Sistemas operativos - 2C 2022

Trabajo Práctico 2

Fecha de entrega: 7/11/2022

Integrantes:

Federico Shih (62293)

Máximo Rojas (62353)

Agustin Zakalik (62068)

Índice

Abstract

Este trabajo práctico consiste en utilizar uno anterior, de la materia Arquitectura de las Computadoras, implementando importantes mejoras. Dentro de estas mejoras se encuentran la creación de un sistema de scheduling para poder ejecutar distintos procesos concurrentemente, dos sistemas distintos de manejo de memoria, y herramientas de sincronización y comunicación entre procesos.

Instrucciones de compilación y ejecución

Al descargar el repositorio, se necesita la imagen de docker provista por la cátedra para garantizar el correcto funcionamiento del trabajo práctico. Una vez dentro de la imagen, en el directorio del proyecto, moverse a la carpeta toolchain y ejecutar los comandos:

1) make clean
2) make all

Luego se debe regresar a la carpeta root para terminar de compilar el proyecto. Antes de hacer este último paso, se debe decidir con qué memory manager se quiere compilar. Las posibles opciones son un free list memory manager y un buddy memory manager.

A continuación, se debe correr nuevamente el comando ‘make clean’, seguido del comando ‘make all MM=XXX’, donde XXX puede ser STANDARD o BUDDY, dependiendo del memory manager elegido anteriormente.

Otra opción de compilación es utilizar el bash script ./compile.sh. Este tiene 4 comandos importantes:

1. ./compile.sh docker <nombre-container>
   Sirve para crear el docker container. Si ya existe omitir este paso.
2. ./compile.sh buddy <nombre-container>
   Sirve para compilar con buddy cómo MM. Pasar el nombre del container como segundo parámetro.
3. ./compile.sh standard <nombre-container>
   Compila con el memory manager estándar.
4. ./compile.sh clean <nombre-container>
   Limpia los objetos de compilación

Finalmente, para ejecutar el proyecto, correr el comando desde un terminal de wsl o linux el comando `./run.sh` (si se intenta ejecutar desde la imagen de docker, el comando dará error).

Decisiones Tomadas

Physical Memory Manager:

Para el trabajo práctico, se implementaron dos administradores de memoria física distintos. El denominado “Standard” está basado en la implementación del libro “The C Programming Language” (ver bibliografía) que utiliza una lista circular de bloques de memoria libres. Por el otro lado, el segundo es una implementación de un sistema buddy, que está basado en la implementación por el usuario de github evanw (ver bibliografía). Para verificar las dos implementaciones, utilizar el comando `test_mm x` (con x siendo un número entero mayor a cero). Tener en cuenta que el tamaño máximo de la memoria es de 0x100000 al elegir el valor x.

El comando anterior ejecuta el test de memoria provisto por la cátedra. También se puede ejecutar el comando mem (que imprime el estado de la memoria) antes y después de correr cualquier otro comando para comprobar que no hay memory leaks.

Free List:

El memory manager ‘Standard’ como lo definimos en la compilación, fue implementado utilizando una free list, donde cada nodo apunta con el uso de un puntero al siguiente nodo y el tamaño de bloques libres que contiene el mismo.

Inicialmente la lista comienza con un único nodo con tamaño de toda la memoria disponible. Cuando se intenta reservar memoria a través del llamado a malloc, este se fija si existe un bloque libre del mismo o mayor tamaño al solicitado. Si es de mayor tamaño se fragmenta (manteniendo la alineación de memoria) hasta conseguir un nuevo bloque que sea de tamaño mínimo para ofrecer y guardar la memoria solicitada. Si se encuentra un bloque de tamaño justo, se ofrece ese sin fragmentar. Es importante destacar que a medida que se fragmentan los nodos, se van generando nuevos nodos libres para usar.

