Informe TP 2 Sistemas Operativos:
Construcción del Núcleo de un
Sistema Operativo y Estructuras de Administración de Recursos.
Grupo 5

Participantes:
Matías Manzur, 62498
Franco Rupnik, 62032
Mauro Báez, 61747

Fecha de Entrega: 7/11/2022

Introducción

En el presente informe se darán a conocer decisiones de diseño, inconvenientes y cambios que consideramos necesarios a la hora de desarrollar el segundo trabajo práctico de esta materia.

Decisiones Tomadas

En la presente sección discutiremos únicamente aquellas decisiones de diseño no obvias, esto es, resoluciones las cuales requirieron distintas consideraciones antes de llevarlas a cabo. Las separaremos en sub-secciones para hacer más simple su entendimiento.

Memory Manager Elegido

Haciendo uso de una variación del algoritmo de alocación y liberado de memoria que se presenta en el libro "The C Programming Language" de K&R (Sección 8.7 del mismo) realizamos nuestra implementación del algoritmo. Las variaciones que presenta nuestro algoritmo son pequeñas, una de ellas es no hacer uso de una lista encadenada circular, ya que esa implementación está orientada a pedir más memoria al sistema operativo hasta acabarse la misma. En cambio, nuestra implementación ya tiene toda la memoria disponible para usar, es decir no hay más memoria para darle para alocar.

Por otra parte, decidimos dividir la memoria en bloques de tamaño del struct header, el mismo tiene la información del tamaño y un puntero al siguiente bloque. De esta manera, es más sencillo mantener el alineamiento de la memoria mientras funciona el algoritmo. También, debido a que el header es parte de la memoria que tenemos para administrar, hacemos un simple cálculo de la cantidad de bloques necesarios para proveer al usuario el tamaño de memoria que solicitó y le sumamos uno por el bloque que usamos nosotros para el header. Entonces, de la memoria que reservamos, en el primer bloque ponemos el header, y a partir del segundo es lo que le entregamos al usuario. Esto último, hace que la liberación de memoria sea más fácil pues solo es necesario moverse desde la dirección provista al usuario una cantidad sizeof(struct header) (= 1 bloque) hacia atrás y tendremos acceso al tamaño de la memoria que le dimos.

Memory Manager Buddy

La estructura del Buddy se basa en N listas de bloques de memoria que vamos a entregar al usuario. Cada lista "i" almacenará los bloques libres de tamaño 2^i bytes. A estos grupos de bloques separados por su tamaño los llamamos divisiones. Entonces, por ejemplo en la división 8 tendremos los bloques de tamaño 2^8=256 bytes, y en la división N-1 tendremos únicamente un bloque el cual ocupa toda la memoria (2^{N-1} bytes).

Para ello, únicamente guardamos a partir del espacio de memoria asignado en la inicialización de memoria un arreglo de N punteros los cuales hacen referencia a los primeros "buddy nodes" de cada lista. En estos "buddy nodes" guardamos la información pertinente de cada bloque disponible (el previo y el siguiente de la lista además de una identificación de a qué división pertenece). Si bien, los buddy nodes ocupan los primeros 17 bytes de los bloques a administrar, cuando se entregan al usuario sólo se reserva 1 byte para guardar la división a la que pertenece el bloque, y los otros 16 bytes (con los punteros al previo y al siguiente de la lista) ya no los necesitamos por lo que se los damos al usuario para que los use (junto con el resto del espacio solicitado). Es por esto que el tamaño mínimo de memoria que se puede pedir es de 32 bytes, (si se pide de menos, igualmente se reservarán 32 bytes) pues si fuese más pequeño el buddy node del bloque anterior podría solaparse con la memoria otorgada al usuario.

