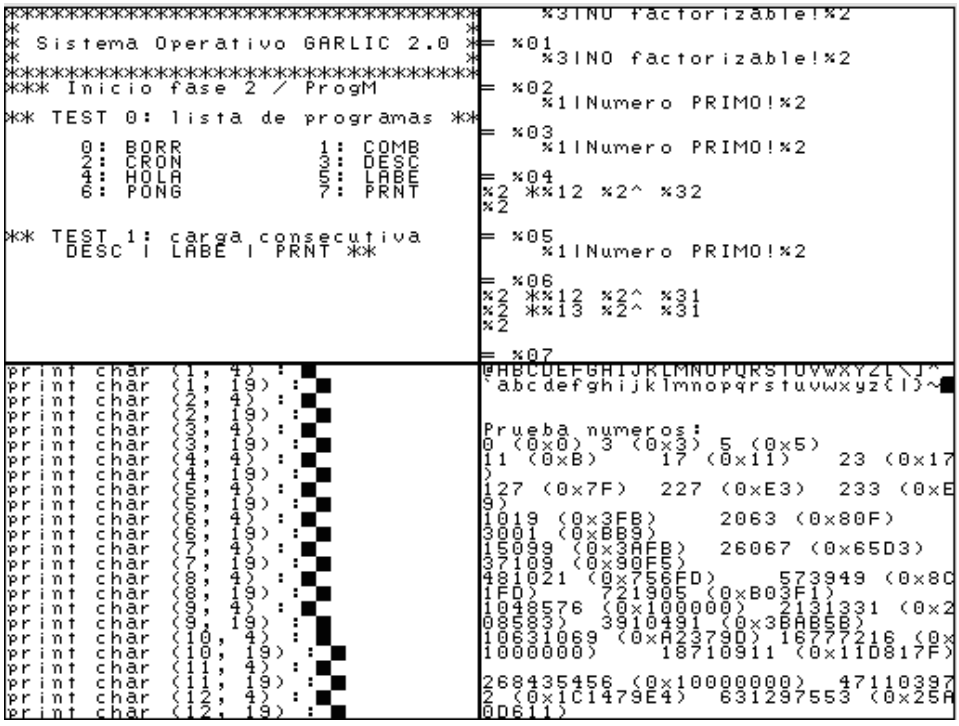
En la función de inicialización del sistema se inicializan ambos procesadores gráficos, la tabla de procesos, el sistema de ficheros y las interrupciones de *VBlank* para la multiplexación de procesos, y del *timer* 1 para la representación de la ocupación de la pila y el estado de los procesos.

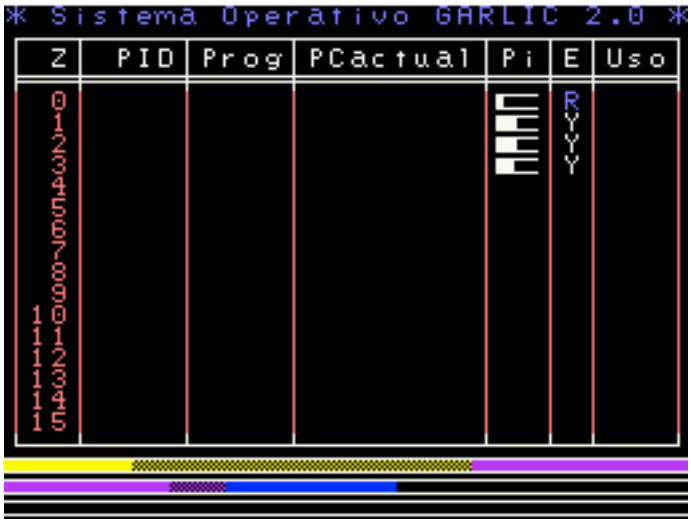
```
//---------------------------------------------------------------------
int main(int argc, char **argv) {
//---------------------------------------------------------------------
	inicializarSistema();
	...
	_gg_escribir("*** Inicio fase 2_M\n", 0, 0, 0);

	if (test0())
	{
		if (test1())
			test2();
	}
	_gg_escribir("\n*** Final fase 2 / ProgM\n", 0, 0, 0);
	while (1) _gp_WaitForVBlank();
	return 0;
}
```

En general, este programa principal realiza tres llamadas a funciones auxiliares que efectúan tres juegos de pruebas encadenados:

- `test0()`: llama a la función `_gm_cargarProgamas()` y devuelve 1 si detecta los programas de usuario DESC, LABE, PONG y PRNT, que son los programas que se utilizarán en los dos tests siguientes,

