Sistemas Operativos

Trabajo Práctico 2 - Kernel

Informe

Instituto Tecnológico de Buenos Aires

Grupo 16

Integrantes:

- Juan Ignacio Cantarella (64509) - jcantarella@itba.edu.ar
- Ian Cruz Oniszczuk (64984) - ioniszczuk@itba.edu.ar
- Federico José Magri (64508) - fmagri@itba.edu.ar

Fecha de Entrega: 9 de junio de 2025

Introducción

En el presente informe se detalla el desarrollo del trabajo práctico. El mismo consiste en continuar el desarrollo de un kernel monolítico de 64 bits, comenzado en la materia Arquitectura de Computadoras. El mismo cuenta con manejo de interrupciones, system calls, driver de teclado, driver de video y binarios de kernel y user space separados.

Para conseguir un correcto funcionamiento y desarrollo del kernel, hemos desarrollado a su vez dos memory manager que pueden ser intercambiados en compilación, un sistema de procesos con multitasking preemptivo con una cantidad máxima de procesos, el algoritmo de scheduling usando round-robin con prioridades, una librería de semáforos como mecanismo de sincronización y una librería de pipes para IPC.

Instalación y ejecución

Prerrequisitos

Para poder hacer uso de nuestra consola de comandos, deberá tener instalado el emulador Qemu x86 (vea más adelante los pasos necesarios para su instalación en la sección Instalación de Qemu x86)

De no contar con el archivo qcow2 , deberá tener instalado Docker (vea más adelante los pasos necesarios para su instalación en la sección Instalación de Docker). El archivo qcow2 es la imagen de disco duro virtual usada por QEMU, y que será de donde se levante el SO a memoria cuando QEMU se inicie. Dicho archivo es generado dentro de la carpeta Image

Por último, algunos dispositivos o computadoras requieren tener instalado y ejecutándose en segundo plano XLaunch para la visualización de la consola de comando.

Instalación de qemu x86

Ejecute el comando ```sudo apt install qemu-system-x86 qemu-utils```

Ejecutar

Ejecutar ./compile.sh
Ejecutar ./run.sh

Arquitectura general del sistema

Physical Memory Manager

El sistema operativo desarrollado implementa dos mecanismos alternativos de administración de memoria física: uno utiliza el algoritmo Free List y el otro un Buddy System. Ambas implementaciones comparten una interfaz común para garantizar transparencia desde el punto de vista del kernel y de los procesos usuario. La selección entre uno u otro se realiza en tiempo de compilación mediante la variable de Makefile MEM_IMPL. Si se ejecuta make MEM_IMPL=buddy, se compilará el Buddy memory manager. Si no se especifica, se asume por default que se utilizará el de Free List.
El memory manager que utiliza free list se basa en una lista doblemente enlazada de bloques de memoria. Cada bloque incluye un header con la siguiente información: tamaño del bloque, estado (libre u ocupado) y punteros al bloque anterior y siguiente. Los bloques se alinean a 8 bytes mediante la macro ALIGN y se fusionan automáticamente al liberar memoria si los bloques contiguos también están libres (coalesce).
La segunda implementación implementa el algoritmo Buddy. Se administra mediante un árbol binario implícito, en el que cada nodo puede estar en estado FREE, SPLIT o FULL. Los bloques de memoria son fusionados con su hermano (buddy) cuando ambos están libres.

Procesos, Context Switch y Scheduling

El sistema operativo desarrollado implementa dos mecanismos alternativos de administración de memoria física: uno utiliza el algoritmo Free List y el otro un Buddy System. Ambas implementaciones comparten una interfaz común para garantizar transparencia desde el punto de vista del kernel y de los procesos usuario. La selección entre uno u otro se realiza en tiempo de compilación mediante la variable de Makefile MEM_IMPL. Si se ejecuta make MEM_IMPL=buddy, se compilará el Buddy memory manager. Si no se especifica, se asume por default que se utilizará el de Free List.