Obviamente no guardamos todos los bloques posibles ya que eso requeriría cantidades desorbitantes de memoria, en lugar de eso, empezamos únicamente con un bloque en la división N-1. Por ejemplo, si alguien pidiera entre un octavo y un cuarto de la memoria total, entonces el buddy encontraría la división N-3 vacía, así que se fijaría en la división N-2 cuyo puntero también apunta a NULL, por lo que dicha división también se encuentra vacía. Finalmente llegaría a la división N-1 la cual tiene solo un bloque, al mismo lo partiría en dos (sacándolo de la división N-1), agregando ambas mitades a la división N-2, y sucesivamente, una de estas mitades es removida y

separada a la mitad (¼ de la memoria). De estos dos cuartos de memoria con los que contamos, uno es agregado a la lista de división N-3, mientras que su "buddy" es devuelto al usuario que pidió la memoria.

Para el free, hacemos el viaje inverso, nos fijamos la división de memoria del puntero que nos enviaron, en dicha división vemos si se encuentra su buddy, si no está, significa que está siendo usado, por lo que agregamos a la lista de la división el espacio de memoria devuelto, el cual espera por su buddy. En cambio, si se encuentra el buddy del puntero devuelto, los unimos y removemos quien estaba en la lista, para unificar el bloque y que pase a estar en la división siguiente, ahora debemos realizar el mismo procedimiento pero en esta otra división.

Scheduler

Como se muestra en el *Diagrama 1* a continuación, el scheduler cuenta con 5 listas doblemente encadenadas (una para cada prioridad posible de los procesos) de estructuras de tipo "nodePCB_t".

Estos "nodePCB" tienen un puntero al anterior "prev" y al siguiente "next" para formar la lista doblemente encadenada. Además, cuentan con un puntero a la PCB donde se guarda toda la información relevante de cada proceso (nombre, pid, ppid, argumentos, stack pointer, estado, razón de bloqueo, prioridad, tabla de fds, entre otros). Por otro lado, cada uno cuenta con una lista doblemente encadenada de sus procesos hijos, por lo que el nodePCB del padre tiene un puntero al nodePCB del primer hijo "children", y los nodePCB de los hijos están encadenados entre hermanos a través de los punteros "prevSibling" y "nextSibling".