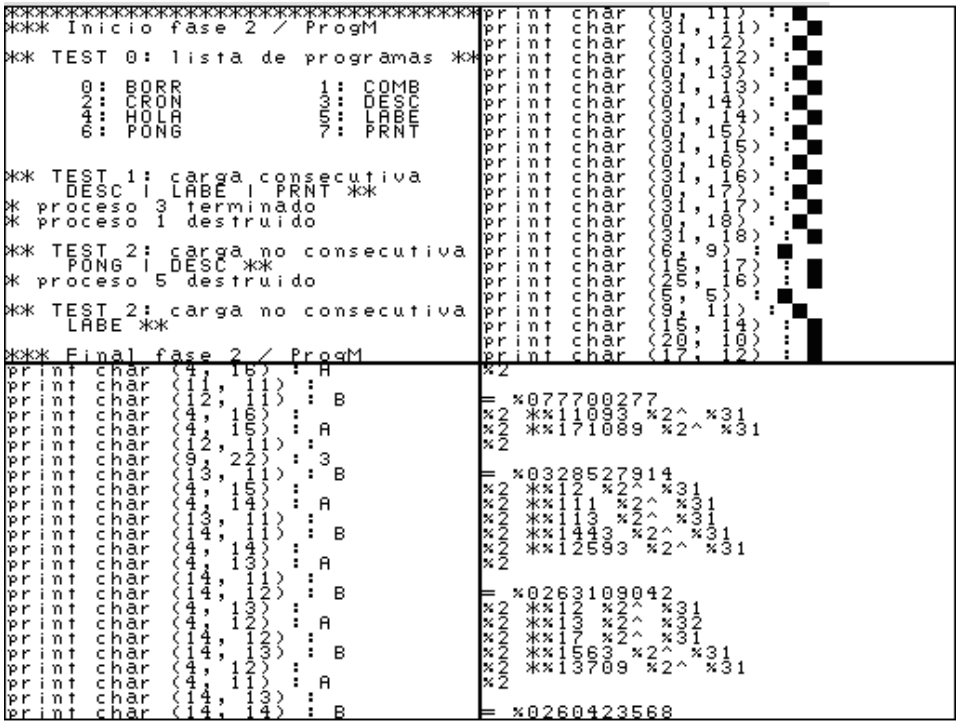
- `test1()`: carga los programas de usuario DESC, LABE y PRNT de forma consecutiva en memoria. El resultado que se observará en las pantallas será similar al siguiente:

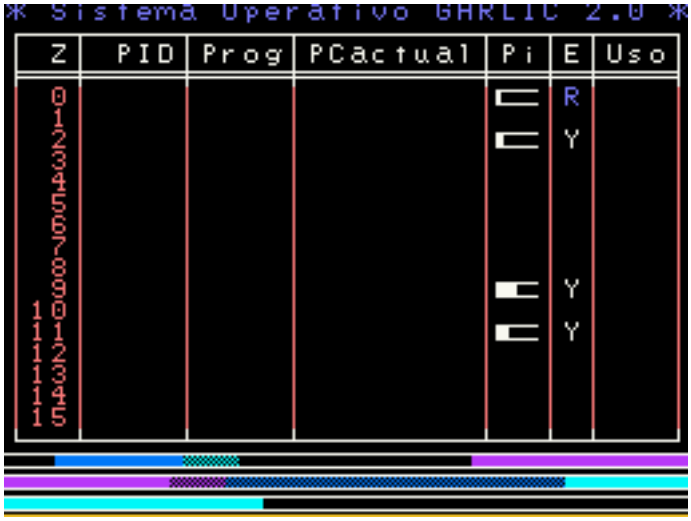




- `test2()`: el test anterior espera a que el programa PRNT acabe su ejecución y después elimina el programa DESC, dejando el programa LABE en ejecución; el test 2 carga primero el programa PONG para ocupar las primeras franjas de memoria, y después carga un nuevo DESC que tendrá el segmento de datos separado del segmento de código, puesto que el segmento de datos no cabrá en el espacio disponible a continuación (hasta la primera franja del LABE en ejecución); después eliminará el PONG y cargará un nuevo LABE, el cual ubicará su segmento de código en las franjas libres del final, mientras que su

segmento de datos se podrá cargar en franjas libres del inicio por ser más pequeño. El resultado que se observará en las pantallas será similar al siguiente:





Por último, hay que destacar que los programas PONG y LABE no podrían ejecutarse de no ser por la versión específica de la rutina `_ga_printchar()` que se encuentra dentro del fichero `garlic_itcm_api.s`, ya que todavía no se puede llamar a la versión definitiva de la rutina `_gg_escribirCar()` que debe implementar el programador `progG`.

4 Tareas de gestión de los gráficos (progG)

4.1 Inicialización del entorno gráfico

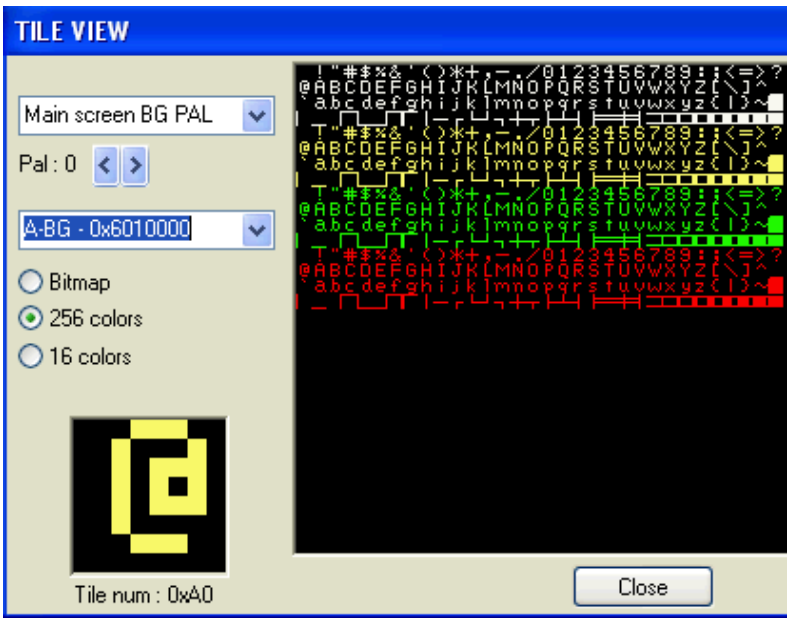
En esta segunda fase se debe modificar la función `_gg_iniGrafA()` para que los fondos 2 y 3 del procesador gráfico principal utilicen un tamaño de 1024x1024 píxeles, necesario para la representación de 16 ventanas de texto.

