SO - 2do cuatrimestre 2022

Trabajo Práctico

Construcción del Núcleo de un Sistema Operativo y estructuras de administración de recursos.

Grupo 2

Gastón Alasia, galasia@itba.edu.ar, 61413
Matías Della Torre, mdella@itba.edu.ar, 61016
Patricio Escudeiro, pescudeiro@itba.edu.ar, 61156

7 de noviembre de 2022

Introducción

A lo largo de este informe, se hace referencia a las decisiones tomadas durante el desarrollo: Memory Managers, scheduler, procesos, IPC, syscalls, Kernel y Userland, entre otros. Luego, se explica cómo instalar y ejecutar el proyecto, así como los lineamientos de su uso correcto. Esto incluye los programas y las herramientas disponibles con sus respectivas explicaciones, como pipes, procesos en background, cierre de procesos, etc. Por otra parte, se detallan claramente los problemas sucedidos durante el desarrollo y sus soluciones. Finalmente, se explican los cambios realizados a los tests de la cátedra y las fuentes externas de código.

Decisiones tomadas durante el desarrollo

Durante el desarrollo del proyecto, se tomaron ciertas decisiones que influyen tanto en la manera de escribir el código cómo en la interacción del usuario con el Sistema Operativo.

Memory Manager

Implementación tomada del heap 2 de FreeRTOS. Se le asignó un tamaño de memoria máximo de 128 MB. La única diferencia que se puede apreciar entre la implementación tomada y la nuestra, es que nosotros decidimos pasarlo a un ADT para luego facilitar el traspaso en este memory manager y el de buddy.

Buddy Memory Manager

Implementación basada en el heap2 de FreeRTOS y con ayuda del libro “The Art of Computer Programming Vol 1” de Donald Knuth y al igual que el anterior memory manager, se le otorga un tamaño máximo de 128 MB. Al igual que en el memory manager anterior se decidió pasar la implementación de un ADT y se modificó levemente la estructura del nodo de la lista, y la manera de asignar memoria. Al ser un buddy se le agregó una manera de volver a unir los bloques de memoria.

Scheduler

El scheduler implementado hace uso del algoritmo Round Robin basado en prioridades, con un sistema de envejecimiento para evitar la inanición de ciertos procesos.
Ofrece funcionalidades para crear procesos, matarlos, cambiarles las prioridades a demanda y hacer que un proceso abandone la CPU.
Además, soporta infinitos procesos corriendo simultáneamente y 10 niveles de prioridades para los mismos, siendo 1 la prioridad máxima y 10 la mínima. En prioridad 1 el proceso corre en foreground, mientras que con prioridades de 2 a 10 corre en background.
Cabe destacar que al momento de la creación de procesos, se decidió tomar como tamaño de su heap 4KB (4096 bytes) siguiendo las sugerencias de la cátedra.

Semáforos

Estos se basan en los semáforos de POSIX, replicando completamente su funcionalidad. Existen syscalls para abrirlos (open), cerrarlos (close), incrementar su valor (post) y para que un proceso se quede colgado del mismo (wait). Se estableció un límite de 15 semáforos activos en el sistema para evitar posibles problemas de memoria que pudieran ocasionarse. En caso de que más de un proceso se encuentre esperando que un semáforo se libere, existe una lista donde se guardan los PIDs de estos procesos. El límite de procesos esperando también es de 15.
Al salir de un proceso, su PID se borra de las listas de espera de los semáforos que lo tienen bloqueado, pero se tomó la decisión de que dichos semáforos sigan estando abiertos.

Pipes

Los pipes están fuertemente basados en los pipes de POSIX, que hacen uso de un buffer circular de 512 bytes para almacenar la información que luego podrá ser leída. Se ofrece una syscall para crearlos, recibiendo los file descriptors a conectar internamente (funcionamiento similar al de Linux, pero los FDs no se encuentran en un array). Además, también se ofrecen syscalls abrirlos, escribir, leer e incluso cerrar file descriptors. Cada pipe tiene un ID que lo identifica en Kernel.
Cabe destacar que los pipes implementados soportan un solo lector y un solo escritor simultáneamente, por lo que tienen una lista de espera al igual que los semáforos pero esta es de solo un proceso.
Por último, es importante mencionar que al hacer exit de un proceso no se cierran los pipes a menos que sean pipes que dicho proceso esté usando como stdin o stdout), y se elimina su PID de la lista de espera.

