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: 11/11/2024

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 realizarón dos administradores de memoria, uno de tipo *Freelist* y otro de tipo *Buddy*, los cuales se encargan de brindar al usuario la opción de reservar memoria de tipo variable, y luego poder liberarla.

En cuanto al memory manager de tipo freelist, cada bloque de memoria cuenta con un header que posee 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.

En cuanto al memory manager de tipo Buddy, se optó por una implementación del mismo usando múltiples listas donde cada una de ellas guarda un tamaño fijo en base a una constante MIN_BLOCK_SIZE y al nivel en el que se encuentra, por ejemplo la lista del nivel 0 guarda bloques de tamaño MIN_BLOCK_SIZE, las de nivel 1 guarda los de tamaño 2 * MIN_BLOCK_SIZE, las de nivel 2 los de 2^2 * MIN_BLOCK_SIZE, y así consecutivamente. Los bloques asignados, consecuentemente, son de tamaño fijo. Sin embargo, si al momento de asignar un bloque de memoria puedo reducir su tamaño a la mitad, lo hago. Así también, en caso de liberar memoria, me fijo si su *'buddy'* del mismo tamaño está libre, y, en caso afirmativo, los uno.

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 abrirlo desde cualquier otro proceso. 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.

2.4 File descriptors

Para el manejo de file descriptors se realizó una estructura que contiene información acerca de si el mismo está abierto, qué permisos tiene (lectura, escritura o ambos) y otra estructura llamada *Stream* que contiene la información necesaria para la utilización de los file descriptors en lectura y escritura: principalmente allí se encuentra el buffer en el que se almacena la información, así como dos semáforos que se utilizan para las operaciones sobre el file descriptor de tal modo que las mismas sean bloqueantes (Es decir, un proceso se bloquea en la lectura si no hay información en el buffer, y lo mismo en la escritura si el buffer está lleno). Así también, se manejó el caso especial de STDOUT y STDERR, ya que dichos file descriptors siempre tienen lugar en pantalla para escribir, por lo que nunca deberían bloquear.

Cada proceso tiene en su estructura sus propios file descriptors, los cuales pueden ser modificados en la creación del mismo. Esto permite que se puedan pipear procesos en la shell.

2.5 Señales

El proyecto soporta el envío de dos señales: SIGINT con CTRL+C y EOF con CTRL+D. Es importante saber que la señal de SIGINT únicamente puede enviarse a procesos corriendo en foreground.

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***. Esto compilará el proyecto con el *freelist* como memory manager. En caso de querer utilizar el sistema buddy, debemos ejecutar el script con el comando 'buddy', es decir, ejecutaríamos ***./compile.sh buddy***.

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.

4. Instrucciones de ejecución de comandos

La shell posee diversos comandos que podran ser ejecutados por el usuario, el primero y principal es help, que lista todos los comandos disponibles junto con una breve descripción de su funcionamiento. A continuación presentamos una tabla con todos los comandos y cuales son built-in y cuales no.

Comandos	
Built-in	**No built-in**
help	testmm

clear	testproc
time	testprio
date	testsync
registers	testnosync
yield	testpipe
ps	cat
kill	wc
suspend	filter
resume	loop
nice	philo
memstatus	

Para ejecutar un comando el orden para la sintaxis es primero el comando, luego sus argumentos y finalmente como opcional '&' para indicar que corre en background, como default se espera que corra en foreground.

Para poder pipear dos comandos se usa '|', la sintaxis es la misma con la excepción de que no se puede decidir si correr un proceso en background y el otro no, o todo el conjunto corre en background o en foreground. Por lo tanto uno tipearia el primer comandos juntos con los argumentos correspondientes, el símbolo para pipear '|', y repite los mismos pasos para el siguiente comando donde al final puede opcionalmente usar '&'.