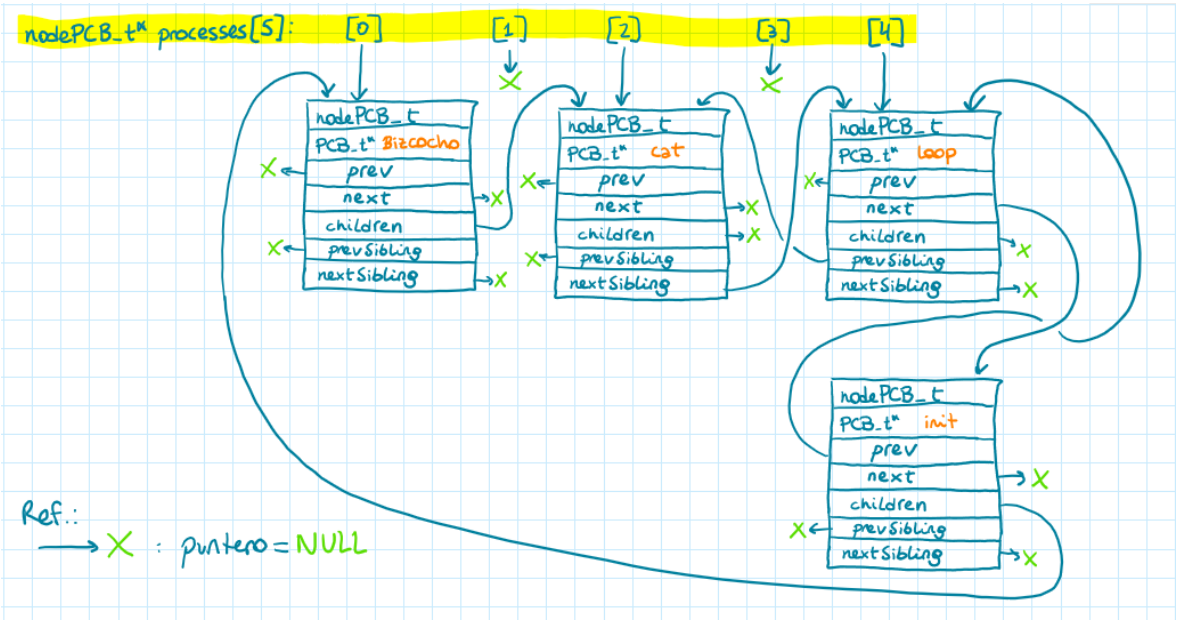

Diagrama 1: Ejemplo del estado de la estructura del scheduler cuando se inició el bizcocho y se ejecutaron "loop | cat" en el mismo. *(Aclaración: loop en realidad se ejecutaría con prioridad 2 como cat, pero se puso con prioridad 4 para ilustrar mejor el funcionamiento de las listas)*

Se implementó un sistema de scheduling Round-Robin en el que los procesos de mayor prioridad se le otorgan más quantums seguidos que a los de menor prioridad. Cada interrupción del timer tick es un quantum. Cuando se terminan los quantums correspondientes a un proceso, el scheduler busca al siguiente iterando por las listas en orden. Las funciones de kill(), block() y yield() setean los quantum restantes del proceso en cero y fuerzan una interrupción del timer tick, para que el scheduler tenga que hacer un cambio de contexto.

Se decidió separar los procesos en una lista por cada prioridad para encontrar los procesos más rápidamente cuando ya se conoce la misma, y no tener una sola lista larga con todos los procesos. Al inicializar el scheduler, se agrega un proceso "init" cuya función será detallada más adelante.

Para el cambio de contexto y la inicialización del stack de un proceso, se aprovechó la implementación que habíamos realizado en el TP de Arquitectura de Computadoras, la cual es muy similar a la descrita en la clase práctica de hace unas semanas.

A cada proceso se le otorga un espacio de memoria para su stack de 4KB, mientras que el máximo de procesos no está limitado por una constante, pues se usan listas dinámicas.

En la PCB de cada proceso guardamos el estado del mismo: READY(esperando su turno), BLOCKED o FINISHED. Para el caso de BLOCKED, además guardamos la razón por la que fue bloqueado, junto con el id del hijo/pipe/semáforo según corresponda. Las razones posibles de bloqueo son: lo pidió el userland arbitrariamente, leer de un pipe vacío, escribir en un pipe lleno, esperar un hijo, o esperar un semáforo. El scheduler tiene una lista de punteros a nodePCB de los procesos bloqueados por cada razón de bloqueo, para que otras partes del Kernel puedan pedirle al scheduler que desbloquee a un proceso si estaba bloqueado por X razón o desbloquear a todos los procesos que estén bloqueados por X razón.

Cuando un proceso termina con un exit() o lo matan con un kill(), se pasa su estado a FINISHED, pero no se elimina de las listas todavía. Esto es por si el padre quiere esperar por él con un waitchild(). Si el padre hace un waitchild() y el hijo todavía no terminó, se bloquea hasta que haya terminado. Cuando el hijo terminó, ahí sí se recupera el status code con el que terminó el hijo y se lo remueve de las listas del scheduler.

En el caso de que el padre termine y no haya esperado a los hijos, estos se vuelven procesos huérfanos y pasan a ser hijos del proceso init. El proceso init, además de funcionar como un idleProcess, se encarga de hacer waitchild() constantemente para ir removiendo de la lista procesos huérfanos que hayan terminado.

Si los procesos huérfanos no terminan, ya sea porque el proceso padre no se encargó de eliminarlos o fue terminado con un kill() externo, estos seguirán corriendo hasta que alguien les haga un kill. Es decir, los hijos no mueren con el padre, a no ser que el padre los mate. Esto se decidió hacerlo así porque quizás se quiere que los hijos sigan corriendo aunque el padre haya terminado.

