Trabajo Práctico Nro. 2
72.11 - Sistemas Operativos
2C 2022

Fecha de entrega: 07/11/2022

Magdalena Flores Levalle 60077
Pedro López Guzmán 60711
Camila Sierra Pérez 60242
Martín E. Zahnd 60401

Grupo 11

Índice

1. Decisiones tomadas durante el desarrollo

1.1. Kernel

1.1.1. Scheduler

Para que funcione el scheduler, lo que realizamos fue interceptar la interrupción de timer tick (20h), la cual ya no llama al dispatcher de interrupciones, sino que pasa por parámetro el rsp a una función llamada 'schedule'. Su trabajo es fijarse cual es el rsp que vino, guardarlo en el proceso que esta corriendo, elegir cual es el próximo proceso que tiene que correrse y devolver el rsp de ese proceso. De esta forma, desde la interrupción, antes de que termine, lo vuelve a poner en el stack pointer y cuando se hace un pop state y un return se levantan todos los registros de ese otro proceso. Esto funciona dado que, luego de una interrupción se guardan ciertos registros en el stack y además nosotros realizamos un push state que guarda los registros faltantes.

Por indicación de la cátedra utilizamos el algoritmo *Round Robin*, elegimos que el sistema de prioridad determine cuántos quantums va a correr un proceso. Para hacer esto se implementó una estructura de procesos con la información del mismo. Además se realizó una queue circular para tener un iterador del proceso que está corriendo, el cual va cambiando cada vez que se invoca a schedule y hay que hacer un cambio de contexto. Una de las ventajas de implementar la circular queue es que el tiempo máximo que tiene para encontrar el próximo proceso es de orden N (el peor caso, sino orden 1), siendo N la cantidad total de processos activos.

Por otro lado, dentro del scheduler hay funcionalidades para bloquear y desbloquear procesos.

Además, hay un proceso en halt. Este proceso solo espera una interrupción y corre si no hay ningún otro proceso listo. Esto se realiza para evitar que la pila este vacía o que quede en una interrupción infinita buscando el próximo proceso listo y no sepa que correr. Cuando se crea un proceso nuevo o pasa a otro, este deja de ejecutarse y no lo vuelve a hacer mientras haya otros procesos listos dentro de la queue.

Para crear un proceso, primero nos pide toda la información y luego lo que hace es elegir una zona de stack, alocar memoria y desde la dirección mas alta del stack inicializa un stack frame, donde se colocan todos los valores inciales que tiene un porceso para que cuando se termine la interrupción y le toque ejecutarse, levante todos esos valores. En el stack frame lo más importante es que el instruction pointer apunta a una funcion llamada main process wrapper, la cual se encarga de llamar a la función main del process, la función principal. Una vez que el proceso termina, se decrementa la cantidad de procesos listos que hay, se cambia su status a muerto y se fuerza un cambio de contexto para que sea eliminado de la queue.

Por último, desde kernel se inicializa el scheduler y se crea un proceso con la dirección de memoria del proceso de la shell en Userland.

1.1.2. Memory manager

Para este proyecto, optamos diseñar un memory manager que tiene todo el espacio de memoria y asume que en un principio está todo libre. Cada vez que se realiza un malloc, toma del final del espacio de memoria un bloque del tamaño que requiere y va asignando la memoria del final al inicio y en el medio va creando una lista de los bloques libres. Cada porción de memoria reservada tiene un header (ubicado contiguo al bloque) con la información de ese bloque de memoria. A medida que se libera memoria, los bloques libres se van enlazando apropiadamente.

Por otra parte, optamos por que el Buddy tenga una estructura de árbol binario. Además, contiene el memHeapAdress y memHeapStartAddr (los cuales se encuentran en el memory.h), que es donde empieza la memoria dinámica y tiene un tamaño *size*. Por otro lado, en el Kernel de K&R (del libro *The C Programming Language*), en cada bloque se ocupan en 64 bytes el header, por lo que a lo sumo pueden usarse `256 megabytes - 64 bytes` en total. En cambio, en el buddy se pueden utilizar los 256 megas completos debido a que el árbol se guarda al final del espacio de memoria. Es decir que está desde la dirección 0x160 hasta los 256 megas. Cuando termina el Heap, empieza el árbol del Buddy.

