INFORME

S.O. - SISTEMAS OPERATIVOS

Trabajo Práctico 2

Construcción del Núcleo de un Sistema Operativo

- Grupo 8: Matías Heimann (57503), Johnathan Katan (56653), Lóránt Mikolás (57347) y Gabriel Silvatici(53122).

- Vencimiento de la entrega: 09/05/2018 (23:59).
- Profesor: Horacio Merovich.
- Adjuntos: Ariel Godio, Rodrigo Rearden.

Introducción

El objetivo de este informe es presentar las distintas etapas de la construcción del núcleo de un sistema operativo a partir del trabajo realizado en Arquitecturas de Computadoras.

Se propone exponer los diferentes problemas encontrados y sus soluciones, así como las diferentes estructuras, algoritmos y syscalls implementadas.

A modo de referencia se recomienda también tener presente el informe realizado para la entrega de Arquitecturas de Computadoras que se encuentra en el repositorio entregado y se aclara que partes de aquel informe como por ejemplo la explicación del uso de comandos fueron repetidas en el actual informe.

Finalmente buscamos recapitular conclusiones de la experiencia y aspectos que podrían ser mejorados en futuras iteraciones.

Implementación

A continuación se explicarán el funcionamiento de los algoritmos y de las estructuras realizadas para el funcionamiento del trabajo.

Se debe tener en cuenta que la última sección de este apartado presenta las system calls del sistema.

Requisitos previos y Drivers

Por lo que se refiere a los drivers, se utilizaron los mismos que en Arquitecturas de Computadoras: driver de teclado y driver de video.

Por otra parte, al desarrollar todo el equipo en el mismo entorno de ubuntu no se consideró necesario utilizar Docker.

Physical Memory Management

En cuanto al memory manager se decidió implementar un buddy-allocator. Para entender su funcionamiento se utilizaron las distintas fuentes que se encuentran. Se debe tener en cuenta que en la implementación realizada el tamaño mínimo de memoria que se le puede exigir es de 4KB contiguos.

En relación al tracking de las páginas se decidió por limitaciones de tiempo para la implementación guardar una metadata en la primer posición de cada bloque de memoria indicando si el mismo está libre o no, si es un bloque izquierdo o derecho y el orden del bloque. Si el bloque es izquierdo o derecho se usa para cuando se libera un bloque de memoria, como se sabe cada bloque tiene un buddy (excepto el primer bloque que ocupa toda la memoria), entonces, la información hace referencia si su buddy está en la posición contigua a la derecha o a la izquierda. Si es izquierdo, estará a su derecha y viceversa. El bloque del orden hace referencia al tamaño del bloque, sabiendo que el orden 0 es el tamaño del mínimo, 4096 bytes incluyendo la metadata inicial. El tamaño del bloque es de 4096 * 2^n siendo n el orden del bloque, en el tamaño se incluye la metadata.

Procesos, Context Switching y Scheduling

El siguiente punto trata sobre el manejo de los procesos, cómo se realiza el context switching y el scheduling. Para crear un nuevo proceso desde userspace simplemente se le debe pasar un puntero a función a la función newProcess. La misma realiza una syscall que añade el proceso a la tabla de procesos (otorgando un nuevo pid). Marca al proceso como proceso nuevo y le asigna una prioridad. Además se le aloca una página para guardar el stack del proceso y se arma un stack “artificial” de manera tal que cuando se lo desencole por primera vez del Round Robin se desarme la pila de manera correcta y comience su ejecución el nuevo proceso luego de llamar iretq. Para el armado de este stack se siguió el procedimiento que se encuentra en sistema operativo de rodrigo rearden en gitlab (link en la bibliografía).

Luego se añade el proceso al Round Robin (de quantum un tick) con prioridades y se le marca el “listo” al proceso. Para manejar las prioridades se implementaron tres colas de procesos: *high priority*, *medium priority* y *low priority*. Sin embargo, en la versión entregada todos los procesos nuevos se marcan como de alta prioridad (aspecto que se debe mejorar en futuras iteraciones).