Semáforos

La implementación de semáforos se llevó a cabo utilizando la instrucción de assembly ‘xchg’ como fue recomendado en clase para evitar posibles condiciones de carrera al modificar ciertas variables clave las cuales eran compartidas entre varios semáforos o podían ser utilizadas por varios procesos. El bloqueo de los semáforos se implementó como una queue, en lugar de “despertar” a un proceso random, siempre despertamos al primero que se había quedado bloqueado, este tipo de implementación fue recomendada en una clase práctica.

Toda la información del semáforo es mantenida en el Kernel, desde Userland lo que se tiene después de realizar una inicialización de un semáforo, enviándole un nombre único, es el id del semáforo creado o semáforo que tiene el mismo nombre que el ingresado. Termina siendo una implementación de Named Semaphores con ids en lugar de punteros. Tanto para la lista de semáforos como para los procesos bloqueados por un semáforo particular, se utilizó el ADT de lista doblemente encadenada.

Para la terminación de los semáforos, a diferencia de Linux, no contamos con funciones close y destroy por separado, sino que directamente un proceso realiza un close de un semáforo en cuestión y el mismo es destruido. Es responsabilidad del usuario que ningún proceso quede bloqueado cuando el close es llevado a cabo.

Pipes

Para el desarrollo de pipes, realizamos una implementación de file descriptors para poder unificar el uso de la entrada/salida y los pipes, de forma que para un proceso pueda usar los pipes o los medios estándar de forma transparente. Los file descriptors los separamos en 2 tipos, el file descriptor que le damos a Userland y el file descriptor que se usa en Kernel (el cual llamamos fileID), la diferencia entre ambos es que el provisto al usuario (Userland) es simplemente un entero que hace referencia a la posición dentro de la tabla de file descriptors del proceso (ya que cada proceso, como mencionamos antes, tiene su propia tabla de file descriptors). En esta tabla de file descriptors, se tiene un fileID y un modo(Lectura o Escritura), el fileID hace referencia a una tabla interna que tenemos dentro del Kernel donde se indica el canal donde se quiere leer/escribir ya sea un pipe o una de las entradas/salidas, mientras que el modo solo se usa realmente al momento de escribir o leer pipes (si se decide leer

con un fd que solo acepta escritura no será posible realizar la lectura). Esto último fue una decisión de diseño para poder implementar de alguna manera el hecho que los pipes sean unidireccionales, es decir cuando se abre un pipe para lectura solo te provee permisos de lectura y viceversa con escritura.

Ahora para el manejo de los pipes en sí, decidimos implementar algo parecido a named pipes. Para poder crearlos primero es necesario hacerlo con sys_mkpipe("Nombre") y darle un nombre. Subsecuentemente, sería necesario realizar un sys_open("Nombre", modo_deseado), esto devuelve un file descriptor que luego deberá ser usado en conjunto con sys_read/sys_write para realizar lo que se desee. Cuando se termine de usar el pipe es necesario realizar un sys_close con los file descriptors del pipe que se hayan abierto. El pipe en sí, es un buffer circular de tamaño estático que maneja un índice de escritura y lectura, y en base a estos se realiza la lógica de si está vacío o lleno.

Tanto la lectura de un pipe vacío como la escritura en un pipe lleno son bloqueantes, y el proceso será desbloqueado en cuanto esta condición cambie. En particular, para la lectura de un pipe vacío también se desbloqueará el proceso en el caso que no haya ningún otro con el lado de escritura abierto para ese pipe. En ese caso, hace que el proceso lea un EOF y este decida qué hacer según corresponda.