El funcionamiento del Buddy se da de la siguiente manera: al estar dividido en bloques de 64 bytes, cada vez que se requiere memoria, se fija a qué altura del árbol caería el tamaño que se necesita. Desde ahí, busca en esa altura el nodo más a la izquierda que esté libre, que no tenga ningún hijo en uso o cuyos padres estén marcados en uso también. Esto último para no entregárselo a otro usuario y sobreescribir memoria. La desventaja que tiene es que el tamaño que ocupa es constante y, salvo el peor caso, ocupa más espacio en memoria que otras implementaciones. Como ventaja tiene que, utilizando fórmulas matemáticas, puede conocerse rápidamente qué nodo del árbol usar y dónde está el espacio en la memoria. Es decir que, para encontrar un nodo que quiero reservar, me fijo el tamaño de memoria pedido por el usuario, se calcula a que altura va a estar del árbol, cual es el bloque mas chico que contiene ese tamaño y comienza a iterar sobre las hojas del subárbol que tienen ese tamaño. Puedo conocer el índice del arreglo de acuerdo a la hoja en la que estoy. Voy de izquierda a derecha recorriendo el árbol, buscando aquella que esté libre y cuyo padre también este libre. Por eso se marcan los padres, para que cuando el usuario requiera memoria y el Buddy deba encontrar un nodo, se fije que los padres estén libres desde el nodo que encontró hasta la raíz.

La máxima cantidad de niveles es 22, entonces el peor caso seria chequear 21 nodos que estén vacíos para reservar el nodo 22, lo cual no presenta un grado de gravedad notable. Como el Heap esta primero en memoria y atrás esta el árbol, entonces para encontrar la posición de memoria del Heap a la que corresponde determinado nodo del árbol, como conozco cuántos nodos hay a esa altura del árbol y cual es el primer índice de esos nodos. Al conocer el primer índice y el índice del nodo que se está reservando, entonces la diferencia es qué tanto se mueve el índice a la derecha en esa altura del árbol. También se conoce la cantidad de nodos en esa parte del árbol, entonces, se divide el total del Heap por la cantidad de nodos que se desplaza hacia la derecha.

1.1.3. Semáforos

Los semáforos están implementados usando *stdatomic*, la biblioteca estándar (aunque no era obligatorio, hicimos de cuenta que había que hacer una implementación suponiendo un kernel reentrante). Los semáforos cada vez que hacen un wait guardan el pid de los procesos que bloquearon, y después de guardar el PID hacen atómicamente una resta del valor del semáforo y de la cantidad de procesos en espera. Por otro lado, el post primero desbloquea el primer proceso que encuentra y después hace también atómicamente un add en el valor del semáforo. La cola de PIDs bloqueados tiene un tamaño fijo de 32 que es un buffer circular y cada semáforo a su vez puede tener como mucho 32 procesos con un wait. Esto se realiza para no alojar y desalojar memoria constantemente, lo que permite agilizar todo. El primer proceso que se canto el wait lo guardo al final del buffer y cuando ese proceso sea el primero en haber hecho el wait es el primero que se libera, y así, dependiendo del orden de llegada (es una cola FIFO en un buffer circular). En cuantoa a Userland, cuando se hace un wait, como para hacer todo lo demás tengo que bloquear un semáforo, se hace justo antes de desbloquear el mismo. Pero cuando se hace un post, para evitar un deadlock se intenta actualizar userland. Si no se puede se asume que está colgado en medio del

wait algún proceso y ese proceso va a actualizar el semáforo cuando le toque. Toda la información de userland se imprime y maneja allí, lo único que hace el kernel es generar y mantener actualizada una estructura (soseminfo_t) que tiene información pública útil para userland, nada se imprime desde el kernel.

1.1.4. Pipes

Los pipes tienen la estructura en Kernel, desde donde se le envía la información a Userland. Reciben los dos fd empezando del 3 y 4, para reservar el 0, 1 y 2 para stdin, stdout y stderr respectivamente (aunque no utilicemos este último).

Cada vez que se crea un pipe se devuelve el par de fd. Read y write funcionan como en posix y close cierra de a un fd. Cuando los dos fd de un pipe están cerrados, se cierra el pipe completo. Si no se cierran ambos fd, los pipes quedan guardados en una lista de pipes (un buffer circular con un límite de 32 pipes) en la que van quedando y cada vez que reservamos un pipe se fija si el fd corresponde a un pipe ya cerrado. Si ambos fd están cerrados entonces se borra el pipe, queda inactivo y se puede volver a utilizar ese fd. Siempre se intenta que el fd que se asigna a un pipe sea el menor, para ambos casos. Hay lista de fd asignados, se consulta cual es el menor sin asignar para dárselo a un pipe cada vez que se crea uno.