Una vez que se inicializan la tabla de procesos , el primer proceso a correr y el round robin se habilitan las interrupciones del timer tick.

Para atender las interrupciones del timer tick, se implementó el procedimiento explicado en la clase práctica.

La idea central es no perder el contexto de los procesos que se encontraban en ejecución.

También se implementó un exit para liberar la página de un proceso y marca en la tabla de procesos al mismo como finalizado.

IPCs

<u>Envío y recepción de mensajes:</u>

Para el correcto envío y recepción de mensajes entre procesos, se decidió crear una estructura llamada *MessageHolder*, que contiene el *pid* del proceso que envía un mensaje, el *pid* del proceso que recibe un mensaje, un *mutex* que utiliza internamente para sincronizar el envío del mensaje, un buffer interno por donde se envía el mensaje, y por último, un *id* que identifica a este *MessageHolder*.

Lo que permite el *id* es que un proceso pueda acceder a un *MessageHolder* creado por otro proceso mediante su *id*, de esta manera logrando la comunicación entre procesos. Un proceso, una vez obtenido un *MessageHolder* mediante el *id* correspondiente, puede envíar y recibir mensajes a través de ese *MessageHolder*.

Cuando un proceso intenta leer de un *MessageHolder* y este *MessageHolder* está vacío, el proceso queda en estado de espera (no activa, sino que el scheduler no lo ejecuta) hasta que otro proceso envíe un mensaje. De manera similar, cuando un proceso intenta leer del *MessageHolder* y el buffer de este *MessageHolder* está lleno, el proceso que lee se bloquea hasta que otro proceso intente leer de este *MessageHolder*.

Se proveen una serie de System Calls para que un usuario pueda hacer uso de la comunicación entre procesos. El prototipo de estas funciones se pueden encontrar en el archivo *stdlib.h* en el trabajo práctico, en *user space*.

En la *shell* del trabajo práctico, se puede ejecutar el comando *ipc demo* para correr una demostración del uso de los *MessageHolders*. El archivo de este demo se llama *MessageHolderTesting.c*, y se puede encontrar en *user space*.

Es importante mencionar que el bloqueo y desbloqueo de los procesos que intervienen en la comunicación se maneja mediante *Mutexes*. A continuación, se explica cómo estos mutexes fueron implementados.

Mutexes:

Para proveer un sistema de sincronización entre procesos se implementaron *Mutexes*. Estos son usados por los *MessageHolders* internamente, y también pueden ser usados por el usuario del sistema operativo, para desarrollar aplicaciones que requieran de sincronización.

Un *mutex* consiste de dos estados, bloqueado y desbloqueado. Cuando se intenta bloquear a un *mutex* y éste ya estaba bloqueado, el proceso que llamó debería bloquearse hasta que se desbloquee el mutex. La comparación y asignación del estado del *mutex* al llamarse a una función que lo bloquee, debe ser atómica, es por esto que se utilizó la instrucción *cmpxchg*, que permite hacerlo en una sola instrucción atómica.

Los *mutexes* implementados también cuentan de una cola interna de procesos. Cada vez que un proceso es bloqueado por el *mutex*, se agrega a la cola y se asigna como estado de proceso a *WAITING*, de esta manera el scheduler no ejecuta ese proceso y se evita realizar busy waiting. De la misma manera, cuando un proceso es desbloqueado por un *mutex*, se retira de la cola de procesos de ese *mutex* y se agrega de nuevo al scheduler. Cambiando el estado del proceso cuando se bloquea y se desbloquea, se aprovecha el diseño implementado del scheduler y se evita realizar busy waiting.

También, al igual que con los *MessageHolders*, se pueden crear *Mutexes* con un *id* definido por el usuario, y se pueden obtener *Mutexes* creados por otros procesos mediante el *id*. De esta manera, el *mutex* puede ser usado por múltiples procesos.

Se proveen una serie de system calls al usuario para que pueda usar *Mutexes* en *user space*. Los prototipos de estas se pueden encontrar en *stdlib.h*, en *user space*.

