Instituto Tecnológico de Buenos Aires

Informe

72.11 - Sistemas Operativos - TP2

Integrantes:

Farbiarz, Bruno Ilan.
62644

Fernandez Dinardo, Juan Ignacio.
62466

Martinez, Tomas.
62878

1. Decisiones tomadas durante el desarrollo

1.1 Physical Memory Management

El mecanismo usado para la compilación exclusiva del memory manager elegido está presente en el archivo docker_build_command.sh (todas las líneas que involucran la variable TARGET). Luego pasa por el makefile del proyecto que hace make buddy o make all dependiendo del caso para luego ir al makefile del kernel donde se excluye la compilación de una archivo o otro dependiendo del caso.

1.1.1 Buddy Memory Manager

El administrador de memoria utiliza una serie de funciones y estructuras auxiliares para su funcionamiento. Se define un tamaño mínimo de asignación (MIN_ALLOC) y un tamaño máximo de asignación (MAX_ALLOC) para limitar el rango de tamaños de bloques de memoria admitidos. Esto permite controlar el tamaño mínimo y máximo de los bloques que se pueden asignar.

Para gestionar los bloques de memoria disponibles, se utiliza un arreglo de listas enlazadas llamado "buckets". Cada lista enlazada representa un tamaño de bloque específico y contiene los bloques disponibles en ese tamaño. Esto permite un acceso eficiente a los bloques de memoria según su tamaño.

Además, se utiliza un bitmap llamado "node_is_split" para marcar si un nodo en el árbol binario de asignación está dividido o no. Esto ayuda a realizar un seguimiento de la división de bloques y facilita la asignación y liberación adecuada de memoria.

Durante la asignación de memoria, se busca el tamaño de bucket adecuado para satisfacer la solicitud. A medida que se recorre la lista de buckets desde el tamaño inicial hasta el tamaño solicitado, se dividen los bloques de memoria disponibles en el proceso. Esto permite asignar el bloque más cercano al tamaño requerido, minimizando el desperdicio de memoria.

En el caso de la liberación de memoria, se busca el bucket correspondiente al tamaño del bloque liberado. Se comprueba si el nodo padre del bloque liberado está dividido y, de ser así, se une con su "buddy" (el otro bloque del mismo tamaño). Esto ayuda a consolidar bloques adyacentes y evitar la fragmentación de memoria.

1.1.2 Memory Manager

El administrador de memoria utiliza dos estructuras principales: MemoryManagerCDT y Heap_Node. La estructura MemoryManagerCDT contiene un puntero a la próxima dirección de memoria disponible para asignación. La estructura Heap_Node representa un bloque de memoria y contiene información sobre su tamaño, estado (libre o utilizado) y punteros a los bloques adyacentes.

La función createMemoryManager se encarga de inicializar el administrador de memoria. Recibe como argumento dos punteros: uno para la estructura MemoryManagerCDT y otro para la memoria administrada. Establece el puntero memoryManager al puntero proporcionado y guarda la dirección de memoria administrada como la próxima dirección disponible para asignación.

La función allocMemory se utiliza para asignar memoria. Recibe como argumento el tamaño de memoria que se desea asignar. La función recorre la memoria administrada en busca del primer bloque libre lo suficientemente grande utilizando un enfoque de "First Fit". Si encuentra un bloque que puede contener el tamaño solicitado, se realiza un "split" en el bloque si es necesario, dividiéndolo en dos bloques: uno que se asigna al tamaño requerido y otro que permanece como bloque libre. Luego, se marca el bloque asignado como utilizado y se devuelve un puntero al inicio del mismo. Este enfoque busca utilizar eficientemente el espacio de memoria y evitar fragmentación en el proceso de asignación.

Es importante mencionar que si el tamaño solicitado es cero o no queda suficiente memoria disponible, la función retornará NULL, indicando que la asignación de memoria no pudo realizarse.

La función free se utiliza para liberar la memoria asignada previamente. Recibe como argumento un puntero al bloque de memoria que se desea liberar. La función marca el bloque como libre y realiza una coalescencia con los bloques adyacentes libres, si los hay, para evitar la fragmentación de memoria.