Finalmente, cuando se quiere liberar un nodo, la función de free se encarga de mantener la fragmentación al mínimo por lo que cuando se puede unificar nodos adyacentes libres, lo hace.

A continuación se puede ver la declaración de un nodo:

```
typedef long Align;

typedef union header
{
  struct
  {
    union header *next;
    unsigned int size;
  } data;
  Align x;
} Header;
```

Buddy System:

El buddy system fue implementado con un array de nodos de una lista circular, donde cada índice del mismo representa un orden de la potencia de 2, siendo la posición 0 del array la potencia más alta y la última posición la más baja.

Cuando se pide memoria a través de la system call malloc, primero se busca un orden o posicion ideal en el cual guardar la memoria necesitada (donde 2^i >= memoria requerida). Luego en ese mismo orden, se busca un bloque libre menor tal que este sea al menos de la cantidad de bytes solicitados. Si no se encuentra, los bloques comienzan a fragmentarse hasta llegar al tamaño mínimo requerido. Una vez presente el bloque ideal para guardar la memoria necesitada, se provee.

Cuando se necesita liberar memoria a través de la syscall de free, se recorre el mismo proceso pero en sentido contrario. Una vez que se libera la memoria, el bloque que queda liberado intenta reducir la fragmentación y comienza a unirse con bloques para así intentar llegar a una cantidad mínima de bloques (idealmente 1 bloque). De esta manera se reduce la fragmentación de memoria

Comandos en shell:

- mem: imprime estado del memory manager. Imprime memoria total, libre y ocupada.
- test_mm <# de memoria>: corre el test de memory manager de la catedra.

Process Manager:

El process manager fue implementado con una cola de procesos. Cada proceso guarda la siguiente estructura:

```
typedef struct pcb
{
 int pid;
 int ppid;
 int foreground;
 process_state state;
 priority_type priority;
 char name[30];
 int fileDescriptors[2];
 void *rsp;
 void *rbp;
 int argc;
 char **argv;
 int semId;
} pcb;
```

- **pid** representa el id del proceso
- **ppid** representa el id de su proceso padre
- **foreground** es una variable booleana que representa si se corre en foreground o no.
- **state** representa el estado en el cual se encuentra el proceso. Es un enum de 4 valores, READY, BLOCKED, EXITED, TERMINATED.
- **priority** representa la prioridad, que en nuestro sistema se refleja en la cantidad de ciclos de ejecución puede tomar el proceso. Cada vez que sucede una interrupción por timer tick, se incrementa en uno a una variable que cuenta la cantidad de veces que se corrió el proceso. Cuando llegue al límite, finalmente cederá la CPU al siguiente proceso en la cola. La máxima prioridad es 20 y la mínima prioridad es 1.
- **fileDescriptors** representa el stdin y stdout a la cual imprimirá cada proceso. Por defecto se heredan los filedescriptors de su padre, y si no tiene padre sus valores comenzarán con STDIN_PIPENO y STDOUT_PIPENO.
- **rsp y rbp** representan los punteros al stack frame. Cada vez que se haga context switching, se pushean los registros al stack y se actualiza el rsp del proceso corresondiente.
- **argc y argv** son los argumentos con los que inicializa el proceso
- **semId** se utiliza para avisar a que este proceso ha llegado a su EXIT.

Al inicializar el proceso, se inicializa el PCB y también se aloca su stackframe. Como limitación, definimos un STACK_SIZE de 4096 bytes para el stackframe de cada proceso. En el stackframe se arma el estado inicial con el cual se inicializa el proceso. También hicimos un proceso wrapper, que se encarga de llamar la función del proceso. Cuando el proceso finalmente termine, processWrapper se encarga de llamar exit.