Para conseguir caracteres en color habrá que generar nuevas baldosas. Esto se conseguirá **copiando tres veces** las 128 baldosas básicas, descomprimidas de la variable `garlic_fontTiles[]` del fichero `garlic_font.s`, a continuación de la ubicación de las baldosas originales de color blanco, y **cambiando el valor de los píxeles de todas las baldosas** (originalmente `0xFF`) por otro valor que corresponda al índice adecuado de la paleta de colores.

Para que todas las prácticas utilicen la misma gama de colores, el fichero `garlic_graf.c` incorpora un vector `char_colors[]` con índices de la paleta definida en `garlic_fontPal[]` del fichero `garlic_font.s`, para representar los colores amarillo, verde y rojo:

```
const unsigned int char_colors[] = {240, 96, 64};
```

Si la creación de estas baldosas es correcta, desde una visualización con la herramienta "**View Tiles**" del *DeSmuME* se debería observar el siguiente conjunto de baldosas (la posición inicial puede variar):



4.2 La función de escribir con formato

La función del API `GARLIC_printf()` se modifica en la fase 2 para incorporar color en el texto, aunque su declaración como función externa en `GARLIC_API.h` no varía respecto a la fase 1:

```
	/* GARLIC_printf: escribe string en la ventana del proceso actual
(en el comentario del fichero GARLIC_API.h se explica el uso del color)
*/
extern void GARLIC_printf(char * format, ...);
```

El color del texto será un valor entre 0 y 3 que se almacenará en los 4 bits altos del campo `pControl` de la estructura `garlicWBUF`, lo cual supondrá una ligera modificación en el tratamiento de los bits de dicho campo:

```
typedef struct                  // Estructura del buffer de una ventana
{                               // (WBUF: Window BUFfer)
	int pControl;           // control de escritura en ventana
                                // 4 bits altos: cód. color actual (0..3)
                                // 12 bits medios: número de línea (0..23)
                                // 16 bits bajos: car. pendientes (0..32)
	short pChars[32];       // vector 32 car. pendientes de escritura
                                // 16 bits por entrada, indicando núm. de
                                // baldosa corresp. al caracter+color
} PACKED garlicWBUF;
```

El valor del código de color actual indicará en qué color se deben imprimir los caracteres en cada momento, utilizando el 0 para el blanco, el 1 para el amarillo, el 2 para el verde y el 3 para el rojo.

Según el color actual y el código ASCII de cada carácter, dentro del buffer de caracteres pendientes `_gd_wbfs[v].pChars[32]` se almacenará el índice de baldosa correspondiente al gráfico del carácter con el color requerido. Para poder almacenar dichos índices, el buffer de caracteres ya no será de bytes, sino que se declarará de tipo *short* (16 bits por posición).

Para cambiar el color actual se introducen nuevas marcas de formato: `%0`, `%1`, `%2` y `%3`. Cuando una de estas marcas aparezca en el *string* de formato, todos los caracteres que se tengan que escribir a partir de ese momento lo harán con el color indicado en la marca. Este color también se aplicará a los caracteres de los siguientes *strings* que se envíen

con la función `GARLIC_printf()`, mientras no se fije un nuevo color con otra marca de formato de color.

La descripción de `_gg_escribir()` se expande para informar de esta nueva funcionalidad:

```
/* _gg_escribir: escribe una cadena de caracteres en la ventana
                 indicada;
     Parámetros:
                 formato -> string de formato:
                            admite '\n' (salto de línea), '\t'
                            (tabulador, 4 espacios) y códigos entre
                            32 y 159 (los 32 últimos son caracteres
                            gráficos), además de marcas de formato
                            %c, %d, %x y %s (máx. 2 marcas por string)
                            y las marcas de cambio de color actual
                            %0 (blanco), %1 (amarillo), %2 (verde)
                            y %3 (rojo)
                 val1  ->   valor a sustituir en la primera marca de
                            formato, si existe
                 val2  ->   valor a sustituir en la segunda marca de
                            formato, si existe
                 ventana -> número de ventana (0..3)
*/
extern void _gg_escribir(char *formato, unsigned int val1,
                                 unsigned int val2, int ventana);
```

Como la característica del color solo afecta al código de baldosa y no a la conversión de los valores de los parámetros `val1` y `val2` en sus respectivos formatos, se sugiere **no** modificar la rutina `_gg_procesarFormato()` para el tratamiento de las marcas de color, sino que esta rutina simplemente copiará dichas marcas de color literalmente en el *string* resultante, relegando su gestión a la rutina `_gg_escribir()`.