La función dump muestra el estado actual de la memoria. Calcula la memoria total, la memoria utilizada y la memoria libre en función de los bloques de memoria asignados y libres. Luego, muestra esta información en la salida de video.

1.2 Procesos, Context Switching y Scheduling

Los procesos tienen una estructura con toda la información relevante a los mismos, para el scheduler y el context switching, como el ID, el estado, la prioridad y file descriptors. También tiene una estructura para el stack frame que guarda el estado de los registros del CPU durante la ejecución del proceso para poder permitir el context switching.

El scheduler implementado usa el algoritmo Round-Robin, con posibilidad de priorizar procesos dándoles más quantums para correr que el resto de los procesos. Se mantiene una lista con todos los procesos listos para ejecutar, agregando procesos al final de la lista y sacando procesos del comienzo de la lista para darles tiempo de ejecución. También tiene las funcionalidades de crear nuevos procesos, cambiar el estado y prioridad, y matar un proceso.

El scheduler se llama con cada timer tick para cambiar entre procesos. Con cada timer tick se llama el scheduler que actualiza la información de cada proceso y si se le terminó el tiempo de ejecución a uno, lo saca del comienzo de la lista y lo agrega al final, dando lugar al proceso siguiente para correr. También se programaron funcionalidades como el bloqueo y desbloqueo de procesos y el manejo de los file descriptors de los procesos que se habla más en profundidad en la parte de Inter Process Communication.

Dentro de la funcionalidad del mismo scheduler se crea un proceso “halt” que está programado para correr cuando no hay procesos listos para correr. Este solo ejecuta la instrucción “halt”, que pone el CPU en un estado IDLE hasta que un proceso esté listo para correr.

1.3 Sincronización

Para la sincronización del sistema operativo se define una estructura de semáforo que incluye un identificador, un valor que representa el número de recursos disponibles, una lista de identificadores de procesos bloqueados y un mutex para proteger el estado interno del semáforo. También mantiene un array de todos los semáforos activos en el sistema.

La función “sem_create” se utiliza para inicializar un nuevo semáforo con un identificador, nombre y valor inicial dado. La función “sem_open” se utiliza para vincular un proceso a un semáforo, ya sea encontrando un semáforo existente con el identificador dado o creando uno nuevo si no existe.

La función “sem_wait” es utilizada por un proceso para solicitar un recurso de un semáforo. Si el valor del semáforo es mayor que cero, se decrementa y la función retorna inmediatamente.

Si el valor del semáforo es cero, el proceso se añade a la lista de procesos bloqueados del semáforo y se pone en un estado de bloqueo.

La función “sem_post” es utilizada por un proceso para liberar un recurso de vuelta a un semáforo. Si hay algún proceso bloqueado en el semáforo, uno de ellos se desbloquea y se elimina de la lista de procesos bloqueados del semáforo. Si no hay procesos bloqueados, el valor del semáforo se incrementa.

La función “sem_close” se utiliza para desvincular un proceso de un semáforo. Si el semáforo no tiene más procesos vinculados, se destruye.

Esta implementación de semáforos asegura que no se corra riesgos entrando a regiones críticas de memoria por distintos procesos, sin condiciones de carrera, deadlocks ni busy waiting.

1.4 Inter Process Communication

En cuanto a la estructura de datos, se utiliza una estructura llamada PipeArray, que consiste en un array de Pipe. Esta estructura almacena información sobre el estado, el identificador y los índices de lectura y escritura de cada pipe. Al utilizar un array, se facilita la búsqueda y gestión de los pipes mediante índices.

En la versión original del código, se utilizaron semáforos en las funciones pipeRead y pipeWrite para sincronizar y proteger el buffer de los pipes. Sin embargo, al considerar que la syscall no puede ser interrumpida debido a ser una trap, se decidió comentar los semáforos. Esto se debe a que la incapacidad de interrupción por parte de otras operaciones asegura la protección del buffer en el kernel. Además, es importante mencionar que en la consigna se solicitó explícitamente que los semáforos sean protegidos dentro del kernel. Por lo tanto, se determinó que no era necesario agregar semáforos adicionales para proteger el buffer.