En cuanto a la implementación, si bien teníamos un semáforo para read y otro para write, desestimamos el de write dado que tener ambos era una solución muy limitada. Como no puede haber dos escrituras concurrentes dentro de una interrupción, no hace falta semáforo de write. Asumimos que siempre se va a poder escribir. Por el mismo motivo, no va a haber un proceso que escriba y al mismo tiempo otro que lea. Sí utilizamos el semáforo de read por que si un proceso quiere leer de un pipe que esta vacío ahí si se bloquea hasta que esté lleno.

1.1.5. Funciones y tests

Todas las funciones del kernel se llaman so + nombre de la función y tienen un prototipo igual o similar al de POSIX.

Algunos cambian ligeramente, por ejemplo en semáforos 'seminit' en POSIX tiene un segundo argumento, pero nosotros no lo utilizamos así. Sino la nomenclatura y los prototipos coinciden con los de POSIX.

Además, hay tests escritos utilizando CuTest para malloc, free, calloc, (tanto utilizando el memory manager de K&R como el BUddy), semáforos y pipes. Estos tests se compilan aparte.

'make tests' y 'make buddyTest' son los comandos a utilizar.

1.2. Userland

1.2.1. Shell

Rehicimos la shell preexistente debido a que la anterior no era escalable fácilmente ni nos permitía trabajar de manera cómoda sobre los comandos a realizar en este trabajo y, aprovechando que contamos con manejo de memoria, creamos una shell nueva. Esta shell contiene una *Queue* para el historial de líneas, con un tamaño máximo. Si este último se excede, se borra la última, se coloca al principio y se renderizan en la pantalla dependiendo de la posición que tengan. Además, posee un linea de comandos en la parte inferior de la pantalla, donde se colocan los caracteres de

los comandos. Cuando se ingresa un comando, una vez que se ingresa *Enter*, se parsea el comando y se ejecuta dependiendo de qué se ingresó. La Shell se inicia al empezar a ejecutarse Userland, lo cual inicializa la lista y todo lo necesario para funcionar. A partir de ese momento, queda en un *game loop*, en el cual se hace un getChar, por lo que esta se bloquea hasta que se aprieta una tecla. Una vez ingresado Enter, se llama a una función para procesar el comando y si se ingresa *backspace* se borra el último caracter. Además, se implemento una lista de comandos, en la que cada nodo tiene además de la conexión, el nombre del comando y un puntero a una función. Este puntero siempre es del mismo tipo, devuelve int y sus parámetros son int y char**, que corresponden a argc y argv, simulando un main. Estas tipo de funciones son las que crean un proceso. Una vez que se procesa un comando, dentro de la Shell se separa el comando en tokens y el primer token es el que define el comando que se debe ejecutar, se realiza un get en base al nombre y se devuelve la función. Una vez que se obtiene la función, si es que existe, se crea un proceso donde se le pasan los parámetros que se especificaron en el comando y se llaman a las syscalls necesarias para crear dicho proceso. Si es un proceso del foreground, la Shell hace un wait al proceso y se bloquea. En cambio, si el proceso fuera background, se sigue ejecutando. Para delimitar si un proceso es foreground o background, se hace al igual que en Unix, se le agrega un "&" al final del nombre del comando de dicho proceso.

Con respecto a los pipes dentro de la Shell, se parsea el comando entero, se divide en tokens y si un token corresponde a un símbolo de pipes (|), se crea un pipe nuevo. Al commando detras del caracter especial se lo crea utilizando el *write end* del pipe como su *stdout*. Luego, al comando por delante del caracter especial se lo crea utilizando el *read end* del pipe como su *stdin*. La única excepción, es que al primer parámetro siempre se le pasa 0 en su *stdin*, que es el keyboard, y al ultimo parámetro siempre se le pasa el 1 en su *stdout*, que seria la shell, entonces se crea una cadena de pipes. Una vez que termina de ejecutarse todo, se liberan todas las pipes que hagan falta. Vale la pena aclarar que si se hace un sistema de pipes y se ejecuta un proceso de foreground, se va a bloquear el resto de procesos porque la shell va a esperar a que ese proceso termine para crear el proceso que sigue. Por lo tanto, no hay concurrencia si todos los procesos son foreground. En el caso de que se haga un proceso de background, el segundo proceso sí se ejecuta antes de que termine el primero.

