72.11 Sistemas Operativos

TP2 - INFORME

Grupo 22

Santiago Devesa (64223)

Tiziano Fuchinecco (64191)

Tomas Balboa (64237)

Profesores

Alejo Ezequiel Aquili

Ariel Godio

Fernando Gleiser Flores

Guido Matías Mogni

Índice

Introducción

El objetivo de este trabajo práctico es desarrollar un kernel básico monolítico de arquitectura de 64 bits. Este kernel debe incluir un sistema de manejo de interrupciones, implementar system calls y proporcionar drivers, como los de teclado y video. Además, se requiere la separación de binarios entre el espacio de kernel (kernel space) y el espacio de usuario (user space) para una mejor organización y control del entorno de ejecución.

El trabajo se divide en tres entregas parciales, en primer lugar la administración de la memoria, en donde se debe desarrollar un administrador de memoria física seleccionable en compilación (entre el Buddy System y otro mecanismo elegido).

En segundo lugar el manejo de procesos y su planificación, utilizando cambios de contexto y un algoritmo de scheduling (Round Robin con prioridades).

En la tercera entrega parcial se implementó el sistema de sincronización a través de semáforos con nombre. Finalmente, el trabajo consta de una última entrega para la cual se deben implementar pipes unidireccionales y una serie de aplicaciones de user space que permitan reflejar el cumplimiento de los requisitos mencionados previamente.

Compilación del trabajo práctico

1. Ejecutar el comando:

   docker pull agodio/itba-so-multi-platform:3.0
2. Clonar el repositorio:

   git clone git@github.com:smdevesa/TP2_SO.git
3. En el directorio del repositorio, ejecutar el comando:

   docker run -v ${PWD}:/root --security-opt seccomp:unconfined -ti agodio/itba-so-multi-platform:3.0
4. Dentro del contenedor, ejecutar los siguientes comandos:

   cd /root/Toolchain

   make all

   cd ..

   make all (compilación con administrador de memoria naive) o *make buddy* (compilación con buddy system)

5. Salir del contenedor con el comando:

   exit

Compilación del testeo de memoria en aplicación externa

1. Dirigirse a la carpeta *memory_tests* con el comando:

   cd Tests/memory_tests
2. Ejecutar los comandos:

   make all

Ejecución del trabajo práctico

Ejecutar el archivo *run.sh* con el comando: *'./run.sh'* en Linux o macOS.

De esta forma es posible ejecutar los testeos del manejo de memoria en el kernel del trabajo práctico.

Una vez ejecutado el comando ./run.sh, escribiendo *'help'*, se listan todos los comandos a ejecutar de la shell.

Test de memory manager

Se puede ejecutar el comando *'tmm [X] &'* para correr el test de memory manager en background siendo X la cantidad de direcciones de memoria a utilizar para el test. Los tests corren en foreground por defecto, por lo que este test debe ser corrido en background para no perder el procesador para siempre.

Test de context switching y prioridades

Para correr el test_processes se debe ejecutar *'ts [X] &'* siendo X la cantidad de procesos a crear. Por limitaciones del sistema la cantidad máxima de procesos por default es 25. Por otro lado, para ejecutar el test_priority hay que ejecutar *'tp'*. Los tests corren en foreground por defecto, por lo que test_processes debe ser corrido en background para no perder el procesador para siempre.

Test de sincronización y no sincronización

Para correr el test_sync se ejecuta *'tsy [n] [use_sem]'*, siendo n la cantidad de veces que se incrementará o decrementará la variable global *'global'* en el test de sincronización. Por otra parte, el valor de [use_sem] será 1 cuando se desee el uso de semáforos para sincronizar el manejo de la variable global, y 0 en caso contrario. Por defecto el test corre en foreground, aunque se tiene la opción de correrlo en background con el comando *'tsy [n] [use_sem] &'*.

Ejecución del testeo en aplicación externa

Test de memory manager

Si se desea ejecutar el testeo de memoria en una aplicación externa al trabajo práctico se debe ejecutar el comando:

./mmTest

Decisiones tomadas durante el desarrollo

Physical Memory Management

En primer lugar, el *naive_mm* actúa como un administrador de memoria muy básico. El código utiliza bloques de tamaño fijo (denominados BLOCK_SIZE), a diferencia de un sistema variable, esto simplifica la administración de la memoria. Más adelante, se implementó la función *my_free()* para gestionar la liberación de la memoria alocada. Sin embargo, esta implementación no hace un manejo de la fragmentación de la memoria.