El memory manager que utiliza free list se basa en una lista doblemente enlazada de bloques de memoria. Cada bloque incluye un header con la siguiente información: tamaño del bloque, estado (libre u ocupado) y punteros al bloque anterior y siguiente. Los bloques se alinean a 8 bytes mediante la macro ALIGN y se fusionan automáticamente al liberar memoria si los bloques contiguos también están libres (coalesce).

La segunda implementación implementa el algoritmo Buddy. Se administra mediante un árbol binario implícito, en el que cada nodo puede estar en estado FREE, SPLIT o FULL. Los bloques de memoria son fusionados con su hermano (buddy) cuando ambos están libres.

Sincronización

Se encuentra implementado un sistema de sincronización por semáforos identificados por nombre. Para garantizar la atomicidad se utilizó la operación de ASM xchg que realiza un intercambio de valores entre registros o un registro y una dirección de memoria. El método utilizado para aprovechar lo anterior recibe el nombre de “spinlock”, donde la función spinlockAcquire hace que solo el proceso actual pueda modificar el valor del semáforo, hasta que este lo libere ejecutando la función spinlock Release.

En Kernel, la implementación consiste de un manager llamado “semManager”, que contiene la tabla “sems”, con la información individual de cada semáforo.

Cada entrada de la tabla representa al semáforo que tiene la id de la correspondiente posición, y es un struct sem_t que tiene los siguientes campos:

- value: Entero que representa el valor del semáforo.
- used: Flag que indica si la entrada de la tabla está en uso (1) ó libre (0).
- locked: Flag que indica si el semáforo está bloqueado por el sistema de spinlock.
- waiting: Cola de pids implementada en queueADT, que contiene los procesos bloqueados por el semáforo correspondiente.
- openedBy: Arreglo que contiene un flag por cada proceso, si el flag de la i-ésima posición está en 1, significa que el proceso con pid i está utilizando el semáforo, si está en 0, no.
- name: Nombre identificatorio del semáforo.

Para comunicar con userland, se implementaron las syscalls semOpen, semClose, semWait, semPost, semValue y semDestroy, a continuación una breve descripción de cada una de ellas:

- semOpen: Recibe un nombre y un valor, si ya existe un semáforo con dicho nombre, solamente lo abre para el proceso que invocó la llamada, si no existe lo crea e inicializa su valor según el parámetro envíado. Devuelve el id del semáforo creado, siendo este el valor a utilizar para referenciar el semáforo en userland.
- semClose: Recibe un nombre, si existe un semáforo identificado por ese nombre, lo cierra para el proceso que invocó la llamada, si no existe no hace nada.
- semWait: Recibe un id de semáforo, si su valor es mayor a 0, lo decrementa, si su valor es exactamente 0, bloquea el proceso que invocó la llamada, si no existe no hace nada.
- semPost: Recibe un id de semáforo, incrementa su valor, si inicialmente el valor es 0, desbloquea todos los procesos que hayan sido bloqueados por un wait de dicho semáforo.
- semValue: Recibe un id de semáforo, si este existe devuelve su valor, en caso contrario devuelve -1.

- semDestroy: Recibe un id de semáforo, si este existe lo destruye, desbloqueando todos los procesos que se encontraban en espera y marcando en la entrada de la tabla sems que el correspondiente semáforo ya no está en uso, liberando ese espacio para futuros nuevos semáforos. Si no existe, no hace nada.

Inter Process Communication

La comunicación entre procesos se implementó mediante pipes unidireccionales basados en file descriptors. Cada pipe utiliza un buffer circular y semáforos para garantizar operaciones bloqueantes, evitando busy waiting y asegurando sincronización entre procesos lectores y escritores.

Los procesos pueden acceder a los pipes a través de descriptores de archivo que abstraen su modo de uso (lectura o escritura). El sistema mantiene un recuento de referencias por modo y detecta automáticamente condiciones de fin de flujo (EOF). Además, se permite redirigir los descriptores estándar (stdin, stdout, stderr), lo cual habilita la conexión entre procesos en la shell sin requerir cambios en su lógica interna.

Se manejan correctamente los casos especiales de salida estándar, que nunca deben bloquear, y se proporciona una interfaz uniforme para leer o escribir tanto desde terminal como desde pipes, cumpliendo con los requisitos de transparencia del sistema.

