Trabajo Práctico 1

Programación de Sistemas Operativos

1er Cuatrimestre 2011

1. Introducción

1.1. Objetivo

En este primer trabajo práctico la propuesta es comenzar con la confección de un Sistema Operativo desde cero. Se busca construir un sistema minimal que permita correr varias tareas concurrentemente y dar base para los siguientes dos trabajos prácticos.

1.2. Entrega

Fecha límite de entrega: **Jueves 5 de Mayo - 16hs**

La entrega del trabajo práctico debe realizarse en forma digital e impresa. Se debe enviar un correo electrónico a `pso-doc@dc.uba.ar` cuyo asunto (*subject*) sea "Entrega TP1 - Apellido1, Apellido2, Apellido3", conteniendo:

- Nombre, apellido y LU de los 3 (tres) integrantes del grupo
- Código fuente, en un archivo comprimido.
- Informe en formato digital.

Paralelamente, se debe entregar **impreso** el informe del trabajo práctico en una carpeta. No se debe imprimir el código fuente. Tampoco es necesario entregar el código en un medio de almacenamiento físico junto con el informe. El informe del trabajo debe contener la documentación de lo realizado, explicando *cómo* fue realizado.

1.3. Introducción

Para este trabajo se utilizará como entorno de pruebas el programa Bochs. El mismo permite simular una computadora IBM-PC compatible desde el inicio y realizar tareas de debugging.

Una computadora al iniciar comienza con la ejecución del POST y el BIOS, el cual se encarga de reconocer el primer dispositivo de booteo. En este trabajo utilizaremos un Floppy Disk como dispositivo de booteo. En el primer sector del dispositivo se almacena el boot-sector. El BIOS se encarga de copiar a memoria los 512 bytes del sector, a partir de la dirección `0x7c00`. Luego se comienza a ejecutar el código a partir esta dirección. El boot-sector proporcionado debe encontrar en el floppy el archivo `kernel.bin` y copiarlo a memoria. El kernel se copia a partir de la dirección `0x1200` y luego se ejecuta a partir de esta dirección. Es importante tener en cuenta que el código del boot-sector se encarga exclusivamente de copiar el kernel y dar el control al mismo, es decir, no cambia el modo del procesador.

Este es el punto de partida del trabajo práctico.

1.4. Alcance

El alcance de esta primer entrega del trabajo llegará hasta cubrir los siguientes temas o unidades del kernel:

- Manejo de pantalla e información de debug
- Manejador de memoria básico
- Loader y Scheduler
- Tareas en anillo 3
- Semáforos en el kernel

1.5. Organización general

El sistema se deberá programar mayormente en lenguaje C, con las partes necesarias en lenguaje ensamblador. El mismo se divide en difrentes módulos, con responsabilidades definidas, compilados todos juntos dentro del kernel.

Cada módulo tiene un nombre que lo identifica, y una lista de funciones y constantes que exporta al resto del kernel. Suponiendo que tenemos un módulo de nombre *modu* tendremos entonces los archivos `modu.c` y `modu.h` con la implementación de la funciones requeridas. Cada módulo deberá contar con una función de inicialización, llamada `modu_init`, en la cual se inicializan las estructuras, dispositivos, tablas, registros o mecanismos en general que sean necesarios para ofrecer la funcionalidad que exporta el módulo en cuestión. Por convención, todas las funciones exportadas por un módulo comenzarán con el nombre del módulo, como `modu_do_something`.

Se proveen una serie de archivos con el prototipo de algunas de las funciones a implementar y algunas estructuras de datos útiles. Para estos casos, no está permitido modificar este prototipo, pero se pueden agregar funciones auxiliares nuevas. Estos archivos se encuentran divididos de forma que los prototipos (archivos `.h`) están en el directorio `include` mientras que los archivos con la implementación están en el directorio `kernel`.

1.6. Organización de la memoria

Dado que las tareas deben ejecutar en su propio espacio de anillo 3 de manera aislada, es necesario proteger tanto la memoria del kernel como la memoria de cada proceso del acceso de otros procesos.