4.3 Las funciones de escritura directa a ventana

Se han añadido dos nuevas funciones al API para escribir caracteres en posiciones concretas de la ventana, que invocarán las siguientes rutinas a implementar dentro del fichero `garlic_itcm_graf.s`:

```
/* _gg_escribirCar: escribe un carácter (baldosa) en la posición de
      la ventana indicada, con un color concreto;
     Parámetros:
          vx    ->    coordenada x de ventana (0..31)
          vy    ->    coordenada y de ventana (0..23)
          c     ->    código del carácter: número de baldosa (0..127)
          color ->    número de color del texto (0..3)
          ventana->   número de ventana (0..15) */
extern void _gg_escribirCar(int vx, int vy, char c, int color,
                                                    int ventana);


/* _gg_escribirMat: escribe una matriz de 8x8 caracteres a partir de
      una posición de la ventana indicada, con un color concreto;
     Parámetros:
          vx    ->    coordenada x inicial de ventana (0..31)
          vy    ->    coordenada y inicial de ventana (0..23)
          m     ->    matriz 8x8 de códigos ASCII
          color ->    número de color del texto (0..3)
          ventana->   número de ventana (0..15) */
extern void _gg_escribirMat(int vx, int vy, char m[][8], int color,
                                                    int ventana);
```

Hay que tener en cuenta que el quinto parámetro (`int` ventana) **se pasará por la pila**, no por registro. Consultar el apartado 3.2 sobre la rutina de reubicación, donde se muestra un gráfico que representa el acceso al quinto parámetro a través de la pila.

Estas nuevas funciones **no** se sincronizan con el retroceso vertical, de modo que internamente no llamarán a la rutina `_gp_WaitForVBlank()`, tal como sí debe hacer la función `_gg_escribir()`. El motivo de la no sincronización es permitir que un proceso pueda llamar múltiples veces a estas rutinas sin perder la CPU. Como consecuencia, el programa que ejecutará el proceso deberá sincronizarse con el *Vertical Blank* antes de realizar dichas llamadas a las rutinas de acceso a las posiciones de la ventana, por ejemplo, utilizando una llamada a `GARLIC_delay(0)`. De otro modo se podrían producir problemas de visualización.

4.4 Información de la tabla de procesos

Para gestionar la representación de la tabla de procesos de la pantalla inferior de la NDS hay que programar la siguiente rutina en lenguaje ensamblador, dentro del fichero `garlic_itcm_graf.s`:

```
/* _gg_escribirLineaTabla: escribe los campos básicos de una línea de
           la tabla de procesos, correspondiente al número de zócalo
           que se pasa por parámetro con el color especificado;
           los campos a escribir son: número de zócalo, PID y
           nombre clave del proceso (keyName) */
extern void _gg_escribirLineaTabla(int z, int color);
```

Esta rutina escribirá la línea de la tabla correspondiente al número de zócalo. El parámetro de color determinará tanto el color del texto como el color de los separadores.

Además, hay que programar la rutina `_gg_rsiTIMER2()`, también dentro del fichero `garlic_itcm_graf.s`, la cual se invocará por interrupciones periódicas para actualizar el contenido de la columna `PCactual`, la cual indicará el valor del PC actual de cada proceso para todos los zócalos activos.

```
      .global _gg_rsiTIMER2
      @; Rutina de Servicio de Interrupción (RSI) para actualizar la
      @; representación del PC actual.
_gg_rsiTIMER2:
      push {lr}

      pop {pc}
```

Para escribir el valor del PC de cada proceso habrá que consultar el campo correspondiente del vector de PCBs e invocar a la rutina `_gs_num2str_hex()`, que se encuentra implementada dentro del fichero `garlic_itcm_sys.s`, la cual permitirá convertir la dirección de memoria en un *string* con formato hexadecimal.

4.5 Programa principal para progG

Dentro de la rama `progG` del respositorio `Garlic_OS_2.git` se proporciona un programa principal `main.c` específicamente preparado para poder probar las rutinas relativas a la gestión de gráficos sin tener que depender del trabajo de los otros programadores. Sin embargo, este programa principal requiere disponer de una versión operativa de las funciones y rutinas de los roles `progP` y `progM` para la fase 1.

```
/*------------------------------------------------------------------
     "main.c" : fase 2 / progG
------------------------------------------------------------------*/
#include <nds.h>
#include <stdlib.h>

#include "garlic_system.h"
                                        // frecuencia TIMER2 = 4 Hz
const short divFreq2 = -33513982/(1024*4);
...

/* Función para permitir seleccionar un programa entre los ficheros
ELF disponibles, así como un argumento para el programa (0..3) */
void seleccionarPrograma()
{
     ...
     i = 1;                // buscar si hay otro proceso en marcha
     while ((i < 16) && (_gd_pcbs[i].PID == 0))
     {
          i++;
     }
     if (i < 16)           // en caso de encontrar otro proceso activo
     {
          _gd_pcbs[i].PID = 0;  // liberar su PCB
          _gd_nReady = 0;       // eliminar proceso de cola de READY
          _gg_escribir("* %3%d%0: proceso destruido\n", i, 0, 0);
          _gg_escribirLineaTabla(i, (i==_gi_za ? 2 : 3));
          if (i != _gi_za) // si no se trata del propio zócalo actual
               _gg_generarMarco(i, 3);
     }
     _gs_borrarVentana(_gi_za, 1);
     _gg_escribir("%1*** Seleccionar programa :\n", 0, 0, _gi_za);
     ind_prog = escogerOpcion((char **) progs, num_progs);
     _gg_escribir("%1*** seleccionar argumento :\n", 0, 0, _gi_za);
     argumento = escogerOpcion((char **) argumentosDisponibles, 4);
```

```
      start = _gm_cargarPrograma((char *) progs[ind_prog]);
      if (start)
      {
         _gp_crearProc(start, _gi_za, (char *) progs[ind_prog],
                                                      argumento);
         _gg_escribir("%2* %d:%s.elf", _gi_za,
                                 (unsigned int) progs[ind_prog], 0);
         _gg_escribir(" (%d)\n", argumento, 0, 0);
         _gg_escribirLineaTabla(_gi_za, 2);
      }
}
```