Al inicializar el scheduler, se inicializa la queue de procesos y 2 procesos iniciales. El proceso idle, que reemplaza cuando no hay procesos READY para ejecutar, y el proceso Init, encargado de recibir todos los procesos sin padre o no waiteados por el Shell. Todo proceso background inicializado por Shell es manejado por Init, así que cuando termina tal proceso background, Init se encarga de limpiarlo. Init limpia los procesos cada 10 segundos, definido como FLUSH_TIME. Cada vez que un proceso es matado, si este sigue teniendo hijos les pasa los hijos al Init. Si el propio proceso es huérfano, se le asigna como padre al proceso init.

Tanto blockTask, killTask y resumeTask se basan en la función changeState. Este se encarga de cambiar el estado, aumentar o decrementar la cantidad de procesos en READY, y también en caso de que el nuevo estado sea EXITED, avisar al semáforo del proceso que ya puede ser recolectado.

Cuando el propio proceso se libera de memoria, este se encarga también de avisar que el proceso haya terminado (en caso que no pase por la etapa de EXITED) y también de cerrar los semáforos y pipes. Si tiene como stdout un pipe que no es stdout, envía un EOF al cerrar el pipe.

La existencia de TERMINATED y EXITED surge ya que un proceso cuando termina no se libera de memoria, sino que tiene que ser liberado por el proceso padre mediante waitpid. Este es el estado de EXITED. Cuando finalmente es liberado por el padre, este proceso pasa a ser TERMINATED, y cuando sea su turno en cambio de contexto este es liberado del queue y se libera su pcb.

Los procesos también pueden ser liberados forzosamente mediante una interrupción de teclado. Con CTRL+C, el sistema busca cual es el proceso corriendo en foreground actualmente, y lo fuerza a pasar a EXITED. Esto no se aplica para los procesos de Shell y el Init.

A la hora de hacer un cambio de contexto, los registros son pusheados al stack. Por este motivo, se guarda el valor del rsp cada vez que haga el pasaje de un proceso a otro. La funcion scheduleTask es la encargada de reasignar el rsp para que el proceso a ejecutar recupere sus registros al ejecutar el comando pop. Esto también asigna el valor correcto al registro RIP.

Comandos en shell:

- ps: lista información sobre los procesos
- kill <pid>: mata el proceso.
- block <pid>: bloquea el proceso
- resume <pid>: resume el proceso
- nice <pid> <priority>: cambia la prioridad del proceso
- test_prio: prueba el sistema de prioridades del scheduler
- test_processes <n>: inicializa n cantidad de procesos, que son bloqueados y luego terminados para probar que funcione el sistema de scheduling entre otras cosas.
- loop: proceso que imprime regularmente su pid

Sincronización de procesos

El sistema de sincronización de procesos almacena una cantidad variable de semáforos en una lista. Para asegurarnos de que cada llamada a un semáforo es atómica, implementamos un lock general (que habilita o deshabilita cualquier tipo de operación sobre semáforos en un instante dado) utilizando acquire y release del lock mediante la función de xchg y locks individuales de cada semáforo. Esta es la estructura de un semaforo:

```
typedef struct t_sem
{
 queueADT blockedPidsQueue;
 uint16_t attachedProcesses; // para init y destroy automatico
 int id;
 int value;
 int lock;
 char *name;
} t_sem;
```

Debido a que necesitábamos utilizar una gran cantidad de semáforos en toda la aplicación, y no todos requerían tener un nombre identificador, decidimos hacer tanto semáforos nombrados como semáforos no nombrados. La variable name almacena el nombre de un semáforo nombrado, o tiene el valor “” si el semáforo es no-nombrado.

El lock se utiliza para asegurar la atomicidad (conforme a lo escrito anteriormente), y el value es el valor actual del semáforo. BlockedPidsQueue se encarga de almacenar los pids de los procesos bloqueados por un semáforo específico. Estos procesos se irán desbloqueando en un orden de cola luego de llamadas a semPost.

Finalmente, attachedProcesses almacena la cantidad de procesos que tiene abierto el semáforo, lo que permite lograr que semClose solo cierre un semáforo cuando no haya más de un proceso que esté utilizándolo en ese instante.