Para ello, cada tarea cuenta con su propio espacio de memoria virtual, desde donde no es posible acceder a las estructuras del kernel en anillo 3. Definimos 4 regiones de memoria en este espacio:

- Espacio de kernel (4MB a partir de `0x00000000`)
- Espacio de tarea usuario (código y datos, a partir de `0x00400000`)
- Espacio de bibliotecas de usuario (4MB a partir de `0xFFC00000`)

El espacio de kernel, los primeros 4MB, es memoria de supervisor con permiso de lectura y escritura, por lo que sólo puede ser accedida desde los anillos 0, 1 y 2. Estos primeros 4MB están mapeados con *identity mapping* a los primeros 4MB de la memoria física, a excepción de la siguiente página:

- la primer página (4KB desde la dirección `0x00000000`) la cual no está presente, con el fin de atrapar a través de un fallo de página los posibles accesos inválidos a la dirección NULL, incluso dentro del kernel.

Para finalizar, cada tarea tiene reservado su propio espacio de memoria que se reserva por el módulo `loader` al cargar la tarea (vea Sección 3.3), destinado a código y datos. La única excepción es el proceso de identificador 0 o *idle task* que corre en anillo 0 y en consecuencia puede acceder a su código directamente en el espacio del kernel.

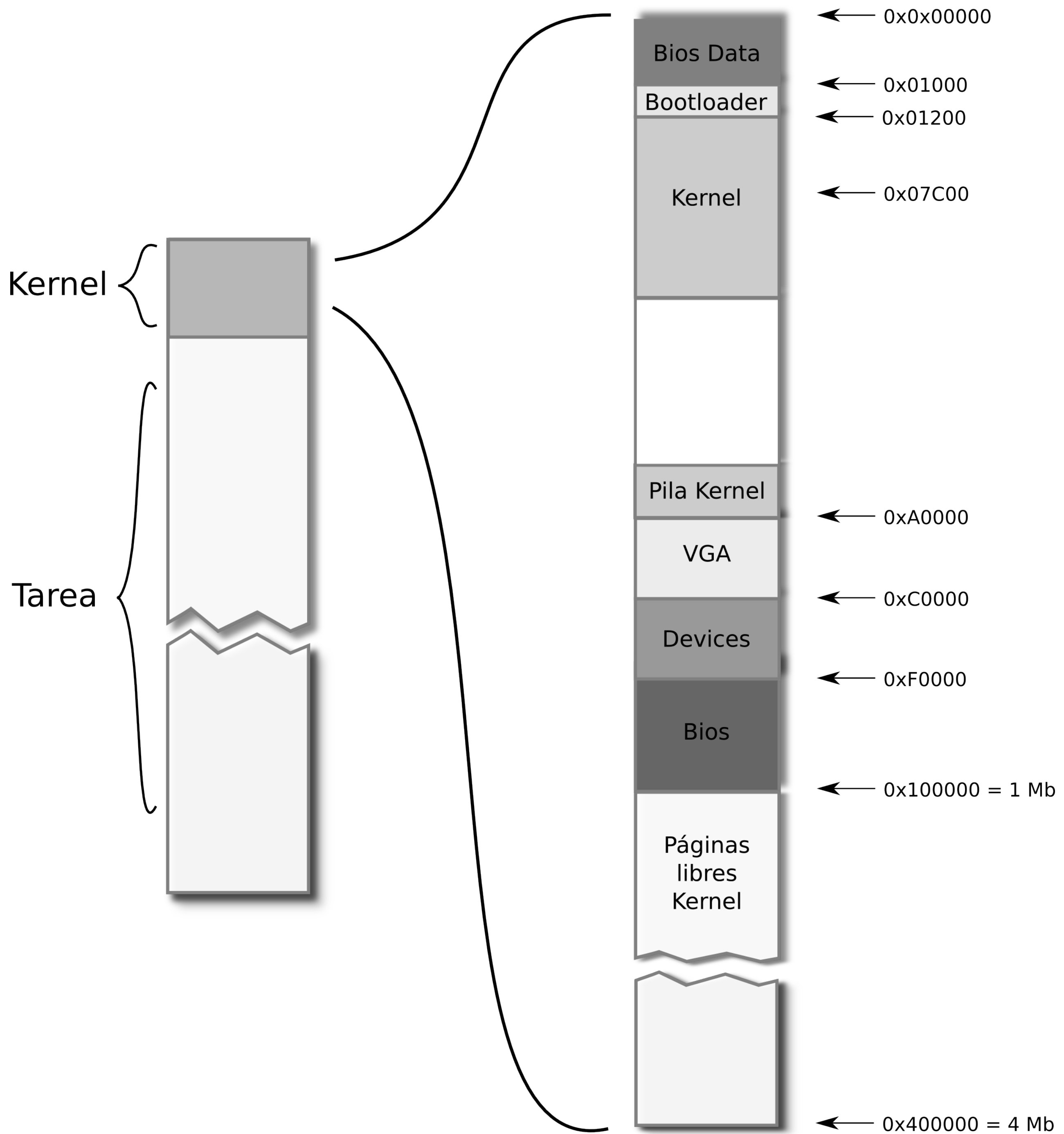

Figura 1: Mapa de memoria

2. Parte 1: Kernel básico

En una primer etapa, es necesario llegar a un kernel que si bien no tenga funcionalidad en sí mismo, permita las herramientas básicas para depurar y controlar el comportamiento del kernel.

2.1. `kernel` - Inicialización

Este módulo es responsable de la incialización del kernel, llamando a las funciones de inicialización de los demás módulos en un orden adecuado. Exporta una única función `kernel_init` que realiza esta inicialización y nunca retorna.

Ejercicio 1: Implementar la función `kernel_init`.

2.2. `gdt` - Global Descriptor Table

Inicialmente, incluso antes de que se pueda llamar a la función `gdt_init`, el kernel debe contar con una GDT. Por tal motivo, la inicialización debe ser estática, como constantes globales.