Lectura y escritura en background

La escritura por salida estándar la puedan hacer los procesos tanto en foreground como en background, lo que permite que al correr algunos tests en segundo plano se pueda ver en tiempo real qué están haciendo mediante prints en pantalla.

Con respecto a la lectura o input de usuario, se impidió que los procesos en background puedan leer de entrada estándar, mientras que los que están en foreground sí pueden hacerlo.

Comunicación Kernel y Userland

Como se realiza en un Sistema Operativo, para garantizar la correcta separación entre el Kernel y Userland, todas las comunicaciones entre estos dos sectores se realizan mediante una serie de system calls.

La convención de llamadas a las mismas es la de ABI 64 bits: se pasan los parámetros en los registros rdi, rsi, rdx, rcx, r8 y r9 y el valor de retorno queda en el registro rax.

Lista de syscalls:

%rax	syscall	%rdi	%rsi	%rdx	%rcx	%r8	%r9	Descripción
1	sysRead	int fd	char *buf	size_t count				Lee la entrada del teclado. Recibe un file descriptor, un buffer donde dejar el texto leído y el máximo de caracteres a leer.
2	sysWrite	int fd	const char *buf	size_t count				Escribe en pantalla. Recibe file descriptor, un buffer con el string y la longitud del mismo. Devuelve la cantidad de caracteres que escribió.
3	sysClear							Borra todo el contenido de la pantalla.
5	sysInforeg							Imprime los valores de todos los registros guardados
6	sysDate	char value						Recibe como parámetro un valor del 1 al 7 y devuelve el dato correspondiente del módulo RTC.
7	sysSleep	int ms						Pausa el procesador. Recibe como parámetro el tiempo en milisegundos.
9	sysSnapshotRegs							Carga los registros secondary a los registros principales.
10	sysMalloc	unsigned int bytes						Reserva memoria dinámica. Recibe como parámetro la cantidad de bytes a reservar y devuelve el puntero a dicha memoria.
11	sysFree	void *memTo						Libera memoria dinámica. Recibe como

		Free						parámetro un puntero a la memoria a liberar.
12	sysMemStatus	unsigned int *status						Devuelve el estado de la memoria en las primeras 3 posiciones del array que se le pasa como parámetro: tamaño del heap, memoria disponible y memoria ocupada.
13	sysCreateProcess	uint64_t ip	uint8_t priority	uint64_t argc	char *argv	fd*customStdin	fd *customStdout	Crea un nuevo proceso. Recibe como parámetros ip, prioridad (1 a 10), cantidad de argumentos, array de argumentos, pipe de stdin y pipe de stdout. Devuelve 1 si pudo crear el proceso, 0 sino.
14	sysExit							Finaliza el proceso actual.
15	sysGetpid							Devuelve el PID del proceso actual
16	sysPs	char * buffer						Recibe un buffer con una cantidad de memoria suficiente para almacenar la lista de procesos corriendo actualmente.
17	sysKill	uint32_t pid						Mata el proceso con el pid pasado como argumento.
18	sysChangePriority	uint32_t pid	uint8_t priority					Recibe como parámetro un PID y una prioridad (1 a 10) que se le aplica a dicho proceso.
19	sysChangeState	uint32_t pid						Cambia el estado de un proceso entre bloqueado y listo. Recibe como parámetro el pid del proceso, y devuelve 1 si pudo cambiarlo y 0 sino.
20	sysYield							Renuncia la CPU del proceso actual.