Para abrir o crear un semáforo nombrado, se utiliza semOpen. Si no existe el semáforo, se crea, y si existe, se suma uno a la variable attachedProcesses. Para estos semáforos, cada vez que se cierra, se resta uno a attachedProcesses y cuando no queden más procesos con el semáforo abierto (attachedProcesses = 0) se destruye el semáforo.

Para crear semáforos sin nombre, se utiliza la función semInit.

Nuestro sistema provee las funciones semPost, semWait y sem en userland para interactuar con los semáforos. Cuando un proceso hace semWait, se fija si puede adquirir el semáforo. Si no puede, se agrega el proceso a la cola de blockedPids y se bloquea a sí mismo. Cuando otro proceso haga un post, se fija primero si hay procesos bloqueados. Si hay, en vez de incrementar el valor del

semáforo, se desbloquea el primer proceso bloqueado por una llamada a semWait.

Para demostrar el funcionamiento de las herramientas de sincronización de procesos que provee nuestro SO, se puede correr el test llamado test_sync desde la consola del mismo.

El funcionamiento de la función test_sync consiste en crear una cantidad de procesos predefinida por la macro TOTAL_PAIR_PROCESSES en el archivo test_sync.c. Durante la creación de dichos procesos, se les asigna en sys_startTask una función a correr, llamada my_process_inc. La misma decide mediante los argumentos provistos por el usuario cuantas veces los procesos creados anteriormente deben ejecutar las instrucciones de suma y resta en una variable global, así como si se desea usar o no semáforos.

El primer argumento recibido por test_sync indica la cantidad de sumas y restas a ejecutar por cada proceso, y el segundo argumento indica si se quiere utilizar semáforos o no.

Dos ejemplos de uso son:

1) “test_sync 4 1”: una vez finalizada la función, se puede ver que la variable “global” es igual a cero, ya que se hizo un buen uso del sistema de semáforos. Notar que reemplazando el número 4 por cualquier número >= 1 el resultado será siempre el mismo.
2) “test_sync 6 0”: en el segundo argumento se indicó que no se deben usar semáforos. Esto provocará que el resultado final de la variable “global” sea indefinido. Notar que si se reemplaza el número 6 por cualquier número >= 1 se observa el mismo comportamiento.

Además de lo anterior, el SO provee la función “sem”, que al ejecutarse en la consola imprime en pantalla todos los semáforos abiertos en ese instante. Al ejecutar la función sem antes y después de cualquier otra función (incluyendo test_sync) se puede notar que el número de semáforos abiertos es siempre el mismo. Esto indica que se hizo un cierre correcto de los semáforos que ya no están en uso.

Comandos en shell:

- test_sync <# de inc> <sync o no>: test de la cátedra.
- sem: imprime el estado de los semáforos
- phylo: corre filósofos

Comunicación inter proceso

Para la comunicación inter-proceso, se implementó un sistema de pipes identificados con un id numérico. Se crearon las funciones pipeWrite y pipeRead, que permiten escribir y leer de un pipe determinado. Cada pipe posee un buffer de tamaño PIPE_BUFFER_SIZE.

A continuación se encuentra la declaración de un pipe:

```
typedef struct t_pipe {
 int id;
 char buffer[PIPE_BUFFER_SIZE];
 int writeIndex, readIndex;
 int totalProcesses;
 int writeSemId, readSemId;
} t_pipe;
```

En el struct se pueden ver 2 índices y 2 semáforos. Esto es debido a que ambas operaciones pueden ser bloqueantes, y se necesita una forma de avisar cuando es posible escribir o leer. WriteSemId inicializa con valor PIPE_BUFFER_SIZE y readSemID con valor 0. Cada vez que se agrega un carácter, se hace wait en writeSemId y se hace un post a readSemId. A su vez, cada vez que se lee un carácter, se hace un un wait en readSemId y un post a writeSemId. De esta forma, nos aseguramos de que no haya una espera activa para escribir y leer al pipe.