Ejercicio 2: Completar la tabla de descriptores global `gdt` en el archivo `gdt.c` para que tenga, en este orden, un descriptor nulo, código de anillo 0, datos de anillo 0, código de anillo 3, datos de anillo 3 y un descriptor de TSS que represente la tarea actual.

2.3. `vga` - Pantalla

La pantalla de la computadora en modo texto no se puede acceder utilizando las funciones del BIOS en modo protegido, porque estas ejecutan sólo en modo real. Este módulo permite acceder a la pantalla en modo texto de forma elemental. La pantalla inicialmente se encuentra en un modo de 80 columnas por 25 filas, a partir de la dirección `0xB8000`.

Como mínimo, se desea contar con una función que permita escribir un texto arbitrario (un string de C) en la pantalla, en una posición dada y con un color determinado.

Ejercicio 3: Implementar la función `vga_write` que permita escribir un texto en pantalla.

Ejercicio 4: *(Opcional)* Implementar la función `vga_printf` que permita escribir un texto formateado en la pantalla, reemplazando las ocurrencias de `%d` y `%x` en el string de formato por la representación en decimal y en hexadecimal de 8 dígitos, respectivamente.

2.4. `idt` - Administrador de interrupciones

El procesador, al detectar un error en la ejecución, se manifiesta a travez de excepciones. Los eventos externos al procesador indican a este su necesaria intervención a través de interrupciones. Los procesos o tareas solicitan servicios del kernel a través de *syscalls*. Todos estos eventos se numeran y tabulan en la IDT. Un correcto manejo de las excepciones, interrupciones y syscalls puede ser la diferencia entre un kernel sólido y uno inestable e inseguro.

Este módulo permite administrar las excepciones, interrupciones y *syscalls*. Debe exportar dos funciones, una de inicialización y otra que permite registrar una rutina de atención de interrupcion en un índice determinado en la IDT.

Ejercicio 5: Implementar la función `idt_register`, que dado un número de interrupción, una rutina de atención de interrupción y los atributos, registre esa interrupción en la IDT.

Ejercicio 6: Implementar la función `idt_init` que inicialice una IDT con todas sus entradas nulas. Además, se debe remapear la PIC para que para que las interrupciones de hardware se generen a partir del número 32 (`0x20`).