La función `seleccionarPrograma()` permite cargar en el zócalo `_gi_za` (zócalo en foco) un nuevo proceso con el programa seleccionado con la variable global `progs[ind_prog]`. La variable global `_gi_za` está gestionada por la interfaz de usuario, que ya está implementada en el fichero `garlic_itcm_ui.s`.

Sin embargo, antes de cargar un nuevo proceso, la función verifica la posible existencia de otro proceso de usuario en marcha y, en caso afirmativo, procede a eliminarlo. Como todavía no podemos contar con la implementación de la rutina `_gp_matarProc()` (`progP`), se realiza una eliminación *ad hoc* del proceso, aunque este procedimiento no será válido para la versión final del sistema operativo.

```
//--------------------------------------------------------------------
void inicializarSistema() {
//--------------------------------------------------------------------
      ...
      _gg_iniGrafA();       // inicializar gráficos
      _gs_iniGrafB();
      _gs_dibujarTabla();
      _gi_redibujarZocalo(1);// marca tabla de zócalos con el proceso
      ...                    // del S.O. seleccionado (en verde)
      irqInitHandler(_gp_IntrMain);         // inicializar IRQs
      irqSet(IRQ_VBLANK, _gp_rsiVBL);
      irqEnable(IRQ_VBLANK);

      irqSet(IRQ_TIMER2, _gg_rsiTIMER2);
      irqEnable(IRQ_TIMER2);      // instalar la RSI para el TIMER2
      TIMER2_DATA = divFreq2;
      TIMER2_CR = 0xC3;           // Timer Start | Timer IRQ Enabled

      irqSet(IRQ_VCOUNT, _gi_movimientoVentanas);
      REG_DISPSTAT |= 0xE620;     // fijar línea VCOUNT a 230 y
      irqEnable(IRQ_VCOUNT);      // activar interrupción de VCOUNT
```

```
	REG_IME = IME_ENABLE;		// activar las interr. en general
}
```

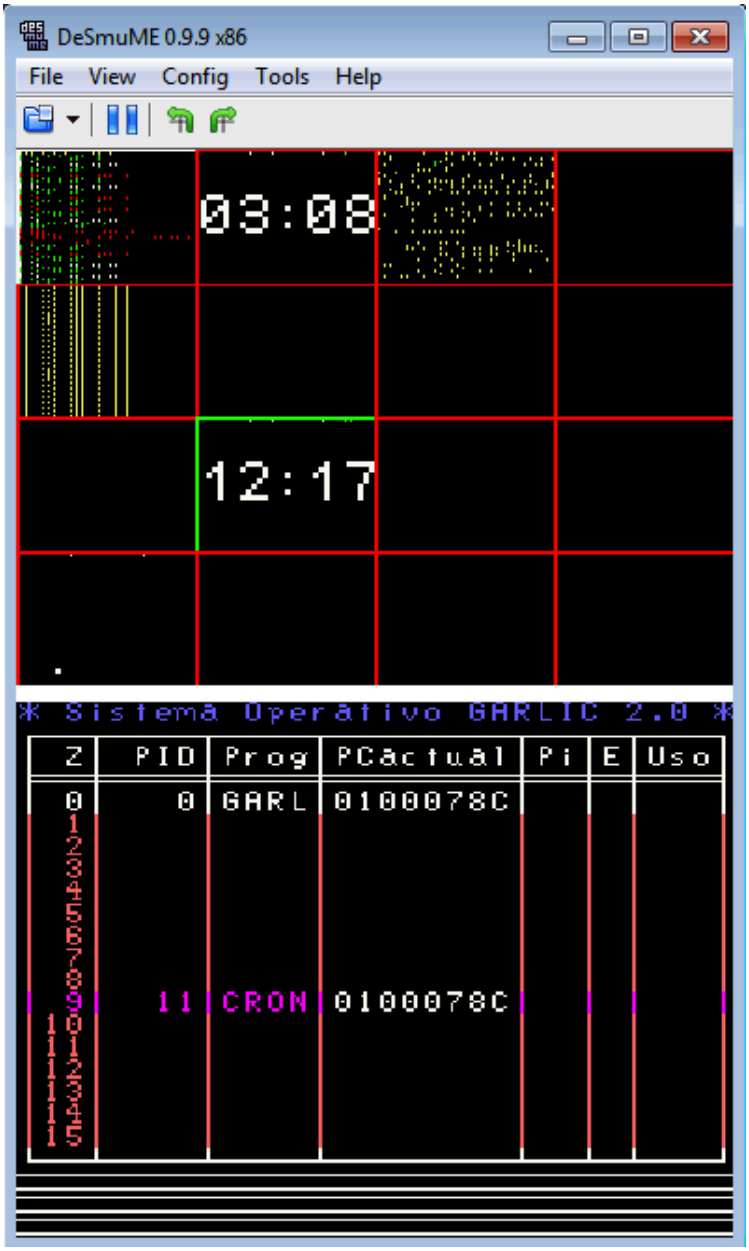
En la función de inicialización del sistema se inicializan ambos procesadores gráficos, la tabla de procesos, el sistema de ficheros y las interrupciones de *VBlank* para la multiplexación de procesos, y del *timer* 2 para la representación del PC de los procesos activos a una frecuencia de 4 Hz.