21	sysSemOpen	uint32_t id	int value					Abre un semáforo. Le asigna el id y el valor pasados como parámetros, y devuelve una estructura Semaphore.
22	sysSemClose	Semaphore * sem						Cierra el semáforo que se le pasa como parámetro.
23	sysSemPost	Semaphore * sem						Incrementa el valor del semáforo que se le pasa como parámetro.
24	sysSemWait	Semaphore * sem						Hace un wait sobre el semáforo que se le pasa como parámetro. Luego de la espera, decrementa el valor del mismo.
25	sysGetAllSems	char * buffer						Recibe un buffer con espacio suficiente para almacenar la lista de semáforos actuales.
26	sysCreatePipe	fd *fd1	fd *fd2					Crea un pipe con el extremo de lectura fd1 y el extremo de escritura fd2.
27	sysGetAllPipes	char * buf						Recibe un buffer con el tamaño suficiente y guarda en él la lista con el estado de todos los pipes del sistema.
28	sysOpenPipe	fd *user	uint32_t id	uint8_t permisions				Crea un pipe en caso de que no esté creado o se une en caso de que ya exista. Recibe un id y los permisos (0: escritura, 1 lectura).
29	sysPipeRead	fd *userPipe	char *buffer					Lee de un pipe, recibiendo como parámetro un file descriptor y un buffer donde deja los bytes leídos.
30	sysPipeWrite	fd *userPipe	const char *string					Escribe en un pipe con file descriptor fd el string que se le pasa

								como parámetro.
31	sysCloseFd	fd *user						Cierra el file descriptor que se le pasa como parámetro.

Cabe destacar que todas estas syscalls se acceden en Userland mediante funciones wrapper de la stdlib.

Shell

El sistema operativo bootea y automáticamente se ejecuta desde su Kernel el primer proceso del sistema: la Shell. Este tiene prioridad 1, es decir que corre en foreground, y su PID es también 1. La Shell permite acceder a todos los programas del sistema, de manera tal que se puedan testear todas las funcionalidades requeridas en el enunciado. La lista de programas se menciona en el instructivo de uso.

Lanzamiento de procesos

El lanzamiento de procesos se puede realizar mediante la función de la librería estándar createProcess. De acuerdo a la prioridad que se le pasa en el parámetro priority: si se le pasa 1, el proceso se lanzará en foreground; si se le pasa una prioridad de 2 a 10, será en background.
Se tomó como decisión que, en caso de correr un proceso en background mediante la Shell (con el argumento &), éste correrá con prioridad 2 (es decir la máxima prioridad para background).

Programas built-in

Ciertos programas son built-in de la Shell: esto quiere decir que no corren como procesos, sino que son funciones llamadas por la Shell que forman parte de la misma. Algunos ejemplos son *ps*, *help*, *mem*, entre otros. Estos no tienen PID ni estado, no los listados por ls y no se pueden correr en background. Cabe destacar que ninguno de los tests es un programa built-in.

Compilación y ejecución

Luego de descargar el repositorio, navegar al mismo.

cd TP2-SO

En la misma ruta, abrir el contenedor Docker con la imágen provista por la cátedra:

docker run -v "${PWD}:/root" --privileged -ti agodio/itba-so:1.0

Luego, ejecutar un make all en la carpeta Toolchain:

cd Toolchain; make all

Para compilar el proyecto con el Memory Manager estándar, ejecutar un make all en la carpeta raíz del mismo:

make all

En cambio, para compilar con el Buddy System:

make MM=BUDDY

Para ejecutar el sistema, ejecutar el script de bash run.sh ubicado en la carpeta raíz:

./run.sh

Nota: para la ejecución es necesario tener instalado en el sistema el virtualizador QEMU.

Instrucciones de uso

Ejecución de programas

Para ejecutar un programa en la Shell, se replicó el funcionamiento de la terminal de Linux:

Para lanzar un programa, escribir su nombre. Ejemplo:

> *User$: help*

Para pasarle argumentos, escribirlos a continuación del nombre separandolos por un espacio. Ejemplo:

> *User$: testmm 75*

Correr procesos en background

Para ejecutar un proceso en background, escribir su nombre, sus argumentos (opcional) y luego el caracter '&'. Ejemplo:

> *User$: testproc 15 &*

Nota: los programas built-in de la Shell no pueden ser ejecutados en background.