Por otra parte, se implementó un buddy system al proyecto con la finalidad de hacer más eficiente las reservas de memoria. A diferencia del *naive_mm*, el *buddy_mm* reserva únicamente la cantidad de memoria solicitada al llamar a la función *my_malloc()*. Este sistema tiene forma de árbol, inicialmente se tiene en la raíz del árbol el total de memoria disponible y a medida que se le solicita alocar porciones de memoria, este bloque inicial se va partiendo en dos obteniendo en las hojas del árbol el nodo con el tamaño solicitado. Asimismo implementamos la función *align* en el buddy para alinear el tamaño que se desea reservar de memoria a la siguiente potencia de 2. Se tiene la función *my_free()* que analiza los nodos del árbol y dependiendo si este tiene descendientes o no, se lo marca como vacío y se actualiza la memoria asignada.

Procesos, Context Switching y Scheduling

Para la segunda entrega parcial, se desarrollaron dos archivos en la parte del kernel, processes.c y scheduler.c. Se implementó un Round-Robin con prioridades haciendo que las

aplicaciones que requieran mayor prioridad tengan un quantum más largo para evitar hacer colas multinivel. Además, por cuestiones de simplicidad se tomó la decisión de tener una cantidad máxima fija de procesos que por defecto es 32. De esta forma evitamos utilizar memoria dinámica excesiva.

Sincronización

En la tercera entrega parcial, se creó el archivo semaphore.c dentro de la sección kernel. Se crearon los *spinlocks* con la instrucción de assembler XCHG como se vió en la clase práctica de sincronización con el fin de cuidar las estructuras que se encargan de los semáforos: cada uno de los semáforos tiene su propio lock interno para asegurar atomicidad. Luego, cada semáforo tiene un nombre y una cola de procesos que están esperando a que se desbloquee. Utilizando este método evitamos la inanición ya que los procesos que se pusieron antes en la "fila" del semáforo recibirán el control del procesador antes que los demás.

Inter Process Communication

Para la entrega final se desarrolló un sistema de pipes anónimos que llena un vector con los file descriptors correspondientes a los extremos de escritura y lectura. Luego se modificó la system call de creación de procesos para que se le pueda especificar un descriptor de escritura y otro de lectura. De esta forma se logró que la escritura y lectura de pipes de los procesos de la shell sea transparente para los procesos involucrados.

Aplicaciones de User Space

En relación al desarrollo de las aplicaciones de User Space, tanto la shell como el comando *help* ya habían sido implementados en el trabajo práctico de "Arquitectura de Computadoras". Con respecto al physical memory management, para el comando *mem* se desarrolló una estructura (*mem_info_t)* que guardara la memoria total (*total_mem),* la memoria utilizada (*used_mem)* y la memoria libre *(free_mem).* Lógicamente, la impresión en userland se realizó a través de una syscall(*_sys_get_mem_info).*

Más adelante, en cuanto al comando *loop,* se decidió que el sleep fuera bloqueante con el objetivo de ahorrar recursos del CPU y que las esperas no dependan de la velocidad del procesador.

Por otro lado, para la implementación del problema de los filósofos (*phylo*) se necesitó la correcta coordinación de los procesos y resultó un gran desafío para poner varias de las implementaciones desarrolladas a prueba. Consecuentemente, se diseñó un sistema que coordina el acceso a los recursos compartidos (cubiertos) entre los múltiples filósofos. El proceso *"phylo",* se encarga de la administración en el creado y borrado de los filósofos así como también de la finalización del proceso.

Para garantizar la sincronización adecuada, se emplearon dos semáforos: uno general, denominado MUTEX, que asegura la exclusión mutua en las zonas críticas, y semáforos individuales para cada filósofo, denominados según el nombre del filósofo (*semName*). El semáforo general MUTEX se utiliza para proteger las operaciones críticas que modifican el estado global del sistema, como la adición y eliminación de filósofos. Por otro lado, el semáforo individual de cada filósofo se asegura de que cada uno espere su turno para tomar los tenedores sin necesidad de hacer busy waiting, lo que mejora el rendimiento y evita la espera activa, contribuyendo a la eficiencia del sistema.