Además, se instala la rutina `_gi_movimientoVentanas()`, implementada en el fichero `garlic_itcm_ui.s`, que permite la gestión de la interfaz de usuario desde la RSI de *Vertical COUNT*, que se activará cada vez que los procesadores gráficos estén representando la fila 230 del *display*; de hecho, el *display* solo dispone de 192 filas, las filas de la 192 a la 262 son ficticias y solo sirven para generar el retardo del VBL.

```
//-------------------------------------------------------------------
int main(int argc, char **argv) {
//-------------------------------------------------------------------
	...
	inicializarSistema();

	_gg_escribir("%1**********************************", 0, 0, 0);
	_gg_escribir("%1*                                 *", 0, 0, 0);
	_gg_escribir("%1* Sistema Operativo GARLIC 2.0 *", 0, 0, 0);
	_gg_escribir("%1*                                 *", 0, 0, 0);
	_gg_escribir("%1**********************************", 0, 0, 0);
	_gg_escribir("%1*** Inicio fase 2 / ProgG\n", 0, 0, 0);

	while (1)			// bucle infinito
	{
		scanKeys();
		key = keysDown(); // leer botones y controlar la interfaz
		if (key != 0)					// de usuario
		{	_gi_controlInterfaz(key);
			if ((key == KEY_START) && (_gi_za != 0))
				seleccionarPrograma();
		}
		gestionSincronismos();
		_gp_WaitForVBlank();  // retardo del proceso de sistema
	}
	return 0;
}
```

La función principal es muy corta porque el trabajo se realiza básicamente en las rutinas/funciones `_gi_controlInterfaz()`, `seleccionarPrograma()` y `gestionSincronismos()`.

En la imagen siguiente se muestra un ejemplo de visualización de la ejecución consecutiva de varios de procesos con el programa de test de la rama `progG`: en la pantalla superior se muestran 16 ventanas de texto (zoom ¼) y en la pantalla inferior está la tabla de procesos.



En las ventanas de texto se observa la salida de procesos que se están ejecutando en ese momento (proceso de control del sistema en zócalo 0, programa CRON en zócalo 9), pero también se puede observar la salida de otros procesos que se habían ejecutado anteriormente.

En esta versión del programador `progG` solo se puede cargar y ejecutar un programa de usuario a la vez. Aunque sería factible cargar varios programas de usuario simultáneamente, se propone esperar a disponer del código del programador `progM` de la fase 2 para realizar una gestión de la memoria disponible que evite un desbordamiento de dicha memoria. Por este motivo, cuando se invoca la ejecución de un nuevo proceso de usuario, el programa principal propuesto se encarga de eliminar cualquier otro proceso de usuario.

5 Tareas de gestión del teclado (progT)

5.1 Ampliación del API del sistema

En esta segunda fase hay que modificar el conjunto de funciones del API del sistema operativo, tal como se explica en el apartado 1.4 de este manual.

Para conseguir la máxima compatibilidad entre todos los programas de GarlicOS 2.0, habrá que guardar las direcciones de las rutinas API de teclado al final del vector de direcciones definido en `garlic_vectors.s`:

```
.section .vectors,"a",%note

APIVector:              @; Vector de direcciones de rutinas del API
      .word _ga_pid     @; (código de rutinas en "garlic_itcm_api.s")
      .word _ga_random
      .word _ga_divmod
      .word _ga_divmodL
      .word _ga_printf
      .word _ga_printchar
      .word _ga_printmat
      .word _ga_delay
      .word _ga_clear
      .word _ga_getstring
      .word _ga_getxybuttons
```

Las dos últimas entradas del vector corresponderán a la función de la fase 1 `GARLIC_getstring()` y a una nueva función `GARLIC_getXYbuttons()` que se explicará más adelante (apartado 5.4).

Por lo tanto, será necesaria la consecuente modificación de los ficheros `GARLIC_API.h` y `GARLIC_API.s`, además de la **recompilación** de los programas para GarlicOS 1.0 que llaman a la función `GARLIC_getstring()`, ya que la posición en el vector de la dirección de `_ga_getstring()` será diferente con la nueva configuración del vector.

Sin embargo, el resto de los programas que no realicen llamadas a la función `GARLIC_getstring()` no será necesario recompilarlos para poder ejecutarlos con la nueva disposición del `APIvector`.

5.2 Modificación de _gt_getstring()

Aunque la comunicación con la función `GARLIC_getstring()` será idéntica a la fase 1, en esta segunda fase hay que cambiar el modo en que la rutina `_gt_getstring()` bloquea los procesos que están esperando un *string*.

Concretamente, en vez de utilizar un bucle de encuesta periódica, habrá que desbancar el proceso que invoca la función (en *Run*), salvar su contexto y ponerlo en la cola de procesos del teclado `_gd_kbwait[]`, pero **no** en la cola de *Ready*.