Para modificar los file descriptors del proceso, implementamos el sys_dup2 el cual copia el contenido (file descriptor) de la entrada indicada en el primer argumento en la posición de la tabla de file descriptors que marca el segundo argumento. Debido a las decisiones de diseño para la creación de procesos hijos, implementamos una forma de revertir todo cambio realizado con el dup2 a como estaba al momento de la creación del proceso. De esta manera, es posible crear y abrir un pipe, cambiar la salida/entrada estándar por los file descriptors del pipe, hacer la llamada para la creación de los hijos (de forma que tenga los cambios realizados en el padre en sus file descriptors) y subsecuentemente restaurar los file descriptors del padre a como estaban anteriormente.

Finalmente para manejar el hecho de que pueda haber una variedad de procesos que usen un mismo pipe y saber en qué momento es posible liberar el espacio que ocupa el pipe, es decir destruirlo, tenemos un contador de escritura y lectura que aumenta en el momento que un proceso abre al pipe en escritura y lectura respectivamente. De manera inversa, al momento que un proceso cierre un pipe, se decrementa el valor de apertura en el pipe. Cuando ambos contadores llegan a 0, el kernel procederá a destruir el pipe y liberar el espacio de memoria que ocupa.

Userland

Para la shell, Bizcocho, como decisión de diseño los comandos están separados en dos categorías: los comandos built-in y los no built-in. Los comandos built-in son aquellos que no es posible ni encadenarnos con '|' (Pipe) ni usar '&' (Background), ya que son comandos que realizan cosas simples (no son procesos que se agreguen al scheduler del Kernel) ya sea matar/bloquear a un proceso en base a su PID o imprimir información. Para saber qué comandos son built-in y cuáles no, simplemente se tendrá que correr el comando "help" (muestra los built-in) o "help 2" (muestra los no built-in).

Para el caso de los comandos no built-in, es posible que corran en background incluyendo como último argumento el '&', o si se quiere conectar la salida de un comando con la entrada de otro se usará el ' | ' entre ambos comandos. También se puede combinar ambas opciones por ejemplo, "loop & | filter &", que correrá todo en background. Si un proceso no corre en background, al terminar, mostrará un mensaje con el valor del código con el que terminó.

Cabe destacar que hay comandos que corren indefinidamente, ya sea loop o parte de los tests, para matar a los mismos es necesario apretar el "ESC" el cual matará a todo proceso que esté corriendo en foreground excluyendo al init y a bizcocho (si el proceso que terminamos con ESC tenía hijos estos no se matan). Esto no funciona con procesos en background, si se desea matar estos procesos se deberá buscar su PID con el comando "ps" y luego usando el comando "kill pid". También se puede bloquear con el comando "block pid" o si se quiere aumentar/disminuir la prioridad se usa "nice pid prioridad" (Prioridad debe ser entre 0-4, siendo 0 la mayor prioridad).

A continuación, se mencionan qué es lo que realizan ciertos comandos para el entendimiento de lo que está sucediendo al correrlos. Primeramente tenemos a loop, que cada 5 segundos imprime a la salida estándar un saludo con su PID y la hora actual (por dentro está usando el tiempo del RTC y viendo que pasen mínimo 5 segundos).

Luego tenemos el trío, wc, cat, filter, éstos comandos leen de la entrada estándar y hacen algo distinto sobre este input. En el caso de cat y filter estos imprimen directamente a salida estándar lo que reciben pero filter no incluye las vocales. Por otra parte, wc recibe la entrada estándar pero no imprime nada hasta que se envíe un EOF, entonces imprimirá la cantidad de líneas que leyó (Líneas las tomamos como la cantidad de ENTERs o '\n'). Para enviar un EOF a la entrada estándar, sólo se necesita usar "CTRL + D".

Finalmente, si un proceso que recibe de la entrada estándar se usa como background('&'), tendra la entrada estándar deshabilitada pues la está consumiendo el bizcocho, por ejemplo si hacemos "cat & | filter", cat nunca recibirá nada en la entrada estándar y se bloqueará por ello.