2. Instrucciones de compilación y ejecución

Las instrucciones de compilación y ejecución se encuentran en el archivo *Readme.txt*.

3. Funcionalidades

3.1. Comandos

- help: Muestra una lista con todos los comandos disponibles.
- mem: Imprime en pantalla el estado de la memoria.
- ps: Imprime en pantalla la lista de procesos con sus propiedades.
- loop: Imprime el ID del proceso actual cada 3 segundos. Este proceso cuando está esperando se bloquea, ya que usa *sleep*.

- activeLoop: Funciona como el comando loop mencionado anteriormente, pero este cuando está esperando no se bloquea, ya que funciona con un ciclo *for*.
- kill [PID]: Mata un proceso dado su ID.
- nice [PID, PRIORIDAD]: Cambia la prioridad del proceso PID al valor dado.
- block [PID]: Cambia el estado de un proceso entre bloqueado y listo dado su ID.
- sem: Imprime en pantalla la lista de semáforos con sus propiedades.
- cat: Imprime en pantalla la entrada estándar. Para ingresar un EOF en stdio y en este comando, debe presionarse la tecla de *esc*.
- wc: Cuenta la cantidad de lineas en el input.
- filter: Filtra las vocales del input.
- pipe: Imprime en pantalla la lista de pipes con sus propiedades.
- phylo: Comienza el problema de los filósofos comensales.
- test_mm: Test del memory manager brindado por la cátedra.
- test_prio: Test de prioridad brindado por la cátedra.
- test_processes: Test de los procesos brindado por la cátedra.
- test_sync: Test de sincronización brindado por la cátedra.

Además, el proyecto cuenta con la opción de ejecutar un programa en el background si se ejecuta agregando el símbolo “&” al final de un comando y permite conectar 2 procesos mediante un pipe agregando el símbolo “|” al final de un comando, al igual que en bash.