En la *shell* del trabajo práctico, se puede ejecutar el comando *mutex demo* para correr una demostración del uso de los *Mutexes*. El archivo de este demo se llama *MutexTesting.c*, y se puede encontrar en *user space*.

System Calls

En relación con las syscalls se presenta la siguiente tabla para explicar las mismas. Las syscalls soportadas son:

Syscall	RAX	RBX	RCX	RDX
print	3	uint64_t fd	uint64_t buffer	uint64_t count
scan	4	uint64_t fd	uint64_t buffer	uint64_t count
clearScreen	5	-	-	-
paintPixelAt	6	uint64_t x	uint64_t y	-
XResolution	7	-	-	-
YResolution	8	-	-	-
displayTimeData	9	-	-	-
memoryAllocation	10			
memoryFree	11	void*	-	-
leaveProcess	12	-	-	-
createProcess	13	void*	int argc	char** argv
getProcessPid	14	-	-	-
getProcessesInfo	15	processesInfoTable* processes	-	-
generateMutex	16	char *id	-	-
getMutex	17	char *mutexId	-	-
deleteMutex	18	char *mutexId	-	-
blockMutex	19	mutex *mutexToLock	-	-
unlockMutex	20	mutex *mutexToUnlock	-	-
generateMessage Holder	21	char *id	-	-
getMessageHolder	22	char *id	-	-
deleteMessageHol der	23	char *id	-	-
send	24	messageHolder *message	char *data	int size

receive	25	messageHolder *message	char *storageBuffer	int size
waitProcess	26	int pid		

Aplicaciones de Userspace

Se debe tener en cuenta como el manejo de excepciones no fue el foco de este trabajo práctico no se realizaron rutinas de limpieza luego de que las mismas se lancen. Los aplicaciones disponibles son:

- **echo**: para imprimir por terminal un string. Se debe tener en cuenta que no se puede ejecutar en background el comando.
- **clear**: limpia la pantalla y la pone en negro.
- **divide by cero**: genera la excepción de división por cero.
- **overflow**: genera la excepción de overflow.
- **invalid opcode**: genera la excepción de opcode inválido.
- **time**: muestra la información proveída por el real time clock. Presenta la fecha actual y la hora actual.
- **help**: muestra los comandos disponibles.
- **graph**: ejecuta un programa para graficar funciones.
  Dentro de la aplicación se puede ejecutar:
  - **addplot:** este comando sirve para graficar una función polinómica de grado menor o igual a dos. Luego de ejecutar el comando el usuario deberá ingresar por separado los coeficientes a, b y c. Seguidos por un enter.
  - **exit:** cierra el programa y vuelva a la shell.
- **ps**: muestra una tabla con los procesos que se encuentran en la tabla de procesos.
- **prodcons**: demuestra una solución al problema del productor-consumidor. En este caso buscamos utilizar una cantidad de productores y consumidores dinámica. Cada productor genera una cantidad de mensajes constante luego de la cual finaliza su ejecución. En este comando se puede demostrar el buen funcionamiento de los mutexes. Se comienza con un productor y un consumidor. Con p se agrega un productor, con c un consumidor y con e se realiza un exit. Se debe tener en cuenta que resulta complicado salir cuando hay varios procesos corriendo porque el proceso principal no recibe la presión de la tecla.
- **four process demo**: crea 4 procesos imprimiendo un mensaje desde cada uno y luego se ejecuta el comando ps. Para mostrar que se encuentran en la tabla correctamente.
- **ipc demo:** crea un proceso que ejecuta una serie de funciones que demuestran el funcionamiento de la comunicación entre procesos.
- **mutex demo:** crea un proceso que ejecuta una serie de funciones que demuestran el funcionamiento de los mutexes.

Tests

Antes que nada se debe aclarar que no se agregaron test unitarios para las funcionalidades previamente entregadas del trabajo práctico de Arquitecturas de Computadoras. Sin embargo, sí se realizaron tests para funcionalidades particulares a la actual entrega.

En esta entrega se presentan test para la queue utilizada y para el memoryManager. Se encuentran situados en la carpeta testing. Para compilarlos se encuentra con el script compile.sh. Otras evaluaciones fueron puestas como demos en el userland.