Los comandos built-in no pueden correrse en background dado que implicaría que la shell puede correr en background, sin embargo, hay una forma de hacer que corra en background un comando built-in y es haciendo que lo haga como proceso. Dado que la implementación hecha permite que si uno o dos comandos built-in son usados en un pipeline, se cree un proceso para correr dicho comando si al final del pipeline se mandará un '&' esto haría que en conjuntos ambos corren en background.

5. Limitaciones

- **Tamaño de la memoria disponible para el test**: Tanto el buddy como nuestro memory manager toman espacio en el bloque que se aloja para almacenar metadata

correspondiente al mismo, debido a que no es posible conocer previamente cuantos mallocs va a hacer un programa, al momento de correr el test si este quiere tomar toda la memoria o un valor muy cercano es posible que no pueda encontrar un bloque libre pues parte de esa memoria libre se usó en la metadata de cada bloque.

- **Cantidad máxima de procesos:** La cantidad máxima de procesos soportada es 64. 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 mismos. 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 2048. Dicho límite puede modificarse fácilmente mediante la constante ***MAX_SEMAPHORES*** en el archivo de semaphore.h.

- **File descriptors:** La cantidad máxima de file descriptors está limitada por una constante ***MAX_FDS*** que se encuentra en el archivo fileDescriptors.h. Asimismo, en dicho archivo se puede encontrar la constante que limita el tamaño máximo del buffer de cada file descriptor, es decir, ***BUFFER_SIZE***.

- **Cantidad de filósofos:** La cantidad mínima y máxima de filósofos se encuentra determinada por las constantes ***MIN_PHILOSOPHERS*** y ***MAX_PHILOSOPHERS*** respectivamente, las cuales se encuentran en philosophers.h.

- **Cantidad máxima de comandos en pipeline:** la cantidad máxima de comandos a ejecutarse en un pipeline son 2, esto siguiendo la sugerencia de la consigna y por ende existe más de un chequeo para evitar el caso de más de dos comandos en un pipeline

- **Cantidad máxima de argumentos:** Para el parseo de comandos de la shell se definió un constante para la cantidad máxima de argumentos permitida para un comando, esto puede modificarse en el archivo modes.c siendo el define de MAX_ARGS

- **Cantidad máxima caracteres para wc,cat y filter:** Para wc,cat y filter que usan un buffer interno para procesar su entrada, en programs.c se definió una constante MAX_BUFFER para determinar la cantidad máxima de caracteres soportada, esta puede modificarse si problema alguno, si la entrada de alguno de estos comandos es mayor a la de su buffer interno, permitirán seguir recibiendo caracteres hasta un EOF, sin embargo su salida reflejara hasta donde logró almacenar

- **SIGINT:** La implementación de ésta señal se encarga de matar a todos los hijos del proceso y a sí mismo. Esto no libera los recursos que el proceso podría haber estado usando, es decir, semáforos, bloques de memoria solicitados por medio de *malloc*, etc.

6. Warnings de PVS-Studio

- **Integer constant is converted to pointer (V566):** Esta nota se trata de un falso positivo dentro del contexto del tp, ya que conocemos de antemano dichas direcciones y es necesario inicializarlas para así utilizarlas.

- **Potentially infinite loop (V776):** Se obtiene una advertencia acerca de que la shell entra un loop infinito, algo que es razonable dado que es el comportamiento esperado dentro del tp, pues no debe haber forma de salir de la shell.

- **Dangerous to call the 'printf' (V618):** Este warning aparece porque considera que el puntero a char que se le pasa a printf podría tener algún formato dentro, como por ejemplo un %d o %s, y se sugiere que se realice algo como *printf(%s, string)*. Sin embargo, este warning aparece porque se compara con la versión de printf perteneciente a la librería estándar de C, no contempla nuestra implementación de printf. Además, el texto que se imprime está garantizado que siempre será ' . ' o ' E ', por lo que nunca ocurrirá el caso por el que se advierte acerca de ésta instrucción.

7. 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.

8. Anexo

8.1 XLaunch

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

9. Bibliografía consultada

- Apuntes de clase

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

- Page Frame Allocation - OSDev

- Memory Allocation - BrokenThorn