Procesos, Context Switch y Scheduling

Unit Testing

Además de los testeos que se realizan desde user space, hemos agregado en la carpeta Tests por fuera del kernel unit tests de ambos memory manager y del queue implementado por el grupo. Los tests siguen un formato similar al visto en la clase de unit testing usando CuTest. Por razones de estructura del código, hemos tenido que “copypastear” todas las implementaciones a ser testeadas otra vez en su archivo de testeo correspondiente.

A su vez, hemos implementado un workflow de github para automatizar el testeo. Cada vez que se hace un push, se corren los tests. En la pestaña “actions” del repositorio pueden verse los resultados de los tests ante cada push.

Para correr los tests, se debe posicionar sobre la carpeta Tests y correr make all. Luego, fuera de la carpeta Tests, en la carpeta raíz del proyecto, se habrá generado el archivo AllTests.out que al ser ejecutado mostrará por consola el resultados de los tests.

Si bien el testing/desarrollo de unit tests no era requisito del trabajo, nos pareció interesante y útil hacerlo para tener más seguridad de que nuestras

implementaciones más fundamentales (los memory manager y la queue) funcionan correctamente.

Decisiones tomadas durante el desarrollo

A continuación se listan las decisiones de diseño separadas por secciones:

Semáforos:

En esta sección, la gran disputa estuvo en la identificación de los semáforos. En un comienzo utilizamos el struct sem_t (entrada de la tabla de semáforos) como identificador, pero esto resultó en mucha información innecesaria y dificultades para su creación. Decidimos cambiarlo directamente por su correspondiente índice en dicha tabla, a lo que llamamos “semId” o id del semáforo. Esta decisión trajo algunos problemas, ya que la función semOpen crea un semaforo en el primer lugar disponible, pero antes verifica si existe un semáforo con la id envíada, abriendolo, siendo este un problema, ya que si se quería crear un semáforo nuevo, el parámetro de entrada salida id, debía estar inicializado con un valor que no estuviera en uso, pero no hay forma de saber eso ya que el kernel se encarga de asignar las ids. Aquí surgió la decisión final, identificarlos por nombre, de esta manera, el usuario es quien le asigna un identificador y el kernel se encarga de asignarle un id.

Problemas encontrados

Durante el desarrollo, debimos afrontar distintas complicaciones que impedían el correcto funcionamiento del sistema. A continuación se presenta una breve descripción de cada una.

1. Desbloqueo de procesos por semPost: Durante la fase de pruebas, notamos que la función post incrementaba el valor de los semáforos correctamente y cambiaba el estado del proceso a ready, pero su ejecución nunca sucedía. Tras debuggear a fondo, encontramos que el problema estaba en semWait, ya que cuando el valor del semáforo era cero, se entraba a un bucle while de la forma:

```
…
sem_t sem = sems[semId];
…
while(sem.value == 0){
    …
    blockProcess(currentProcess);
    …
```

```
}
…
```

Como se puede observar, sem es una variable local, por lo que al hacer semPost, se incrementaba correctamente el valor, desbloqueando el proceso, que continúa su ejecución dentro del while, pero en la variable sem no se ve reflejada la modificación y sem.value vale 0 para siempre.

La corrección consistió en eliminar la variable local y trabajar directo con la tabla de semáforos (sems[semId].value);

2. Semáforos: Al integrar semáforos en la implementación de pipes, detectamos mediante GDB que había procesos bloqueados de forma indefinida. El problema se debía a un uso incorrecto de `spinlockAcquire` y `spinlockRelease`, lo que provocaba condiciones de carrera al acceder a estructuras compartidas como los buffers del pipe.

   El error estaba en el semDestroy, no se usaban bien las funciones spinlockAcquire y spinlockRelease pues los semaforos no se liberaban bien (lo vimos gracias a gdb).