Las operaciones básicas en los pipes están implementadas mediante las funciones newPipe, pipeOpen, pipeClose, pipeRead y pipeWrite. La función newPipe se encarga de crear un nuevo pipe asignando un identificador único y estableciendo sus propiedades iniciales. La función pipeOpen permite abrir un pipe existente o crear uno nuevo si no existe. Por otro lado, la función pipeClose se utiliza para cerrar un pipe y liberar los recursos asociados una vez que ningún proceso lo está utilizando.

En cuanto a la lectura y escritura en los pipes, la función pipeRead lee un carácter del pipe y lo guarda en un buffer proporcionado, mientras que pipeWrite escribe una cadena de caracteres en el pipe. En la versión actualizada, se ha eliminado el uso de semáforos en estas funciones, lo

que implica que no se realiza una protección explícita del acceso al buffer durante las operaciones de lectura y escritura.

Por último, la gestión de índices y búsqueda de pipes se lleva a cabo mediante las funciones getPipeIdx y getFreePipe. La función getPipeIdx busca un pipe con un identificador específico y devuelve su índice si se encuentra, o -1 si no se encuentra. La función getFreePipe busca el primer pipe libre en el array y devuelve su índice, o -1 si no hay pipes libres disponibles. Estas funciones permiten localizar rápidamente los pipes y realizar operaciones sobre ellos.

1.5 User Space

1.5.1 Shell

La shell resulta ser la interfaz de usuario de nuestro sistema operativo. Para el caso de este trabajo práctico, se tuvieron que realizar muchas modificaciones sobre la base previa adquirida de nuestro trabajo en la materia de Arquitectura de las Computadoras. La principal diferencia radica en que ahora la shell es un proceso y cada comando que permite que se ejecute resulta ser un proceso hijo de esta. Sumado a eso, la posibilidad de comunicar comandos a través de pipe hizo que se cambiara casi por completo la forma en la que se parseaba cada línea escrita por el usuario sobre la shell. Además, también se agregó la posibilidad de que cada comando puede ejecutarse en background. Finalmente, resulta importante destacar que la shell es el único proceso en foreground del sistema operativo que el usuario no puede matar.

Para ejecutar un comando en background debemos escribir en la consola el comando que deseamos ejecutar seguido del caracter “&”, mientras que para conectar la salida y entrada de dos comandos utilizamos la siguiente estructura “comando1 | comando2”. Se puede cortar la ejecución de cada comando apretando la combinación de teclas Ctrl + C.

Dentro de algunos comandos, como wc, se realiza un bloqueo hasta recibir una señal de End Of File, esta se puede emitir a través de la combinación de teclas Ctrl + D.

1.5.2 Phylos

El problema de los filósofos comensales es un ejemplo de un problema de sincronización en la computación, donde se debe coordinar el acceso a los recursos compartidos (en este caso, los tenedores) para evitar condiciones de carrera y deadlocks.

Se define una estructura de filósofo que incluye un identificador, un estado (pensando, hambriento o comiendo) y un semáforo. Se mantiene un array de todos los filósofos en el sistema, y se utilizan semáforos para controlar el acceso a los tenedores.

La función “addPhylo” se utiliza para agregar un nuevo filósofo a la mesa. Crea un nuevo filósofo, inicia un nuevo proceso para representar el ciclo de vida del filósofo y agrega el filósofo al array de filósofos. La función “removePhylo” se utiliza para eliminar un filósofo de la mesa, terminando su proceso y liberando su semáforo.

El ciclo de vida de un filósofo se representa en la función lifecycle. Un filósofo intenta tomar los tenedores (representado por la función “attemptsForForks”), come durante un tiempo, luego libera los tenedores (representado por la función “releaseForks”) y piensa durante un tiempo. Este ciclo se repite mientras el programa esté en ejecución.

La función “checkForks” se utiliza para verificar si un filósofo puede tomar los tenedores y comenzar a comer. Un filósofo puede comer si está hambriento y ambos vecinos no están comiendo. Si un filósofo puede comer, su estado se cambia a comiendo y se libera su semáforo.

El manejo del agregar, remover y terminar el programa se maneja leyendo las teclas ‘a’ para el agregado, ‘r’ para remover y ‘q’ para la terminación del programa.

Finalmente, la función printAll se utiliza para imprimir el estado actual de todos los filósofos en la mesa. Esta función se ejecuta en un proceso separado y se actualiza continuamente mientras el programa esté en ejecución.

