Trabajo Práctico N°2

Grupo 16

Sistemas Operativos - 72.11

Arancibia Carabajal, Nicolás Guillermo	64481
Birsa, Juan Pablo	64500
Borda, Tomás	64517

Instituto Tecnológico de Buenos Aires

Fecha de entrega: 23/09/2024

Docentes: Ariel Godio

Alejo Ezequiel Aquili

Fernando Gleiser Flores

Guido Matias Mogni

1. Introducción

Durante el presente informe se detalla el desarrollo del trabajo práctico. El mismo consiste en la creación de un sistema operativo de tipo monolítico, en base al kernel previamente desarrollado en la materia *Arquitectura de Computadoras*, el cual incluye manejo de system calls, driver de teclado, driver de video, interrupciones básicas, y por último, una separación de los binarios del kernel y user space. Durante el trabajo práctico, se implementaron en el kernel funcionalidades básicas para que un sistema operativo pueda gestionar y administrar correctamente los recursos del sistema computacional.

2. Desarrollo del proyecto

2.1. Manejo de memoria

Para el manejo de la memoria dentro del trabajo práctico, se realizó un administrador de memoria, llamado *Memory Manager*, el cual se encarga de brindar al usuario la opción de reservar memoria de tipo variable, y luego poder liberarla.

Inicialmente se comenzó con un memory manager simple que retornaba bloques de tamaño fijo cuyos punteros eran almacenados en un arreglo dentro del kernel. Sin embargo, se optó por complejizar dicho administrador a fin de eliminar la fragmentación interna. Consecuentemente se realizó la implementación de un memory manager a partir de listas, donde cada bloque de memoria cuenta con un header con el tamaño del bloque, un puntero hacia el próximo, y otro puntero hacia el bloque anterior. Esto conlleva que al momento de contar con bloques contiguos, no tendremos una fragmentación externa, ya que al momento de liberar un bloque, si sus consiguientes se encuentran libres, estos mismos se integran en uno solo.

Por otro lado, al momento de asignar memoria al usuario, no contamos con bloques de tamaño fijo, sino que entregamos la memoria que el usuario nos solicita, y en el caso de contar con un bloque de memoria con un tamaño mayor al pedido, lo fragmentamos en 2 bloques de memoria contiguos, lo cual conlleva a no tener una fragmentación interna, dado que el usuario siempre recibe un bloque de memoria del tamaño que él mismo solicitó.

2.2 Scheduler

Para correr los procesos y manejar los cambios de contexto entre los mismos se implementó un Round-Robin con prioridades como scheduler. Para ello se utiliza una única lista cuyos nodos contienen un puntero a la información del proceso a correr (incluyendo el último valor del RSP). Cada vez que un proceso termina su ejecución, se crea nuevamente un nodo con dicho proceso en la lista y se actualiza el valor del RSP. Los nodos de procesos bloqueados o muertos son eliminados de la lista, liberando el espacio ocupado por el mismo.

Para el manejo de las prioridades, se manejaron 5 niveles, yendo del 1 como más baja hasta el 5 como la más alta. Se interpretaron las mismas como la cantidad de ejecuciones consecutivas que el scheduler le otorga a un proceso: cada vez que un nodo se crea para un

proceso, se almacena la cantidad de ejecuciones consecutivas en base a su prioridad en dicho momento. Consecuentemente, cuando el proceso esté corriendo y se produzca una interrupción del Timer Tick, si aún tiene ejecuciones restantes entonces devuelve el mismo contexto. Además se agregó un flag que permite evitar esta verificación en caso de que el proceso esté queriendo cambiar forzosamente el contexto por medio de la syscall *yield*.