3. Pipes con semáforos: A la hora de integrar semáforos a nuestra implementación de pipes dejó de funcionar correctamente todo el sistema. Revisando la inicialización de los file descriptors, encontramos que no se creaban nunca los semáforos de lectura y escritura, por lo que la solución consistió en agregar la creación de los mismos verificando que siempre tengan nombres distintos, utilizando el id del fd. (itoa(fd)).

Análisis estático-dinámico con PVS-Studio

Para realizar el análisis estático-dinámico utilizamos la herramienta PVS-Studio, a continuación se presentan los warnings que quedaron:

```
pvs-studio.com/en/docs/warnings/    1    err    Help: The documentation for all analyzer warnings is available here: https://pvs-studio.com/en/docs/warnings/.
/root/Bootloader/BMFS/bmfs.c    554    err    V595 The 'disk' pointer was utilized before it was verified against nullptr. Check lines: 554, 577.
/root/Bootloader/BMFS/bmfs.c    640    warn    V1032 The pointer 'dir_copy' is cast to a more strictly aligned pointer type.
/root/Bootloader/BMFS/bmfs.c    696    warn    V1032 The pointer 'Directory' is cast to a more strictly aligned pointer type.
/root/Bootloader/BMFS/bmfs.c    454    note    V575 The potential null pointer is passed into 'memset' function. Inspect the first argument.
/root/Bootloader/BMFS/bmfs.c    484    note    V575 The potential null pointer is passed into 'rewind' function. Inspect the first argument.
/root/Bootloader/BMFS/bmfs.c    240    note    V576 Incorrect format. Consider checking the third actual argument of the 'printf' function. The SIGNED integer type argument is expected.
/root/Bootloader/BMFS/bmfs.c    256    note    V576 Incorrect format. Consider checking the fourth actual argument of the 'printf' function. The SIGNED integer type argument is expected.
/root/Bootloader/BMFS/bmfs.c    256    note    V576 Incorrect format. Consider checking the third actual argument of the 'printf' function. The SIGNED integer type argument is expected.
```

```
/root/Bootloader/BMFS/bmfs.c    410    note    V576 Incorrect format. Consider checking the second actual argument of the 'printf' function. Under certain conditions the pointer can be null.
/root/Bootloader/BMFS/bmfs.c    691    note    V576 Incorrect format. Consider checking the second actual argument of the 'printf' function. The SIGNED integer type argument is expected.
/root/Bootloader/BMFS/bmfs.c    371    note    V584 The 'diskSize' value is present on both sides of the '>' operator. The expression is incorrect or it can be simplified.
/root/Bootloader/BMFS/bmfs.c    503    note    V1004 The 'mbr' pointer was used unsafely after it was verified against nullptr. Check lines: 394, 503.
/root/Kernel/kernel.c    35    note    V566 The integer constant is converted to pointer. Possibly an error or a bad coding style: (void *) 0x400000
/root/Kernel/kernel.c    36    note    V566 The integer constant is converted to pointer. Possibly an error or a bad coding style: (void *) 0x500000
/root/Kernel/kernel.c    37    note    V566 The integer constant is converted to pointer. Possibly an error or a bad coding style: (void *) 0xF00000
/root/Userland/SampleCodeModule/modes.c    368    note    V776 Potentially infinite loop.
```

Se observan muchos warnings de la sección BMFS, que tiene que ver con la inicialización del entorno donde se encuentra el sistema, es por eso que se encuentra fuera de nuestro alcance. Lo mismo con kernel.c.

En cuanto al archivo modes.c, se trata de un potencial bucle infinito generado por la implementación del comando “wc”, esto se debe a que hay una estructura de forma while(c = getchar() != EOF), siendo el potencial bucle un comportamiento esperado.

Bibliografía consultada y fuentes de referencia

Memory manager:
https://www.youtube.com/watch?v=DRAHRJEAEso&t=662s
https://github.com/jemalloc/jemalloc
https://github.com/rhempel/umm_malloc

Scheduler:
Lottery scheduling - Wikipedia
US5247677A - Stochastic priority-based task scheduler - Google Patents
(Finalmente implementamos el round robin que pedía la consigna)

Unit testing y workflow de github:
https://github.com/alejoaquili/c-unit-testing-example