1.5.3 Tests

En cuanto a los tests brindados por la cátedra: test_mm, test_processes, test_priority, test_synchro y test_no_synchro. Estos se ven reflejados como aplicaciones de usuario a través de comandos posibles de ejecutar mediante el uso de la shell.

Para el test_synchro, se corre el comando “semTest n use_sem”, con n siendo la cantidad de veces que se va a intentar agregar o sumar a la variable global y use_sem = 0 si se quiere correr sin sincronización y use_sem = 1 si se quiere correr con semáforos.

1.5.4 Resto de los Comandos agregados

- ps: imprime información sobre los procesos presentes (no recibe argumentos);
- mem: imprime información sobre el estado de la memoria (no recibe argumentos);
- nice: cambia la prioridad de un proceso (recibe como argumentos el pid del proceso y un número entero que indica la nueva prioridad);
- block: bloquea o desbloquea un proceso (recibe como argumento el pid del proceso);
- kill: mata a un proceso (recibe como argumento el pid del proceso);
- loop: imprime su pid cada una cierta cantidad de segundos (recibe como argumento la cantidad de segundos);
- cat: imprime el argumento que le pasemos (un solo argumento);
- wc: imprime la cantidad de líneas del texto que se escriba por entrada estándar (no recibe argumentos);
- filter: imprime las vocales del texto que se ingrese por entrada estándar (no recibe argumentos);
- memTest: ejecuta el test de memoria (no recibe argumentos);
- procTest: ejecuta el test de procesos (no recibe argumentos);
- prioTest: ejecuta el test de prioridades (no recibe argumentos).

2. Instrucciones de Compilación y Ejecución

Por un lado podemos ejecutar el sistema operativo con un memory manager ordinario con el comando:

```
$ ./docker_build_command.sh
```

Por otro lado, si se quiere correr el programa con el BUDDY memory system, se debe correr el sistema operativo con el comand:

```
$ ./docker_build_command.sh BUDDY
```

3. Pasos a seguir para demostrar el funcionamiento de cada uno de los requerimientos

Antes es importante destacar que para conocer la totalidad de los comandos disponibles para su uso se deberá escribir el comando ‘help’ en la shell. Además, el mapeado de teclas se corresponde al estándar americano con la diferencia que para teclear el “|” debemos presionar shift derecho y la tecla que se encuentra a la izquierda del 1.

Teniendo esto en cuenta, los siguientes comandos ayudan a testear ciertos requerimientos del sistema:

- Para el testeo de ambos memory managers hay que escribir en la terminal ‘memTest’.
- Para el testeo del round robin scheduler escribir en la terminal ‘procTest’.
- Para el testeo del priority based round robin scheduler escribir en la terminal ‘prioTest’.
- Para el testeo de sincronización con semáforos escribir en la terminal ‘semTest’.
- Para el testeo del problema de los filósofos escribir en la terminal ‘phylo’.
- Para hacer un dump del estado actual de la memoria escribir en la terminal ‘mem’.
- Para hacer un dump de los procesos actuales escribir en la terminal ‘ps’.

4. Limitaciones

Hay algunas limitaciones que nos encontramos durante el desarrollo, a continuación las explicamos:

- No se pueden usar comandos con argumentos variables, por ejemplo cat recibe un solo parámetro, limitándose a ejecutarse con una sola palabra. Esto es así ya que los ipc, y comandos relacionados a ellos, fueron lo último que realizamos y antes no sentimos necesario hacer el agregado para que los comandos soporten argumentos variables;
- El uso de pipes en la consola no soporta cualquier combinación entre comandos. Principalmente se priorizo que se puedan usar algunas combinaciones como ‘cat ‘string’ | wc’, ‘cat ‘string’ | ‘filter’’. Sin embargo, hay otro tipo de combinaciones que sí funcionan como ‘help | filter’ o ‘help | wc’;
- Cuando usamos el pvs para testear hay muchos mensajes que surgen en la carpeta de toolchain y bootloader que ignoramos al no surgir de código hecho por nosotros. También ignoramos todos los relacionados a casteos de direcciones de memorias definidas previamente como constantes ya que de no castearlas, no solo pueden alterar el funcionamiento del programa sino que también recibimos warnings por parte del -Wall.