Phylo consiste de un proceso padre el cual comienza todos los filósofos iniciales y procede a quedarse bloqueado en un read, esperando que el usuario toque una 'r' de remove, 'a' de add o 'q' de quit (Tener en cuenta que si phylo se corre en background, no leerá estas teclas). Se utiliza un sistema de semáforos en donde los filósofos de asientos pares buscarán primero el tenedor izquierdo y después el derecho, mientras que los impares lo harán de forma inversa. Para la remoción, el padre espera por el primer y anteúltimo semáforo para poder eliminar el último proceso y semáforo, después se asegura de que no haya inanición por parte de ningún filósofo. Por su parte, el añadir un filósofo únicamente espera por un semáforo, el primero o el último dependiendo de si el tamaño de la mesa es par o impar. La 'q' mata todos los procesos y también cierra todos los semáforos, haciéndolo distinto a terminarlo con un "ESC".

Instrucciones de Compilación y Ejecución

Este proyecto usa Makefile para ser compilado de manera más cómoda. Una vez clonado el repositorio, colocarse en la carpeta bizcocho_os e ingresar "make all" ó "make" en la consola si se desea compilar sin el sistema de manejo de memoria Buddy, si se desea usar este último se debe compilar con "make buddy" . Esto debe hacerse desde la imagen de docker dada por la cátedra. Para la ejecución, simplemente correr el archivo "run.sh"

Pasos para la Correcta Demostración de los Requerimientos

Directamente se puede ingresar los comandos en nuestra terminal, bizcocho. Usar 'help' y 'help 2' para ver un listado y descripción de cada uno.

Para probar los memory managers (compilar con "make buddy" para probar el Buddy Memory Manager), se puede utilizar 'testmm' y 'mem'.

Para comprobar que está funcionando correctamente el sistema de scheduling, se puede correr los programas 'testproc' y 'testprio', además de ps para ver los procesos activos en un momento dado.

Para probar el uso de semáforos, se puede utilizar 'testsync' y 'phylo n', además de 'sem' para ver todos los semáforos activos en un momento dado.

Para probar pipes, se pueden combinar procesos a través de '|' para que se genere un pipe entre ellos. Además, se puede usar 'pipe' para ver los pipes abiertos actualmente y su estado. Para poder llegar a observar estos pipes creados, es necesario correr ambos lados del '|' en background para poder seguir usando la terminal y correr 'pipe'. Por ejemplo, 'loop & | filter &', 'cat | wc', o 'loop & | testmm 1 &' (para ver un ejemplo en el que se escribe en el pipe, pero nadie lo lee).

Puede usarse 'kill ´pid´' para matar un proceso determinado, 'block ´pid´' para bloquearlo o 'nice ´pid´ ´prio´' para cambiar su prioridad.

Limitaciones

Debido a una recomendación de la cátedra, en la única parte del Kernel en donde se asegura que no hay ninguna condición de carrera (mediante la utilización de semáforos o xchg), en el caso que el sistema fuese multi-core, es en semaphores.

Debido a no utilizar la unidad de paginación, el Kernel no puede controlar de ninguna manera que un proceso no pueda pasarse (ya sea en la lectura o escritura) de su espacio de memoria asignado, el cual es fijo y se asigna cuando se crea el proceso. Entonces si se realizan una gran cantidad de llamadas a función como sería en un algoritmo recursivo o en la declaración de un array muy largo, es posible que se pase de su espacio y pise el espacio de otros procesos.

Debido a la naturaleza del Round-Robin scheduling, si se corren muchos procesos con por ejemplo el ‘testproc 300 &’ en background, si bien la terminal sigue funcionando, hay que esperar muchísimo tiempo para que le vuelva a tocar el turno de Bizcocho y que pueda ser usada por un tiempo muy corto.