3.2. Capturas del proyecto

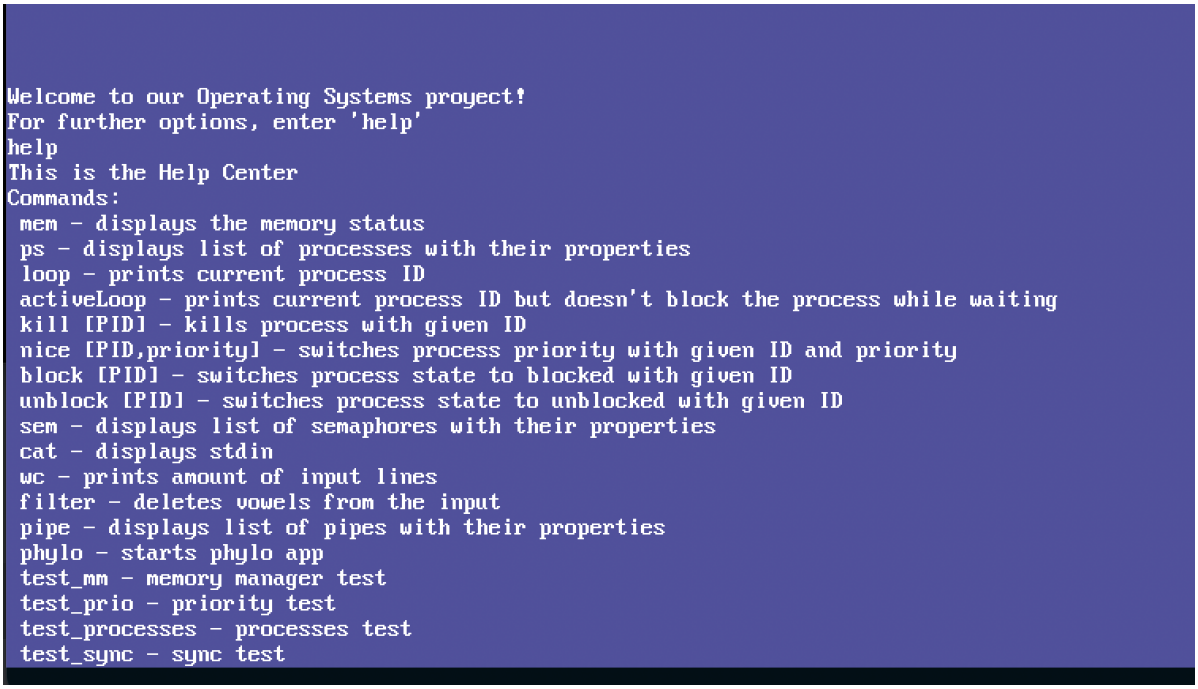

Figura 1: Comando help

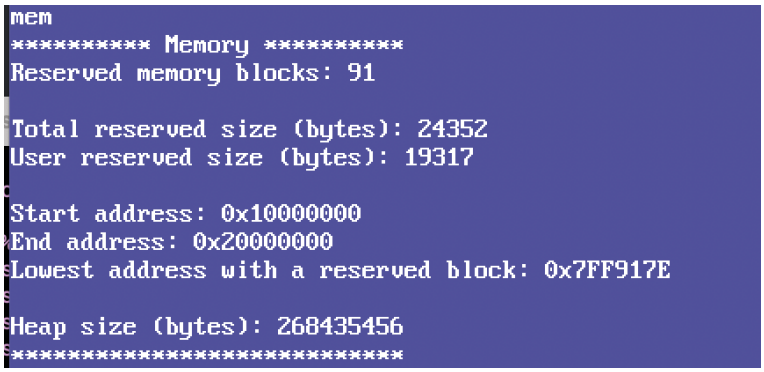

Figura 2: Comando mem

Figura 3: Comando ps

```
 ps - displays list of processes with their properties
 loop - prints current process ID
 activeLoop - prints current process ID but doesn't block the process while waiting
 kill [PID] - kills process with given ID
 nice [PID,priority] - switches process priority with given ID and priority
 block [PID] - switches process state to blocked with given ID
 unblock [PID] - switches process state to unblocked with given ID
 sem - displays list of semaphores with their properties
 cat - displays stdin
 wc - prints amount of input lines
 filter - deletes vowels from the input
 pipe - displays list of pipes with their properties
 phylo - starts phylo app
 test_mm - memory manager test
 test_prio - priority test
 test_processes - processes test
 test_sync - sync test
loop
Hello, this is the LOOP command. Current PID is 102
Hello, this is the LOOP command. Current PID is 102
Hello, this is the LOOP command. Current PID is 102
Hello, this is the LOOP command. Current PID is 102
Hello, this is the LOOP command. Current PID is 102
Hello, this is the LOOP command. Current PID is 102
Hello, this is the LOOP command. Current PID is 102
Hello, this is the LOOP command. Current PID is 102
loop
```

Figura 4: Comando loop

```
 activeLoop - prints current process ID but doesn't block the process while waiting
 kill [PID] - kills process with given ID
 nice [PID,priority] - switches process priority with given ID and priority
 block [PID] - switches process state to blocked with given ID
 unblock [PID] - switches process state to unblocked with given ID
 sem - displays list of semaphores with their properties
 cat - displays stdin
 wc - prints amount of input lines
 filter - deletes vowels from the input
 pipe - displays list of pipes with their properties
 phylo - starts phylo app
 test_mm - memory manager test
 test_prio - priority test
 test_processes - processes test
 test_sync - sync test
activeLoop
Hello, this is the ACTIVE LOOP command. Current PID is 102
Hello, this is the ACTIVE LOOP command. Current PID is 102
Hello, this is the ACTIVE LOOP command. Current PID is 102
Hello, this is the ACTIVE LOOP command. Current PID is 102
Hello, this is the ACTIVE LOOP command. Current PID is 102
Hello, this is the ACTIVE LOOP command. Current PID is 102
Hello, this is the ACTIVE LOOP command. Current PID is 102
Hello, this is the ACTIVE LOOP command. Current PID is 102
Hello, this is the ACTIVE LOOP command. Current PID is 102
activeLoop
```

Figura 5: Comando active loop

Figura 6: Comando sem

Figura 7: Comando cat ingresando 'hola'

Figura 8: Comando wc

Figura 9: Comando filter ingresando 'hola'