5. Problemas encontrados durante el desarrollo y cómo se solucionaron

Resulta complicado escribir esta sección ya que francamente se encontraron problemas en casi todas las fases del desarrollo. Por ejemplo, cuando se comenzó con los memory managers no sabíamos bien qué tipo de algoritmo resultaba conveniente elegir para la implementación ya que todos poseen sus pros y contras, es decir, no existe una "receta" que seguir. Nos surgian preguntas de estilo: ¿Qué tanta fragmentación podría estar generando esto? ¿Cómo hago para reducirla?. Un gran problema que tuvimos al principio fue el de como hacer para testar, por suerte uno de los integrantes del grupo pudo setear gdb y eso hizo que algunas cosas empezaran a fluir. Sumado a eso, un par de semanas más tardes la cátedra dio una clase acerca de cómo configurar este debugger y eso terminó por ayudarnos.

Si algo más vale la pena destacar, fue que, como era de esperarse, el desarrollo del scheduler fue lo más desafiante. Principalmente, poder armar correctamente tanto la estructura de los procesos del sistema operativo como la funcionalidad del mismo scheduler. Fue una constante traba que atrasó el trabajo varios días ya que con una base de código sacada de OsDev no se tenía en lo más mínimo la complejidad requerida para cumplir con el trabajo práctico. Fue un trabajo de ir agregando contenidos a los procesos y testear el scheduler con la herramienta de debuggeo gdb para poder darle forma y el funcionamiento acorde. Algo que nos tuvo varias horas debuggeando fue que nuestro programa arrojaba segmentations faults al iniciarse, esto es porque el timerRoutine empezaba a ejecutar el scheduler cuando ni siquiera todavía se había iniciado el memory manager o el proceso master. Solucionamos esto haciendo que la carga de la tabla IDT sea lo último por hacerse en Kernel, si bien capaz no es lo más limpio fue una solución rápida. Algo que se podría haber hecho capaz es haber jugado con las instrucciones cli y sti. Otro aspecto relacionado al scheduler que nos costó descifrar fue que debíamos tratar a los bloqueos causados por las interrupciones de teclado de forma diferente, en este caso mediante la implementación de un stack que guarde información sobre aquellos procesos que se bloquearon por una sys_read.

Podríamos decir tal vez que las implementaciones de los semáforos y pipes fueron dentro de todo directas y que no nos generaron mayores problemas. Aun así, costó un poco lograr la abstracción necesaria para que un proceso no sepa si está escribiendo o leyendo de teclado, pipe o pantalla. La implementación en Kernel de las funciones getCurrentOutFd y getCurrentInFd facilitaron esto, haciendo que se modifique el comportamiento de las syscalls de write y read según el valor retornado por estas funciones.

Una mención especial, pero no menos importante, el trabajo de Arquitectura de las Computadoras fue realizado por este mismo grupo pero con el agregado de un integrante más. Este chico tuvo un rol importante en las syscalls de read, write y el driver de teclado, por lo que al no contar con el, quien fue quien las ideó e implementó, también hizo que nos atrasaramos un poco al momento de modificar el código relacionado a esto último.

6. Citas de fragmentos de código reutilizado de otras fuentes

Donde más se citó código fue en la implementación del memory manager buddy, pero luego fue uso exhaustivo de algunas guías de osDev, las cuales tenían pseudocódigo e implementaciones básicas. Finalmente, también se realizaron consultas a chatGPT, pero el uso de las líneas de código proporcionadas por éste o Copilot no supusieron una gran relevancia en la implementación de nuestro sistema operativo.

- Buddy Memory Manager:
    - https://github.com/evanw/buddy-malloc/tree/master

- Memory Manager:
    - https://github.com/dreamos82/Osdev-Notes/blob/master/04_Memory_Management/02_Physical_Memory.md
    - https://github.com/dreamos82/Osdev-Notes/blob/master/04_Memory_Management/05_Heap_Allocation.md

- Scheduler:

    - https://github.com/dreamos82/Osdev-Notes/tree/c867b418f6efb453210a8c44d9b714729f2ee271
    - https://github.com/xinu-os/xinu/tree/master
    - https://github.com/dreamos82/Dreamos64