Correr dos procesos comunicados por un pipe

Para hacer un pipe entre la salida estándar de un proceso y la entrada estándar de otro, utilizar el símbolo '|'. Ejemplo:

> *User$: wpipe | rpipe*

Comandos soportados por el sistema

La lista de comandos soportados por la Shell se puede ver utilizando el comando *"help",* y es la siguiente (aquellos resaltados son built-in):

- *help*: Lista todos los comandos.
- *clear*: Limpia la terminal.
- *date*: Muestra la hora del sistema en formato UTC.
- *fibo*: Lista la serie de Fibonacci continuamente.
- *primos*: Lista los números primos continuamente.
- *mem*: Imprime el estado actual de la memoria (tamaño total del memory manager, memoria disponible y memoria ocupada).
- *ps*: Imprime una lista de todos los procesos corriendo en el sistema (PID, prioridad, dirección del stack pointer, dirección del base pointer, si está en foreground o no y si está corriendo o bloqueado).
- *kill [pid]*: Mata al proceso cuyo PID se le pasa por parámetro.
- *nice [pid] [priority]*: Se le cambia la prioridad al proceso cuyo PID es el del primer argumento a la prioridad que se le pasa como segundo argumento. Debe ser un número entre 1 y 10.
- *block [pid]*: Alterna el estado del proceso cuyo PID se le pasa como parámetro entre bloqueado y listo.
- *sem*: Imprime una lista de todos los semáforos corriendo en el sistema (ID, valor, PIDs de los procesos bloqueados esperándolo).
- *cat*: Imprime por pantalla lo que recibe por entrada estándar al presionar enter. Es un proceso continuo, se cierra presionando F2.
- *wc*: Imprime por pantalla la cantidad de líneas que posee lo que se le pasa por entrada estándar.
- *filter*: Imprime el buffer que se le pasa por entrada estándar filtrando (eliminando) todas sus vocales.
- *pipe*: Imprime una lista de todos los pipes corriendo en el sistema (ID, permisos, cantidad de bytes sin leer y los PIDs de los procesos esperando por el pipe).
- *philo*: Imprime una representación visual sencilla del problema de los filósofos comensales (sincronización de proceso). Con las teclas 'a' y 'r' se agregan y remueven filósofos, respectivamente.

Además, se implementaron los siguientes tests:

- *testsargs [arg1] [arg2] [arg3] [arg4] [arg5]:* Este test tiene como objetivo mostrar la recepción de argumentos por parte de los procesos. Recibe dichos argumentos y los imprime listados en pantalla.
- *testmm [max memory percentage]*: Ejecuta el test del memory manager provisto por la cátedra. El parámetro es el máximo porcentaje de memoria que usará el test.
- *testproc [max processes]*: Ejecuta el test de proceso provisto por la cátedra. El tamaño es la máxima cantidad de procesos que se crearán.
- *testprio*: Ejecuta el test de prioridades provisto por la cátedra. Este tiene por objetivo mostrar que los cambios de prioridad en los procesos son efectivos, para esto, muestra la tabla de procesos ante cada cambio realizado.
- *testprio2 [prio1] [prio2] [prio3]*: Tiene el objetivo de demostrar el correcto funcionamiento del algoritmo del scheduler para seleccionar los procesos que corren por prioridad. Recibe 3 prioridades numéricas y luego crea 3 procesos con dichas prioridades. Cada proceso va imprimiendo continuamente en s su número de prioridad. Se espera que los procesos con mayor prioridad impriman en pantalla más veces que los que tienen menos, en la secuencia.
  Nota: se decidió separar el testeo de prioridades en testPriorities y testPriorities2 para evaluar por separado el cambio de prioridades y el correcto funcionamiento de del scheduler considerando las prioridades de los procesos.
- *testsync*: Ejecuta la parte del test de sincronización de proceso provisto por la cátedra con semáforos activados.
- *testnosync*: Ejecuta la parte del test de sincronización de proceso provisto por la cátedra con semáforos desactivados.
- *wpipe*: Imprime un texto 3 veces por salida estándar.
- *rpipe*: Lee lo que se le pasa por salida estándar y lo imprime continuamente. Junto al programa anterior se utiliza para testear que la shell crea un pipe y lo “conecta” en los procesos que levanta al usar el carácter ‘|’.
- testpipe: Tiene por objetivo mostrar la comunicación entre procesos usando pipes unidireccionales. Para esto, crea un proceso que abre un pipe y hace un read. A continuación, se muestra que dicho proceso se encuentra bloqueado dado que el pipe está vacío y se muestra por pantalla el listado de pipes para comprobar que efectivamente se creó. Luego de tres segundos crea un segundo proceso y se lo conecta al pipe. Finalmente dicho proceso escribe en el pipe desbloqueando al proceso que lee. El proceso ahora desbloqueado imprime el input recibido.