Figura 10: Comando pipe

Figura 11: Comando phylo

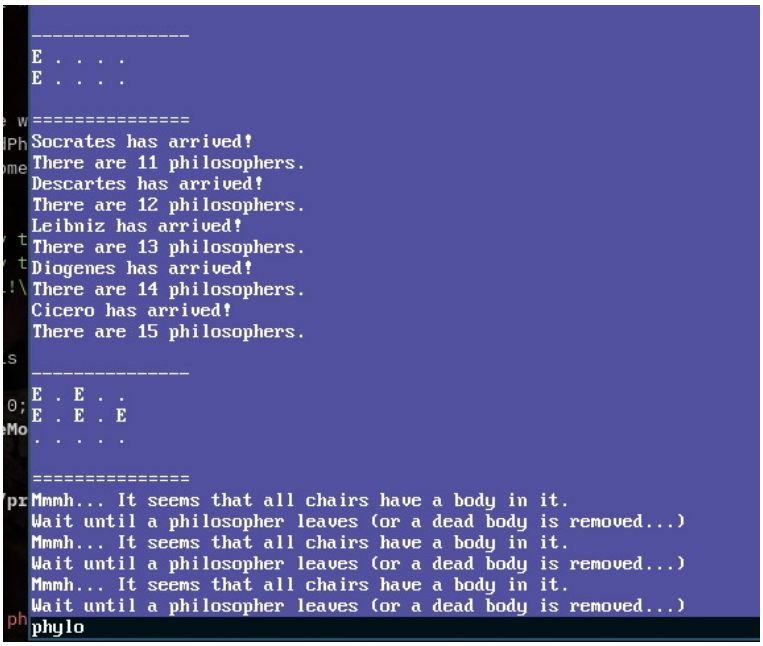

Figura 12: Agregar filósofos

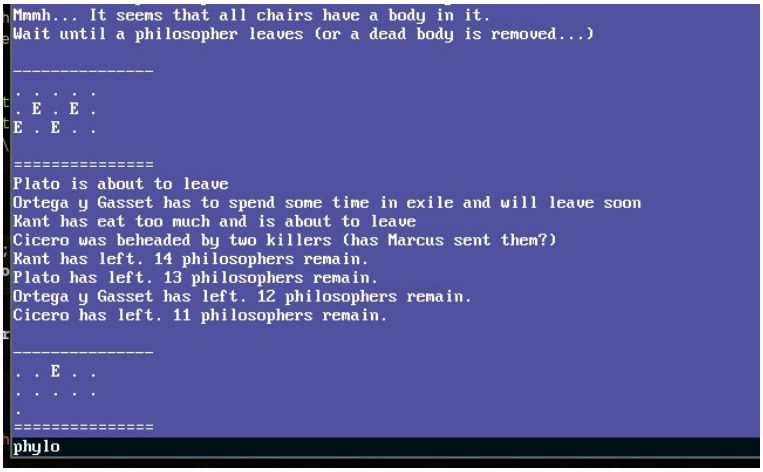

Figura 13: Eliminar filósofos

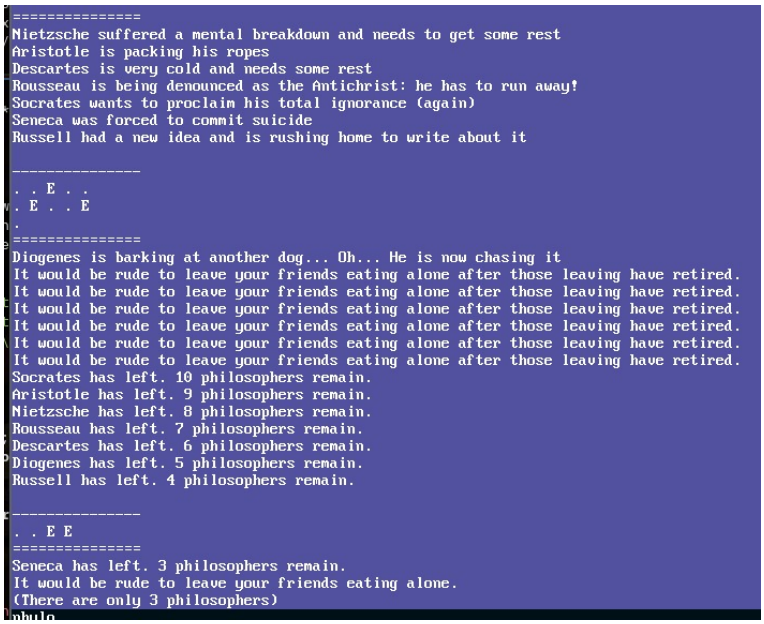

Figura 14: Eliminar filósofos

4. Limitaciones

El buddy del memory manager se implementó utilizando un árbol binario por una cuestión de practicidad a la hora de programarlo. Otra opción era hacerlo con listas y hubiese costado mas porque habría que recorrer toda la lista hasta el final siempre que se necesite ver si un bloque no está reservado e ir guardando un header en un lugar del espacio reservado y perder espacio. Por esto mismo, se optó por utilizar un árbol binario al cuál se lo trata como si fuese un arreglo. La desventaja de esto es que, como no todos los nodos están marcados (salvo que se llene todo el heap), se desperdicia espacio. Otra limitación del proyecto es en el comando ps, el cuál muestra información de un máximo de 32 procesos. Sólo muestra los datos de los primeros 32 procesos.