2.3 Sincronización

Para el manejo de la sincronización entre procesos se crearon semáforos con un comportamiento casi análogo a su implementación POSIX. Para la adquisición de un semáforo se requiere el uso de la syscall *sysSemOpen* la cual necesita de un valor inicial y un string que será la referencia del semáforo y mediante el cual podremos interactuar con el mismo. Al momento de abrir un semáforo este busca si ya fue abierto previamente o debe crearse desde cero, para ambos casos guarda el pid del proceso que lo abrió para más adelante no dejar que procesos que no abrieron el semáforo puedan interactuar con el. Para la adquisición sobre el control del semáforo, más puntualmente, para asegurarse exclusión mutua al momento de modificar el valor de este, se utilizó un spinlock. Así, tanto para las syscalls de *sysSemWait, sysSemPost,sysSemClose* se asegura un uso exclusivo para el proceso que lo solicita, luego en particular para el *sysSemWait* si el valor del semáforo estaba en 0, significa que no quedan accesos a la zona critica que haya sido delimitada con el uso del semáforo, por consiguiente el proceso se encola en una lista de espera y se bloquea cediendo el tiempo de CPU. Finalmente cada proceso puede cerrar el semáforo teniendo en cuenta que no podrán volver a interactuar con él hasta abrirlo nuevamente y si el semáforo fue cerrado por todos los procesos que lo hayan abierto, este procederá a destruirse, asegurándose previamente, acceso exclusivo para que no interrumpan su borrado.

3. Instrucciones de compilación y ejecución

Para la compilación del proyecto se realiza mediante la utilización de una imagen de Docker. En caso de no contar con ella, correr el comando ***docker pull agodio/itba-so-multi-platform:3.0***. Una vez hecho esto, únicamente se requiere ejecutar el script de compilación mediante ***./compile.sh***.

Para ejecutar el proyecto se requiere de qemu. En caso de requerir instalarlo se puede hacer mediante el comando ***sudo apt install qemu-system-x86 qemu-utils***. En caso de estar utilizando WSL para la ejecución, requerirá de XLaunch, el cual es posible instalar desde *Sourceforge* (Ver Anexo 6.1). Una vez que se tengan dichas dependencias, es posible ejecutar el proyecto mediante el script ***./run.sh***. En caso de querer ejecutar sin inicializar el driver de sonido se puede hacer mediante ***./run.sh nosound***. Una vez hecho esto, debería ver una ventana con la consola de comandos. Una vez allí, el comando ***help*** nos permitirá ver todos los comandos disponibles, los cuales pueden ser ejecutados simplemente escribiéndolos.

Para compilar y ejecutar el test de memoria por fuera del kernel, parado sobre la carpeta del proyecto debe moverse al directorio ‘Tests’ mediante ***cd Tests*** y una vez dentro

correr el comando ***make*** el cual se encargará de compilar y generar un ejecutable llamado 'app', una vez que el ejecutable se haya generado podrá correrlo mediante ***./app*** iniciando así el test del memory manager por fuera del kernel.

4. Limitaciones

- **Tamaño de la memoria disponible para el test**: al ser un test de estrés que intenta reservar toda la memoria disponible o hasta que pida los bloques que se le indicaron, siendo que este desconoce cómo se administra la memoria, y en particular cuanta está destinada por bloque para los campos de información como lo son el tamaño, sí está libre, etc., desde kernel se debe reservar memoria suficiente tanto para la que será memoria libre como la necesaria para cada bloque que puede ser creado, dato que de momento se le proporciona al memory manager para indicarle la cantidad de bloques máximo que puede tener hasta la fecha. Dicho valor es 100 y se puede modificar en el kernel al momento de inicializar el memory manager.

- **Cantidad máxima de procesos:** La cantidad máxima de procesos soportada es 10. Dicho límite puede modificarse fácilmente mediante la constante ***MAX_PROCESSES*** en el archivo de *process.h*.

- **Wait:** La implementación de procesos no contempla el estado 'zombie' para los procesos. Si un proceso muere y no hay otro haciendo wait de él, su valor de retorno se pierde y no podrá ser consultado posteriormente.

- **Cantidad máxima de semáforos:** La cantidad máxima de semáforos soportada es 15. Dicho límite puede modificarse fácilmente mediante la constante ***MAX_SEMAPHORES*** en el archivo de semaphore.h.

5. Citas de fragmento de código reutilizado

- Para el driver de sonido se reutilizó código de la página de *OSDev*, disponible en el siguiente link: https://wiki.osdev.org/PC_Speaker.

6. Anexo

6.1 XLaunch

Es posible descargar el programa mencionado en el siguiente link: https://sourceforge.net/projects/vcxsrv/

7. Bibliografía consultada

- Apuntes de clase

- Modern Operating Systems de Andrew S. Tanenbaum y Herbert Bos