2.5. `debug` - Debug

Practicamente todos los seres vivientes que han interactuado alguna vez con una PC conocen la mundialmente famosa “pantalla azul” de Windows. No es tan conocido el “kernel panic” de linux, pero existe. Tanto la pantalla azul como el kernel panic son la respuesta del kernel ante un error o excepción que no puede ser resuelta.

```
Kernel Panic!

Stack:                                                         EAX 00000000
007FFF08:00000002 00000002 00000002 00000002                   EBX 00000000
007FFF18:00000000 00000000 00000000 00000000                   ECX 00000001
007FFF28:00000000 00000000 00000000 00000000                   EDX 00000077
007FFF38:00000077 00000077 00000077 00000077                   ESI 00000077
007FFF48:00000077 00000077 00000077 00000077                   EDI 00000011
007FFF58:00000000 00000000 00000000 00000000                   EBP 007FFF40
007FFF68:007FDFB0 007FDFB0 007FDFB0 007FDFB0                   ESP 0010:007FFF18
007FFF78:007FFFEC 007FFFEC 007FFFEC 007FFFEC                   EIP 0008:000072D4
007FFF88:007FDFB0 007FDFB0 007FDFB0 007FDFB0                   EFL 00000046
007FFF98:007FFFB8 007FFFB8 007FFFB8 007FFFB8
007FFFA8:007FDFB0 007FDFB0 007FDFB0 007FDFB0                   TR  00000050
007FFFB8:00000033 00000033 00000033 00000033 3   3   3   3     CR2 1234ABCD
007FFFC8:00000010 00000010 00000010 00000010                   CR3 00046000
007FFFD8:007FFFEC 007FFFEC 007FFFEC 007FFFEC                   PID 00000003

Backtrace: Current: 000072D4
At 000018FB: CALL 000072B0 ( 00000077, 00000000, 00000000, 00000003, ... )
At 00001CA3: CALL 00001840 ( 00000033, 00000033, 00000033, 00000033, ... )
At 00400023: CALL 00400180 ( 00000000, 00000000, 00000000, 00000000, ... )
return to: 00000000 <Invalid address>

EXP 0x0E err: 0x00000002 Page fault                                       2
```

Figura 2: Ejemplo de falla

Como se indica en la sección anterior, hay eventos externos que generar interrupciones en el procesador. Por este motivo, resulta imposible controlar completamente la ejecución. Esto hace que las tareas de *debugging* no resulten para nada sencillas. Para facilitar esta tarea, el kernel panic, cuando ocurra, debe mostrar por pantalla la mayor cantidad de información posible.

> **Ejercicio 7:** Implementar la función `debug_kernelpanic` que muestre por pantalla los siguientes datos:
>
> - Los registros de propósito general.
> - Los registros de control y el *task register*
> - El estado de la pila y sus correspondientes datos
> - Un backtrace.
>
> La información de los registros de propósito general se recibe por parámetro, dado que se desea mostrar los valores que tenían al momento del error.

Ejercicio 8: Implementar la función `debug_init` que registre, para cada excepción e interrupción de la PIC, una rutina de atención de interrupción que genere un kernel panic y muestre el número de excepción y el código de error de manera similar al a Figura 2. Neceistará de alguna forma proveer una rutina de atención de interrupción que llame a `debug_kernelpanic` con los parámetros adecuados.

Adicionalmente, se provee una macro `kassert(EXP)` que en caso de que el parámetro `EXP` evalúe a falso, muestra un mensaje en la pantalla, deshabilita las interrupciones y se cuelga indefinidamente.

3. Parte 2: Kernel funcional

3.1. `mm` - Manejador de memoria

El manejador de memoria le permite a los demás componentes del kernel compartir el uso de la memoria para los distintos propósitos de manera consistente.

En primer lugar, la función `mm_init` se encarga de iniciar las estructuras de datos necesarias por el kernel y detectar la memoria total disponible en el sistema. Según el esquema de memoria presentado, los primeros 4MB de memoria física se utilizan para páginas de kernel, mientras que las demás direcciones físicas se pueden destinar a páginas que utilizará el usuario.

Las funciones `mm_mem_alloc`, `mm_mem_kalloc` y `mm_mem_free` permitirán obtener páginas de memoría física para el usuario, el kernel y liberarlas.