Cierre forzado de procesos:

Los procesos que corren de manera contínua (como *testmm*, *rpipe*, etc) pueden ser cerrados forzosamente mediante la tecla F2.

Nota: los programas built-in de la Shell que son contínuos (como *fibo* o *primos*) no pueden ser cerrados con F2.

Limitaciones

Escritura del teclado

En un primer momento, para pocos procesos, no presenta limitaciones. Pero a medida que se levantan más programas se empieza a notar que la Shell no es del todo responsiva.

Scheduler

El scheduler no tiene un límite de procesos, y se le otorga a cada proceso 4KB de memoria. Sin embargo, en determinados casos puede fallar el sistema en caso de levantar más de 20 procesos.

Argumentos

Cada proceso puede recibir como máximo 5 argumentos, cada uno de no más de 20 caracteres. Esto se hizo para evitar el uso de arrays dinámicos para dichos strings, usando arrays estáticos bidimensionales con un tamaño predefinido.

Semáforos

Se pueden crear hasta 15 de ellos y cada uno soporta hasta 15 procesos esperando por ellos simultáneamente. Es un límite arbitrario que podría ser aumentado.

Pipes

Se pueden crear hasta 20 pipes como límite. Al igual que antes, es un límite arbitrario. Además, los programas built-in no soportan pipes y en el caso de los procesos solo se soporta que haya un lector y un escritor en simultáneo.

Cierre forzado de procesos

Si bien se implementó el opcional de presionar la tecla F2 para salir del proceso ejecutándose en foreground, dicha implementación se hizo en el keyboard handler por simplicidad pero se podría haber encontrado una solución más elegante. Además, al presionar F2 en un programa built-in contínuo (como *fibo* o *primos*), no hace nada y la solución es terminar la shell. Dado que esta funcionalidad no era un requisito del enunciado, se agregó únicamente para facilitar la corrección del proyecto.

Programa philo

En escasas situaciones, sucede que al haber exactamente cuatro filósofos en la mesa, solo comen dos de ellos de manera indefinida. Además, se hace una syscall en cada iteración de un while lo que es costoso para el sistema (si bien se agregó un wait de 100ms, siguen siendo demasiadas syscalls).

Pipe y procesos en segundo plano

Como decisión de diseño, se planteó que al correr dos procesos con un pipe, el primero corre en background mientras que el segundo lo hace en foreground. Esto, si bien sigue la lógica de Linux, impide por ejemplo que se puedan correr programas de la forma cat | filter ya que al correr cat en background este no puede leer de stdin. Sin embargo, hay manera de probar esos procesos.

Problemas encontrados durante el desarrollo

- Cómo hacer atómico el semáforo: en un primer momento, antes de conocer la función xchg provista por la cátedra, no se encontraba manera de hacer atómicas las operaciones del semáforo.

- El pipe se salteaba el primer carácter: para solucionar este problema se cambió cómo se hacía el llamado a pipeRead y se le agregó un caso específico para cuando hay solo un carácter.

- El pasaje de parámetros a los procesos no funcionaba: en un primer momento se usaba char ** y un alloc para recibirlos pero evidentemente se pisaban en la memoria. Esto se solucionó usando un array bidimensional estático para la recepción de parámetros en *createProcess*.

- El read implementado en Arquitectura de Computadoras hacía busy waiting: para solucionarlo, se creó una lista de espera en el Kernel en donde se almacena el programa que está esperando leer de stdin aunque esto no fue resuelto del todo correctamente ya que se aprovecha del hecho de que solo un proceso puede estar leyendo de stdin a la vez.

- El programa philo no se adaptaba bien a los cambios en la cantidad de filósofos al presionar las teclas 'a' y 'd'. Este problema fue solucionado definiendo una cantidad máxima de filósofos, un nuevo estado QUITTED para los filósofos que se van y un array estático con el estado de cada uno de ellos. Además, para evitar problemas de sincronización, se espera al momento justo para poder ejercer cambios en la cantidad de filósofos.