El comando *phylo* implementa el problema de los filósofos comensales. Permite agregar y quitar comensales, con un mínimo de 3 filósofos y un máximo de 15, por una decisión de diseño. Se pueden agregar mil veces filósofos, una vez alcanzado ese número, es indeterminado el comportamiento.

En el test del memory manager, (brindado por la cátedra y adaptado para que funcione con el trabajo) se tomó la decisión de que tenga un máximo de 128 bloques y la memoria total contiene la mitad de esta, ya que si se utilizaba toda, corría el riesgo de ocurrir un page fold.

5. Problemas encontrados durante el desarrollo y cómo se solucionaron

Tuvimos un problema al alocar memoria para los argv desde la consola, si el proceso estaba en foreground no pasaba nada porque la shell se bloqueaba, y una vez que terminaba el

proceso la shell hacia free de esos agumentos. El problema era si estaba en background, donde el proceso que estaba leyendo esos argumentos cambiaba el scheduling a la shell, en la shell se hacia free y quizás el proceso más adelante iba a necesitar los argv, por eso no estaba bien que sean los mismos. Tratamos de hacer que cuando se cree un proceso se haga una copia dentro del kernel, pero por algún motivo se corrompía todo si hacíamos un malloc desde kernel y lo pasábamos como argumento (argv) a un proceso.

Para solucionar esto, decidimos hacer una lista 'resource list' que desde la terminal cada vez que se corre un proceso se guarda en esa lista el argv y argc y la función de la lista es cada 10 teclas apretadas recorrer los nodos, cada nodo tiene un proceso y se fija si ese proceso está activo (con una syscall). Si el proceso no existe o esta muerto, libera los recursos.

Una posible mejora a la implementación actual, es que el proceso de *garbageCollector* sea un proceso aparte, no una función que se llame desde la Shell cada 10 teclas apretadas. Con esta acción, en nuestro proyecto se hace una limpieza de los recursos de los procesos que invocó la Shell, pero idealmente esto debería realizarse como un proceso separado.

6. Citas de fragmentos de código reutilizados de otras fuentes

Para el diseño del memory manager del proyecto, nos basamos en el malloc del libro *The C Programming Language*, de Brian Kernighan y Dennis Ritchie. Además, tiene código basado en el heap_4 del memory manager de FreeRTOS (https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/main/portable/MemMang/heap_4.c). El memory manager del Kernel de K&R hace una lista enlazada circular de los espacios libres y va reservando el espacio que le pide el usuario.

7. Modificaciones realizadas a las aplicaciones de test provistas

En el test de prioridades, se asignaron los valores 1 a LOWEST, 3 a MEDIUM y 5 a HIGHEST. Además, realizamos una lista de argv's, para realizar un free cuando termine el test. Por último, se agregó un sleep ente los cambios de prioridad y se asignaron que cada uno de los loops que imprimen tengan un busy wait con 25 millones de iteraciones para poder visualizar correctamente los cambios de prioridades.

En el test de procesos, se definió que se le pase por argumento la cantidad de procesos que el usuario quiere crear y aleatoriamente los va bloqueando, desbloqueando y matando. Cuando crea los procesos, se imprime un ps. Una vez que están muertos todos los procesos, termina y ejecuta nuevamente ps para mostrar los cambios realizados.

En el test de sincronización se fijó como cantidad total de pares de procesos sea 5. Para el cambio de contexto.

Para el test de memoria, se definió un tamaño máximo de 48 MB (de 64 MB en total). Se debe pasar por parámetro el porcentaje de esa memoria que desea usarse. El test corre indefinidamente por lo que es recomendable ejecutarlo en background.

8. Repositorio

Repositorio del trabajo práctico: mzahnd/sistemas-operativos-tp2

Hash del commit: `27b5a3c982795e365a966fd7e8cd7b33051c25f5`

Branch del commit: `master`