Ejercicio 9: Implementar las funciones `mm_init`, `mm_mem_alloc`, `mm_mem_kalloc` y `mm_mem_free`. La funciones que reservan páginas nuevas deberán devolver 0 en caso de que no haya memoria disponible.

Ejercicio 10: Implementar la función `mm_dir_new` que genere un directorio de páginas nuevo con el mapa de memoria correspondiente explicado anteriormente.

Ejercicio 11: Implementar la función `mm_dir_free` que libere recursivamente toda la tabla de directorios de páginas apuntada por el `cr3` indicado

Ejercicio 12: Modificar e implementar lo que sea necesario para que con la syscall `palloc` la tarea pueda obtener una dirección virtual de una página de memoria libre de 4KB. Esta función devuelve 0 en caso de error.

3.2. `sched` - Scheduler

El Scheduler de un Sistema Operativo es el administrador del recurso más importante: el CPU. El *scheduler* en sí es sólo un algoritmo de asignación. Lo que se propone aquí es implementar un algoritmo de scheduling Round-Robin, con desalojo.

Ejercicio 13: Programar la función `sched_init` que inicialice las estructuras del scheduler. Esto es, toda la información que necesite el algoritmo de scheduling para funcionar correctamente.

El algoritmo de scheduling se rije por los eventos que van ocurriendo y reacciona en consecuencia. Cuando una nueva tarea es cargada, se le notifica al scheduler llamando a la función `sched_load` para que la tenga en cosideración. Cuando una tarea que estaba bloqueada se desbloquea, se le notifica al scheduler llamando a la función `sched_unblock`.

Por otro lado, la tarea que actualmente tiene el CPU puede generar eventos como bloquearse o terminar, que influyen en el algoritmo de scheduling. El último tipo de evento que puede generar una tarea es no generar ningún otro evento y consumir todo un ciclo de reloj del CPU, lo cual se traduce como la llegada de un *tick* de reloj por la IRQ0. Cuando ocurre alguno de estos eventos, se llama a las funciones `sched_block`, `sched_exit` y `sched_tick` dependiendo del origen del evento. Estas funciones devuelven el número de proceso (*pid*) al que se le debe dar el procesador a continuación. Recuerde que la tarea de *pid* 0 es la tarea que inicializa el kernel que luego se convierte en la *idle task*.

Ejercicio 14: Implementar estas funciones para un scheduler que utilize el algoritmo de Round-Robin.

3.3. `loader` - Loader y administrador de tareas

El `loader` será el encargado de administrar las colas y cambiar las tareas. Para esto cuenta con cierta información para cada tarea que describe su contexto o *Process Control Block*.

Para esto deberá programar varias partes del sistema operativo en este mismo módulo: el loader propiamente dicho, que se encarga de crear nuevas tareas, la parte encargada de hacer el cambio de tareas y la encargada de administrar las colas de tareas bloqueadas.

Ejercicio 15: Implementar la función `loader_init` que inicialice el manejador de tareas. Tenga en cuenta que la constante `MAX_PID` limita la cantidad de procesos en simultáneo que se pueden utilizar en el kernel. Tenga en cuenta que deberá registar una rutina de atención de interrupción para la IRQ0 y que deberá ajustar el valor del PIT para elegir la frecuencia a la que se debe llamar dicha interrupción.

El formato de archivo ejecutable que vamos a utilizar es el formato PSO.

```
typedef struct str_pso_file {
    sint_8 signature[4];
    uint_32 mem_start;
    uint_32 mem_end_disk;
    uint_32 mem_end;
    func_main* _main;
    uint_8 data[0];
} pso_file;
```

El formato PSO consiste en un encabezado y una única sección que contiene el código y datos del programa. El encabezado está compuesto por los siguientes elementos:

1. 4 bytes con los valores hexadecimales `50`, `53`, `4f`, `00`, que representan la cadena "`PSO\0`".
2. 4 bytes con la dirección de memoria virtual donde comienza el programa (usualmente `0x00400000`).
3. 4 bytes con la dirección de memoria virtual donde terminan los datos o código inicializado, que es sólo la parte que se escribe en el archivo.
4. 4 bytes con la dirección de memoria virtual donde termina el área del programa.
5. 4 bytes con la dirección de memoria virtual del punto de entrada del programa.