Código reutilizado de otras fuentes

Ciertos códigos fueron reutilizados de diversas fuentes. Estas son:

https://github.com/Infineon/freertos/blob/master/Source/portable/MemMang/heap_2.c

github.com/mit-pdos/xv6-public/blob/master/pipe.c

https://aticleworld.com/implement-strtok-function-in-c/

https://www.techiedelight.com/implement-atoi-function-c-iterative-recursive/

https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/

Modificaciones realizadas a los tests provistos

En primer lugar, en todos los tests, se cambió el prototipo de la función principal para adecuarse a la forma en la que se manejan los argumentos a procesos en este proyecto (es decir, reciben un array bidimensional estático).

Memory Manager

El *testMM* provisto casi no fue modificado, simplemente se adaptaron las syscalls para adecuarlas a los nombres de las de este proyecto. Además, se agregó un mejor chequeo de parámetros a la hora de llamar al programa y se agregó una llamada al comando *mem* para poder observar en cada momento cómo está el estado de la memoria (tanto cuando la lleva al límite como cuando la libera casi en su totalidad).

Procesos

El *testProc* no fue modificado: al igual que el anterior, se adaptaron las syscalls.

Prioridades

El *testPrio* casi no fue modificado solo se adaptaron las syscalls, en particular la de createProcess, ya que la recepción de argumentos por parte de la misma era ligeramente diferente a la provista por la cátedra. Pero se agregaron prints del listado de procesos activos para demostrar el cambios de prioridades.

Sincronización

El *testSync* fue el que sufrió más modificaciones. En primer lugar, el hecho de no poseer una syscall de *wait(pid)* impidió que se pudiera correr directamente, ya que no había forma de esperar a que terminaran los procesos para imprimir el final value. Entonces, se optó por ir imprimiendo el valor en cada iteración de los ciclos, y lógicamente el valor final será el último que se imprima. También se agregaron otras impresiones de información que resulten relevantes.

Además, para no complicar demasiado el código, se decidió por motivos prácticos separar el test en dos tests separados sin parámetros: *testSync* corre con semáforos, *testNosync* corre sin semáforos.

Warnings y análisis estático

Warnings en compilación

No se presentan warnings en compilación como es requerido por el enunciado.

PVS Studio

En primer lugar se presentan warnings que corresponden a Barebonesx64, por lo que no corresponden al código del proyecto.

Con respecto a los otros, se trata de warnings sobre el casteo de punteros a enteros, algo común en el proyecto por motivos funcionales. Un ejemplo de esto es cuando se inicia el sistema por primera vez, al momento de saltar de Kernel a Userland para cargar la Shell. La función createProcess requiere necesariamente un instruction pointer en formato entero, por lo que es necesario castear la dirección de memoria en hexadecimal a entero por lo que el warning está justificado.

Por último, aparecen advertencias de posible desreferenciación de null pointers, por ejemplo en el scheduler. Se justifica este warning ya que es seguro que dicho puntero jamás será null.

/root/Kernel/utils/scheduler.c V522 There might be dereferencing of a potential null pointer 'scheduler->current

CPP Check

En ciertos casos donde cppcheck pide cambiar una variable de lugar para reducir el scope de la misma, se decidió dejarlas fuera para mayor claridad del código:

[Userland/SampleCodeModule/programs.c:75]: (style) The scope of the variable 'limit' can be reduced.

[Userland/SampleCodeModule/programs.c:76]: (style) The scope of the variable 'isPrimo' can be reduced.

Además, al ser CPP un analizador de código de C, no reconoce las llamadas a función que se realizan desde Assembler.

[Userland/SampleCodeModule/_loader.c:13]: (style) The function '_start' is never used.

[Kernel/utils/scheduler.c:197]: (style) The function 'contextSwitching' is never used.

[Userland/SampleCodeModule/tests-TP2/test_util.c:48]: (style) The function 'endless_loop_print' is never used.

[Userland/SampleCodeModule/programs.c:98]: (style) The function 'infoRegisters' is never used.

[Kernel/kernel.c:41]: (style) The function 'initializeKernelBinary' is never used.

[Kernel/interruptions/irqDispatcher.c:12]: (style) The function 'irqDispatcher' is never used

Conclusión

En conclusión, consideramos que todos los requerimientos fueron implementados y se encuentran funcionando correctamente. El kernel funciona como es debido y presenta funciones para probarlos.