Para abrir un pipe, se debe proveer un identificador numérico que llamamos pipeId.Si ya existe un pipe con tal pipeId, se incrementa totalProcesses en uno. Sino, se crea el pipe. Esto sirve para saber cuando o no destruir el pipe, de la misma forma que con el sistema de semáforos.

El sistema de pipes se utilizó en el kernel para crear el STDIN. Cuando se reciben interrupciones de teclado, se terminan escribiendo los caracteres recibidos al pipe STDIN_PIPENO. Es por este motivo que la función getChar funciona haciendo un readPipe a STDIN_PIPENO.

Demostración del funcionamiento de cada uno de los requerimientos

Para ejecutar un comando una vez inicializado el programa, simplemente escribir el comando en la terminal y presionar la tecla enter. Se recomienda utilizar el comando `help` una vez abierto el programa para ver los comandos disponibles al usuario a correr. La tabla de help contiene 3 páginas, y se puede acceder a cada una de ellas pasando como argumento el número de página que se desea ver. El comando `ejemplos` imprime en la terminal 2 ejemplos, mostrando cómo correr comandos con el operador | (pipe) y como correr procesos en background (utilizando el símbolo & como último parámetro).

Comandos en shell:

- pipes: imprime estado de pipes
- <comando1> | <comando2>: realiza la operación pipe.

Problema de los filósofos comensales

Debido a que el ejercicio de los filósofos comensales requiere que se puedan agregar y eliminar filósofos, se decidió que los filósofos sean capaces de "comunicarse" entre sí, permitiendo que estos puedan averiguar si son capaces de comer (chequeando el estado de los filósofos a sus costados) y bloquearse. Cuando los filósofos de sus costados terminen de comer, le avisan para que empiece a comer. Hay 3 estados, DECIDING, WAITING, EATING, cada uno representando en qué estado está el filósofo.

Esta es la estructura que guarda un filósofo. El conjunto de filósofos se guarda en un array de tamaño fijo de 15 filósofos.

```
typedef struct philosofer_t
{
 int pid;
 philo_state state;
 int semId;
} philosofer_t;
```

El semáforo sirve para que el filósofo no tenga que estar con espera activa a la hora de fijarse si comer o no. Hay un mutex global llamado gateMutex que se encarga de asegurar que no se pueda acceder al valor de currentPhilosofers (la cantidad de filósofos presentes)desde más de un proceso al mismo tiempo. Esto es necesario para tanto todas las operaciones de philo, la impresión del estado (printTable) y para el agregado y el eliminado de filósofos. De esta forma, nos aseguramos de que no exista acceso mutuo a la variable crítica de currentPhilosofers y el array de filósofos.

El funcionamiento se basa en que los filósofos van a intentar comer en todo momento. Cuando se sientan, se fijan si los que están a sus costados están comiendo con la función testEat. Si no es capaz de comer, se bloquea esperando su propio semáforo o en el caso inicial, realiza la espera de vuelta. Aquellos filósofos que sí pudieron comer van a terminar y van a hacer testEat a sus compañeros, avisándoles que pueden comer y que pueden proceder.

Al mismo tiempo, hay un proceso aparte encargado de hacer la impresión de la mesa. Este se llama printTable, y lo único que hace es imprimir "E" si está comiendo y " . " si está en otro estado.

A la hora de agregar y eliminar filósofos, se hace wait a gateMutex para modificar currentPhilosofers y se crean los semáforos del filósofo y su proceso. A la hora de eliminar, simplemente cierran su semáforo y eliminan el proceso.