A continuación de este encabezado de 20 bytes comienzan el código y datos del programa. Tanto el encabezado como los datos siguientes forman parte de la información que será cargada en la dirección virtual referida en 2. Es decir que el encabezado del archivo está disponible para la tarea

en su espacio de memoria. El archivo completo debe tener un tamaño igual a la diferencia entre las direcciones descriptas en 3 y 2, que componen la información inicializada del programa. A partir de la dirección descripta en 3 y hasta la dirección descripta en 4 son datos no inicializados del programa y se completan con ceros.

Ejercicio 16: Implementar la función `loader_load` que, a partir de un archivo en formato PSO almacenado en la memoria, agregue la tarea correspondientes en el scheduler

En esta etapa del trabajo, los archivos PSO se cargarán dentro del `kernel.bin` por el bootloader.

El cambio de contexto entre tareas se deberá realizar *por soft*. Es decir, guardando en el PCB (Process Control Block) la información necesaria para restaurar la tarea en un futuro.

Ejercicio 17: Implementar la función `loader_switchto` que se encarge de cambiar a la tarea indicada por parámetro. Si la tarea indicada es la actual no se hace nada.

Ejercicio 18: Implementar la función `loader_enqueue` que encola la tarea actual en la cola pasada por parámetro a espera del evento correspondiente y salta a la siguiente tarea.

Ejercicio 19: Implementar la función `loader_unqueue` que genera un evento en la cola pasada por parámetro, desbloqueando la primer tarea de la cola, si es que hay alguna. Tenga en cuenta que esta llamada también debe notificar al scheduler de su desbloqueo.

La implementación de colas se realiza a través de números enteros (*pids*) que indican "la primer tarea de la cola", o -1 para indicar una cola vacía. El siguiente y el anterior de cada tarea se mantiene una vez por tarea dado que una tarea sólo puede esperar por un solo evento.

Ejercicio 20: Implementar la función `loader_exit` que libere todos los recursos utilizados por la tarea y la remueva de las colas que correspondan. Tenga en cuenta que esta llamada también debe notificar al scheduler de su remoción.

Ejercicio 21: Modificar la rutina de antención de la interrupción del reloj para que decremente los ticks de la tarea actual, y si es necesario, cambie a la siguiente tarea

Ejercicio 22: Implementar una syscall `getpid` que devuelva el *pid* de la tarea actual.

Ejercicio 23: Implementar una syscall `exit` que desaloje la tarea actual.

Ejercicio 24: Modificar el handler de la excepción de fallo de página para que verifique si es un tarea de anillo 3 la que generó la excepción, y si lo es, desaloje la tarea del mismo modo que lo haría la syscall `exit`

3.4. `sem` - Semáforos de kernel

Este módulo implementa un servicio de semáforos internos para el kernel.

Ejercicio 25: Implementar la función `sem_init` que inicialice todas las estructuras necesarias para utilizar semáforos

Ejercicio 26: Implementar la función `sem_wait` y `sem_signal` que hagan wait y signal sobre el semáforo dado.

Ejercicio 27: Implmementar la función `sem_broadcast` que haga broadcast sobre el semáforo. Evalúe si necesita implementar más funcionalidad en otros módulos.

4. Notas

En el directorio `tasks` del código fuente entregado, está el archivo `task1.c`. Esta tarea se compila y se incluye en el kernel. Esto no es definitivo, pero lo vamos a utilizar hasta que podamos leer de archivos.

Para incluir una nueva tarea `tarea.c` se deben seguir los siguientes pasos:

1. Crear el archivo `tarea.c` en el directorio `tasks`.
2. Modificar el archivo `Makefile`, añadiendo en la definición de `TASKS` el archivo `tasks/tarea.pso`.
3. Modificar el archivo `tasks.asm` la linea `include_task tarea, ‘‘tasks/tarea.pso’’`.

El archivo PSO quedará en un símbolo creado por el último paso: `task_tarea_pso`.

Ejercicio 28: Implementar los programas PSO necesarios para probar la funcionalidad del trabajo. Agregar las llamadas necesarias en la inicialización del kernel para que la tareas efectivamente se carguen en el sistema.