La forma de desbancar un proceso que tenemos disponible es mediante la RSI del *Vertical Blank* `_gp_rsiVBL()`, que, a su vez, invoca a la rutina `_gp_salvarProc()`. Para evitar que esta última rutina guarde el número de zócalo en la cola de *Ready*, habrá que especificar otro código especial sobre la variable `_gd_pidz`, de modo similar a cómo se bloquean los procesos que invocan a `GARLIC_delay()` (ver apartado 2.1).

Habrá que actualizar las rutinas `_gp_numProc()` y `_gp_matarProc()` para que tengan en cuenta el nuevo posible estado de los procesos, así como la rutina `_gm_rsiTIMER1()` que visualiza el estado actual de los procesos activos:

- *RUN* ('**R**'): el proceso que se está ejecutando en cada instante,
- *READY* ('**Y**'): procesos que están preparados para pasar a RUN, cuando les toque el turno,
- *DELAYED* ('**D**'): procesos bloqueados a la espera de que expire su tiempo de retardo,
- *KEYBOARDING* ('**K**'): procesos bloqueados a la espera de que se les pase un *string* por teclado.

Por supuesto, será la RSI que detecta la finalización de la introducción del *string* por teclado la que se encargue de sacar el proceso de la cola de bloqueados en teclado y ponerlo en la cola de *Ready*.

Con este nuevo enfoque, la variable global de la fase 1 `_gd_kbsignal` ya no tiene sentido, y los procesos que invoquen a `GARLIC_getstring()` **no** gastarán intervalos de ejecución mientras estén bloqueados esperando la entrada de información.

5.3 Modificación de la interfaz del teclado

Para conseguir una introducción de texto más ágil de la que se implementó en la fase 1, se propone modificar la interfaz de usuario de modo que se visualice un teclado virtual y se detecten las pulsaciones sobre la pantalla táctil para activar las teclas.

Concretamente, se propone construir dos mapas de baldosas sobre los fondos 1 y 2 del procesador gráfico secundario (el fondo 0 estará reservado para la tabla de procesos). Con estos dos nuevos fondos y las baldosas de colores generadas por la rutina `_gs_iniGrafB()` podemos generar el siguiente gráfico que representa una disposición de teclado de tipo QWERTY:

Las figuras anteriores representan dos estados del teclado, según se haya pulsado o no la tecla **CAPS**.

El teclado se representará gráficamente sobre uno de los fondos, con baldosas sólidas de color azul para la posición de las teclas y de color salmón para el relleno. Además, la zona para representar los caracteres del *string* también serán baldosas sólidas azules y blancas, para distinguir entre la parte de texto introducido y la parte libre. No podemos utilizar espacios en blanco porqué se vería el contenido de la tabla de procesos (el cero es el color transparente).

Al detectar la pulsación de la pantalla táctil sobre el área de una tecla, la baldosa de fondo deberá cambiar de color (de azul a magenta) para indicar al usuario la detección de su pulsación. Esta indicación se mantendrá mientras dure la pulsación, aunque solo sirva para introducir un carácter (opcionalmente, se podría implementar un *autorepeat*). Además, al pulsar la tecla **CAPS**, esta se quedará activada (color magenta) incluso después de liberar la pulsación, y no se desactivará hasta que se vuelva a pulsar.

En el otro fondo (más prioritario) representaremos el gráfico de cada tecla con las baldosas de color blanco correspondiente, así como los caracteres del texto introducido sobre el recuadro destinado a mostrar el *string*. Cuando se pulse la tecla **CAPS** se cambiará el conjunto de gráficos de teclas.

Adicionalmente se puede usar el fondo 3 para representar los bordes de la zona del *string*, así como la posición del cursor.

Como en la fase 1, el teclado se deberá mostrar cuando exista al menos un proceso esperando información, y se deberá ocultar cuando se hayan servido todas las peticiones. Opcionalmente, se puede añadir una tecla virtual de control para ocultar y mostrar de nuevo el teclado a voluntad del usuario.

Por último, para conseguir detectar las pulsaciones sobre la pantalla táctil habrá que sustituir la RSI del teclado por una RSI del mecanismo **IPC-FIFO** de comunicación entre los procesadores ARM9 y ARM7, lo cual requiere también programar específicamente el procesador ARM7 para que envíe un paquete de información con las coordenadas X-Y de pantalla cada vez que haya una pulsación.

5.4 Nueva función GARLIC_getXYbuttons()

La llamada a la función `GARLIC_getstring()` es síncrona, es decir, el proceso que invoca dicha función no continúa su ejecución hasta que el dispositivo haya completado su petición.

La nueva función `GARLIC_getXYbuttons()` tiene el propósito de permitir a los programas realizar una consulta no bloqueante del estado de los botones **X** e **Y** de la NDS, es decir, el proceso que invoque dicha función continuará su ejecución inmediatamente, obteniendo un patrón de bits que indicará el estado de los dos botones:

- **Bit 0**: =1 indica que el botón **X** está pulsado, =0 indica que está soltado,
- **Bit 1**: =1 indica que el botón **Y** está pulsado, =0 indica que está soltado.

Para recibir el estado de los botones, habrá que crear otra RSI de sincronización entre los procesadores ARM9 y ARM7, esta vez con el mecanismo **IPC-SYNC**, que el ARM7 activará cada vez que el usuario pulse o suelte uno de los dos botones.