Comandos en shell:

- phylo: corre el programa

Comandos notables de shell:

- cat: imprime del stdin
- wc: cuenta la cantidad de líneas
- filter: imprime con los vocales filtrados
- help: ayuda con los comandos

Limitaciones

En primer lugar, ambos sistemas de manejo de memoria están limitados a una cantidad de memoria máxima fija.
Por otra parte, no todas las partes del sistema operativo están pensadas para ser escalables a multicore. Aunque se intentó que todo funcionase al ejecutarse en un sistema con múltiples cores, en algunos casos específicos se complejizaba demasiado, por lo que consideramos que escapaba la consigna intentar solucionarlos.
Es por este motivo que algunas system calls hacen uso de semáforos para evitar posibles condiciones de carrera ante un cambio de contexto, a pesar de que esto no ocurrirá nunca en la utilización de QEMU dado que el mismo emula un sistema con un solo core y las system calls fueron implementadas de tal manera que no sean desalojables. Cabe aclarar que todo el sistema de semáforos si es adaptable a un sistema con cualquier número de cores.

Debido a la falta de paginación, se tuvo que limitar la cantidad de memoria de stack de cada proceso a solo 4096 bytes de memoria. Esto genera un límite de cuantos stackframes y variables que se puede almacenar en un proceso.

El ejercicio de filósofos se limitó a 5 filósofos mínimo y 15 filósofos máximo. Se podría haber utilizado una lista o un mapa pero debido a performance y simplicidad se decidió dejarlo en un array de estructuras.

Problemas encontrados durante el desarrollo y su solución

El proceso de creación de scheduling fue extremadamente complejo. Tuvimos problemas con bugs sobre la estructura de colas que habíamos hecho y algunas mutaciones que causan problemas. Sin embargo, el mayor problema sucedió cuando tuvimos que implementar el waitpid. Necesitábamos una forma de que los procesos padres puedan saber cuando terminaron los hijos, por ejemplo el Shell, que tiene que saber si seguir recibiendo comandos o no. Vimos

la oportunidad de utilizar semáforos, agregando a cada proceso un semáforo que representa si su proceso cambió de un estado corriendo (como READY o BLOCKED) a uno eliminado (EXITED, TERMINATED). Entonces, cuando el padre hace waitpid, va a buscar el hijo en cuestión y espera al semáforo de su hijo. Cuando el hijo termine, va a estar en EXITED, y finalmente el padre es el que se encargará de liberar a su hijo pasándolo a TERMINATED. Para resolver esto, tuvimos que primero implementar un método de bloqueo de procesos, implementar los semáforos y finalmente terminar el scheduler.

También tuvimos problemas implementando el problema de los filósofos comensales. Al principio funcionaba utilizando una solución con n-1 filósofos sentados, pero a la hora de agregar o eliminar filósofos tuvimos problemas ya que los procesos esperaban semáforos que no sí o sí existían. Para resolverlo, tuvimos que implementar otra solución, haciendo que los filósofos mismos se comuniquen entre sí, fijándose si pueden comer o no y avisando a los filósofos adyacentes cuando ya terminaron de comer.

Reporte de PVS-Studio ignorado:

Se ignoraron todos los comentarios del archivo bmfs.c

/root/Kernel/drivers/console.c 10 note V566 The integer constant is converted to pointer. Possibly an error or a bad coding style: (uint8_t *) 0xB8000
/root/Kernel/drivers/console.c 16 warn V1009 Check the array initialization. Only the first element is initialized explicitly. The rest elements are initialized with zeros.

/root/Kernel/kernel.c 29 note V566 The integer constant is converted to pointer. Possibly an error or a bad coding style: (void *) 0x400000
/root/Kernel/kernel.c 30 note V566 The integer constant is converted to pointer. Possibly an error or a bad coding style: (void *) 0x500000
/root/Kernel/kernel.c 31 note V566 The integer constant is converted to pointer. Possibly an error or a bad coding style: (void *) 0x600000

/root/Kernel/naiveConsole.c 8 warn V1009 Check the array initialization. Only the first element is initialized explicitly. The rest elements are initialized with zeros.
/root/Kernel/naiveConsole.c 9 note V566 The integer constant is converted to pointer. Possibly an error or a bad coding style: (uint8_t *) 0xB8000
/root/Kernel/naiveConsole.c 10 note V566 The integer constant is converted to pointer. Possibly an error or a bad coding style: (uint8_t *) 0xB8000

/root/Userland/SampleCodeModule/lib/fibonacci.c 15 note V776 Potentially infinite loop.