System calls de lectura bloqueantes

Para implementar las llamadas al sistema de lectura bloqueantes se hizo uso del sistema de semáforos creado anteriormente. Se utiliza un semáforo para saber cuántos caracteres hay disponibles para leer y si no hay suficientes este se encargará de bloquear al proceso y desbloquearlo cuando haya nuevos datos disponibles.

Limitaciones

Physical Memory Management

La principal limitación del naive memory manager es que el tamaño del bloque de memoria es fijo por lo que si se le demanda un bloque más grande, devuelve NULL y si se le pide uno más chico, devuelve el tamaño de bloque establecido (4096 direcciones de

memoria). Esto causa un uso excesivo de memoria y la desventaja de no poder solicitar un bloque muy grande de memoria contigua.

Finalmente, este memory manager no chequea si hay dobles liberaciones y tampoco se asegura de que la dirección enviada sea dirección válida de un bloque.

Con respecto al buddy, una limitación es su dependencia en bloques de memoria de tamaño de potencia de dos. Es decir, si se solicita un tamaño de bloque que no cumple con este requisito, se debe ajustar al siguiente tamaño de potencia de dos. Consecuentemente, este manejo podría provocar fragmentación interna en caso de que las solicitudes de memoria no se alineen con las potencias de dos.

Procesos, Context Switching y Scheduling

La cantidad máxima de procesos es fija y es por defecto 32. Esta puede cambiarse en tiempo de compilación. Además, al no utilizar estructuras de datos complejas para manejar qué proceso será el siguiente, el sistema se puede sentir lento o con poca responsividad cuando hay muchos procesos en background (este efecto es empeorado también por la virtualización del hardware de QEMU).

Sincronización

La cantidad máxima de semáforos es fija y por defecto es 256. A su vez, el nombre de los semáforos también está fijo en un tamaño máximo de 64 caracteres.

Inter Process Communication

La cantidad máxima de pipes es fija y por defecto es 64. No se cuenta con un sistema de pipes con nombre.

Aplicaciones de User Space

Respecto al programa del problema de los filósofos (ejecutable con el comando "*phylo*"), la cantidad máxima de filósofos comensales fue definida como 15. Esto se debe a que cada filósofo es un proceso y con 15 filósofos se pueden tener 15 procesos más ejecutándose en background (son 15 puesto que los procesos init y shell siempre van a estar y

el límite de procesos simultáneos en nuestro programa fue definido como 32). Además tampoco se puede decidir que proceso filósofo eliminar de la mesa, siempre será el último que ingresó a comer.

Análisis con PVS-Studio

Al compilar el proyecto con el PVS studio notamos que los mensajes que nos arroja no son a causa de nuestras implementaciones. Hay ciertos falsos positivos tales como los mensajes que mencionan "*printf*", PVS cree que estamos utilizando la función de la librería "*stdio.h*". "`/root/Userland/SampleCodeModule/programs/programs.c 108 note V576 Incorrect format. Consider checking the third actual argument of the 'printf' function. The integer argument of 32-bit size is expected.`"

El resto de los warnings, notes y errores se encuentran en fragmentos de código propios del x64_Barebones por lo que se decidió ignorarlos.

Conclusión

En síntesis, este trabajo práctico nos ayudó a comprender de una manera más clara todo el contenido teórico de la materia Sistemas Operativos. El enfoque de no solo utilizar las herramientas provistas por un sistema operativo, sino también tener la responsabilidad de crear dichas herramientas es muy enriquecedor para complementar el estudio de mecanismos complejos como scheduling, pipes y semáforos.

Además, al ser lo suficientemente complejo, requirió que pasemos mucho tiempo utilizando herramientas de *debug* como GDB lo que mejoró mucho nuestra habilidad con este tipo de software.

Citas de fragmentos de código

Para el armado del naive memory manager, se utilizó como referencia el código hecho en la clase práctica por Ariel Godio, el lunes 16 de septiembre 2024.

Para el diseño de las funciones putForks, takeForks, philoActions y test del programa de filósofos comensales, se tomó como referencia los fragmentos de código de las paginas numero 37 y 38 de la teórica *"3T - IPC.pdf"*.