Por otro lado, como mencionamos anteriormente que matar al padre no mata a los hijos, puede ser incómodo para probar tests como el testproc, pues la única manera de terminarlo es con un ESC, y van a quedar todos los hijos imprimiendo en la pantalla junto con el Bizcocho (habría que matar uno por uno con ‘kill ´pid´’, se recomienda cerrar y volver a abrir el qemu). Si el printeo de cada proceso hijo es molesto, se puede comentar la línea en la que cada uno printea su PID (testing_utils.c:89) y recompilar el TP.

Problemas durante el Desarrollo

Además con los problemas obvios que suelen surgir en la implementación del Buddy Management System y de algún programa como Phylo, la mayor dificultad apareció en el diseño de pipes, que para resolverlo tuvimos que implementar un sistema de file descriptors. También estuvimos prácticamente un día entero debuggeando un error generado porque como nuestro scheduler buscaba pids de manera recursiva, los stack frames terminaban pisando información importante si uno generaba numerosos procesos. Encontrar que ese era el problema fue complicado, pero gracias al watch de gdb lo pudimos hacer, lo arreglamos haciendo que haga la búsqueda de manera iterativa.

Fuentes de Código de Terceros

La amplia mayoría del código fue brindado por la cátedra o hecho por nosotros. Sin embargo, la implementación del printf y funciones de strings como strncpy fueron obtenidas de:
https://stackoverflow.com/questions/1735236/how-to-write-my-own-printf-in-c
https://codebrowser.dev/linux/linux/lib/string.c.html y modificadas a nuestra conveniencia.

Este paper del “Iraqi Journal of Science” describe una implementación válida del Buddy system, la cual no utilizamos, pero nos dió la idea de guardar la información de las listas en los mismos espacios de memoria disponibles. https://www.iasj.net/iasj/download/b4f148f6b64183c9

Modificaciones a los Tests Brindados

Además de los cambios de nombres de funciones realizados para la correcta adaptación a nuestro sistema operativo, el único cambio notable a simple vista fue el hacer que algunos tests imprimieran información relevante, para controlar de mejor manera qué es lo que está sucediendo en un momento determinado, estas impresiones no cambian el funcionamiento de los tests. Es importante notar que en testsync, vale la pena modificar un poco la cantidad de procesos y el largo del “for” (busy waiting) antes de compilarlo, ya que sobre todo cuando es corrido con semáforos, puede tardar demasiado tiempo si los dos valores antes mencionados son demasiado altos.

Miscellaneous

Se hizo uso de los analizadores de código estático, PVS-Studio y CPPCheck, se omitieron los errores y warning que provenían de la zona de código que no pertenece a nuestra autoría. Especialmente la zona del Bootloader, con el archivo BMFS.c, y el NaiveConsole.c de Kernel.

Por otro lado, se pueden notar mensajes de estilo/portabilidad del CPPCheck y notas del PVS-Studio en archivos de nuestra autoría. En el caso del CPPCheck decidimos ignorarlos tras considerar que cambiaban cosas muy pequeñas del código, como serían reducir el scope de una variable. Para el caso de PVS-Studio, la mayor parte de las notas eran por el uso de las funciones printf, snprintf, fprintf y demás, las cuales nos marcan posibles problemas con el tipo del argumento que se le está pasando a un formato dado, pero debido a que nuestra implementación de estas funciones están hechas de tal forma que tienen en cuenta estos tipos de dato, no lo cambiamos. También, existen notas que aluden a posibles de-referenciaciones de un puntero NULL, pero en estos casos siempre había un chequeo de NULL más arriba en el código, o el único lugar donde se usa dicho puntero siempre será distinto de NULL.

Conclusión

Podemos concluir que este trabajo práctico ha sido de suma importancia para el correcto entendimiento de cómo funcionan los sistemas operativos, además de la correcta comprensión de cosas que antes dábamos por sentado como la utilización de semáforos para evitar condiciones de carrera, scheduling de procesos, manejo de pipes, entre otras cosas.