/root/Userland/SampleCodeModule/lib/phylo.c 59 warn V1009 Check the array initialization. Only the first element is initialized explicitly. The rest elements are initialized with zeros.

(warn v1009) El índice del array que no está inicializado se inicializa una línea después. No creemos necesario tener que inicializar el array completo para no tener este warning.

/root/Userland/SampleCodeModule/include/usersyscalls.h 11 warn V1071 Consider inspecting the '_syscall' function. The return value is not always used. Total calls: 31, discarded results: 3.

/root/Userland/SampleCodeModule/lib/ustdlib.c 251 warn V560 A part of conditional expression is always true: * str >= 'A'.

Falsos positivos de cppcheck marcados en amarillo

[Bootloader/BMFS/bmfs.c:689]: (warning) %lld in format string (no. 1) requires 'long long' but the argument type is 'unsigned long long'.
[Bootloader/BMFS/bmfs.c:262] -> [Bootloader/BMFS/bmfs.c:264]: (performance) Buffer 'DiskInfo' is being written before its old content has been used.
[Bootloader/BMFS/bmfs.c:276]: (style) The scope of the variable 'writeSize' can be reduced.
[Bootloader/BMFS/bmfs.c:451]: (style) The scope of the variable 'percent' can be reduced.
[Bootloader/BMFS/bmfs.c:725]: (style) The scope of the variable 'tfile' can be reduced.
[Kernel/kernel.c:49]: (style) The function 'initializeKernelBinary' is never used.
[Kernel/drivers/console.c:135]: (style) The function 'initializeSingleScreen' is never used.
[Kernel/interruptions/irqDispatcher.c:11]: (style) The function 'irqDispatcher' is never used.
[Kernel/naiveConsole.c:60]: (style) The function 'ncPrintBin' is never used.
[Kernel/lib/queue.c:93]: (style) The function 'peek' is never used.
[Kernel/drivers/console.c:94]: (style) The function 'printBase' is never used.[Kernel/drivers/console.c:110]: (style) The function 'printColor' is never used.
[Kernel/scheduler/processManager.c:117]: (style) The function 'scheduleTask' is never used.
[Userland/SampleCodeModule/lib/usersyscalls.c:41]: (style) The function 'showCursor' is never used.
[Kernel/lib/string.c:28]: (style) The function 'strncpy' is never used.

[Kernel/interruptions/syscalls.c:49]: (style) The function 'syscallHandler' is never used.

Los avisos a archivos propios de barebones como bmfs fueron ignorados. Por otro lado, funciones como peek o strncpy, aunque el cppcheck de warnings, quisimos dejarlas pues son parte de una librería de funciones y podrían ser útiles en algún futuro.

Bibliografía

Dennis Ritchie, Brian Kernighan. The C Programming Language (free list MM, sección 8)
https://github.com/evanw/buddy-malloc (implementación de evanw para el buddy system)
https://stackoverflow.com/questions/1735236/how-to-write-my-own-printf-in-c para parte de la implementación de _fprintf en userland

Modificaciones a las aplicaciones de test provistas por la cátedra

Todos los tests de la cátedra tuvieron que cambiar de argumentos debido a que determinamos que el primer argumento que reciben los programas son el nombre de su proceso.

Además de eso, cambiamos test_process para poder liberar los procesos que iban siendo matados, ya que kill no los termina, sino que los fuerza a hacer exit (leer process manager). También pusimos sleep en el test_process para apreciar el funcionamiento del test_process.