Además, solo podrá recibir pulsaciones el proceso que esté en foco, es decir, aquel cuyo número de zócalo coincida con el contenido de la variable global `_gi_za` (ver apartado 4.5). El resto de los procesos siempre recibirán un cero como estado de los dos botones, aunque haya un botón pulsado. Esto permitirá al usuario controlar a qué proceso quiere mandar las pulsaciones.

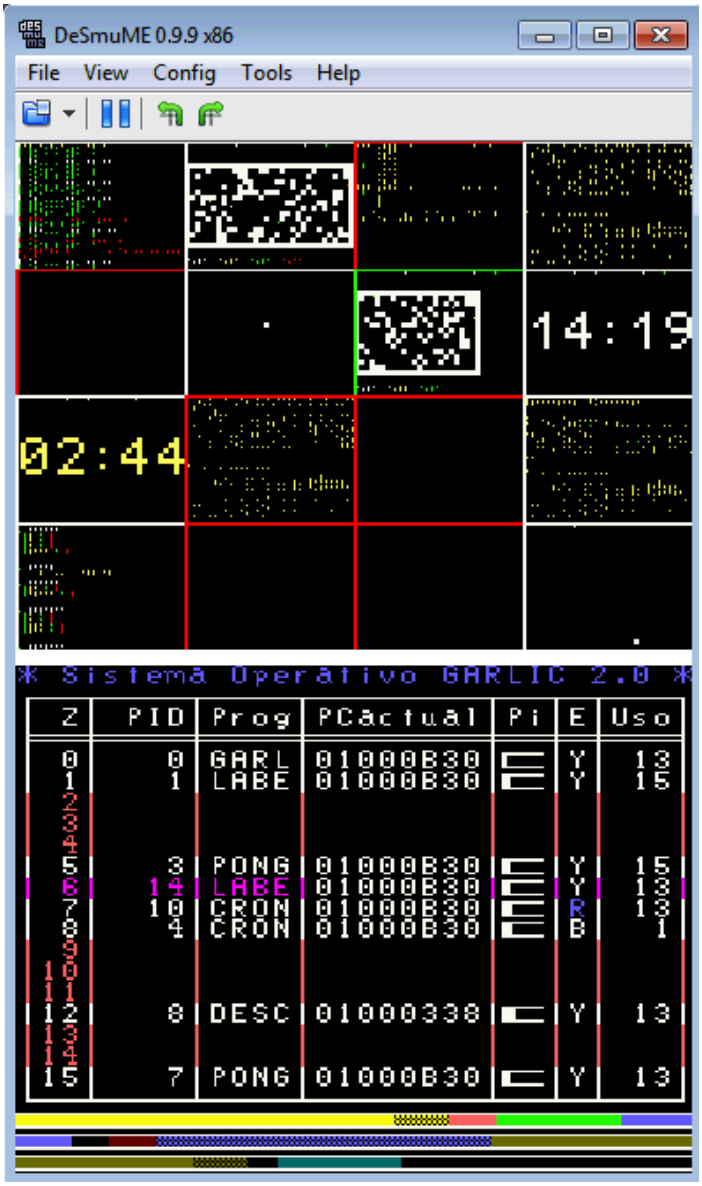
El programador `progT` deberá adaptar su programa para GarlicOS de modo que haga uso de esta nueva función de la API, puesto que los programas proporcionados de ejemplo no la invocan.

6 Tareas de integración del código (master)

Las tareas de fusión e integración de código de las distintas ramas de los programadores se repartirán entre los propios programadores.

Al final hay que conseguir un único programa, fusionado sobre la rama `master`, que integre todas las funcionalidades implementadas. Dicho programa se considerará como la versión definitiva del sistema operativo GarlicOS 2.0.

Se aconseja tomar la versión del proyecto de `progG` como referencia de la estructura final del sistema, puesto que contiene el control de la interfaz de usuario al completo, para ir añadiendo progresivamente el código de las otras ramas. El resultado esperado será similar al siguiente:



En la imagen anterior se puede observar que se han cargado hasta 7 procesos de usuario, más el proceso de control del sistema operativo. Hay algunos zócalos que están sin proceso. La ventana del zócalo 6 es la que está en foco, puesto que está resaltada en verde.

También se observa el porcentaje de uso de la CPU de todos los procesos activos, así como la ocupación de la pila y el estado de los procesos, donde el proceso que estaba en *RUN* en el momento de realizar la captura era el del zócalo 7.

En la zona baja de la pantalla inferior también se observa la ocupación de la memoria por parte de los procesos de usuario activos.

Obviamente, en el gráfico anterior falta representar la gestión de los procesos que esperan una entrada de información textual, lo cual depende de la realización de las tareas asignadas al programador `progT`.

Por último, hay que recordar que todos los programadores deben modificar su programa propuesto para GarlicOS 1.0, de manera que invoquen alguna nueva funcionalidad de la versión 2.0, como el retardo de ejecución o los colores de los caracteres enviados a la ventana. El programador de teclado tendrá que probar, además, la nueva función de la API para leer el estado de los botones **XY** de la NDS.

Atención: será necesario comprobar que la cantidad de memoria de las zonas TCM de la NDS usada por el código y los datos del sistema operativo no supera los límites del espacio disponible, teniendo en cuenta el resto de la información contenida también en dichas zonas (memoria de programas, pilas de procesos, pila del sistema, etc.).