Resultados

Como se mantuvo una modalidad de trabajo similar a Arquitecturas de Computadoras se cita lo que se explicó en ese informe.

"(...) vale la pena mencionar que a lo largo del desarrollo el trabajo fue probado sobre QEMU para acelerar las diferentes pruebas.

Debido a que no se contó con la computadora real en el laboratorio C, el trabajo no pudo ser preparado para funcionar en la máquina requerida. Sin embargo, el tp compila sin errores ni *warnings* y además funciona correctamente sobre QEMU."

Imagen 1: tabla de procesos ps.

Imagen 2: comando time

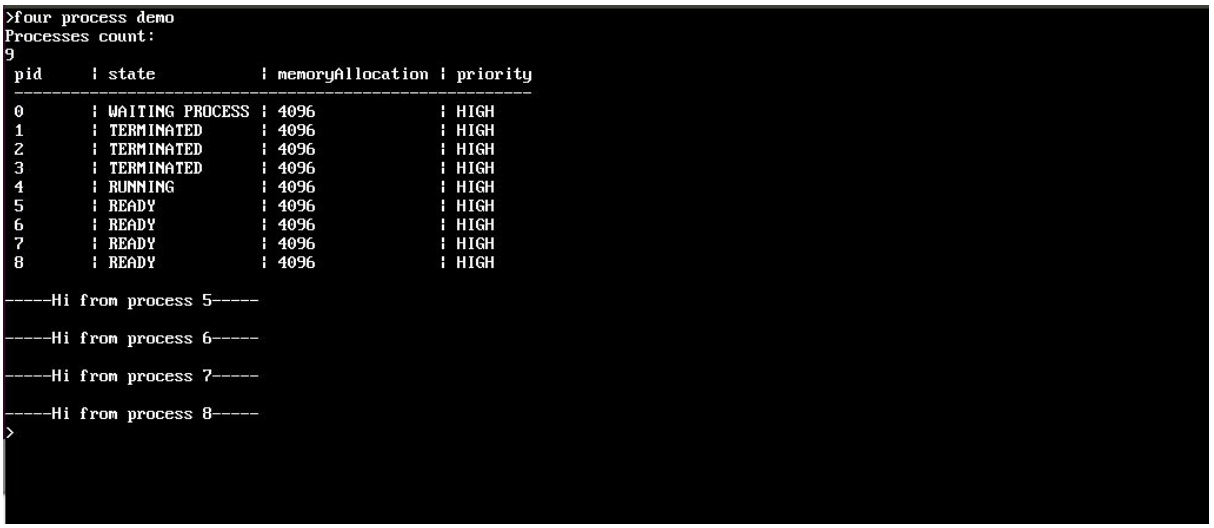

Imagen 3: four process demo.

Imagen 4: shutdown con comando exit.

Estándares utilizados para el estilo del código

Para formatear el código y que todos los archivos *.c* y *.h* respetaran el mismo estándar se utilizó la herramienta open source *clang-format*. Se usó un archivo llamado *.clang-format* que especifica los estándares del código, este archivo se puede encontrar en el directorio principal del proyecto. Se puede observar que el estándar utilizado es *LLVM,* con algunas modificaciones en cuanto a indentación con *tabs* que se puede ver en el archivo *.clang-format*.

Por último, como *clang-format* permite solo formatear un archivo en particular, se procedió a escribir un script para poder ejecutar esta herramienta sobre cada archivo del proyecto. Este script fue escrito en *python*, y se puede encontrar en el archivo *formatter.py* en el directorio principal del proyecto.

Conclusión

En conclusión este trabajo nos permitió entender más claramente cómo funciona un sistema operativo internamente en aspectos relacionados a: manejo de memoria, cambios de contexto , creación de procesos, planificación, comunicación entre procesos y exclusión mutua. Además se pudo mantener la división entre Kernel Space y User Space como se implementó en el trabajo de Arquitecturas de Computadoras.

Por último vale la pena marcar la facilitación del debugging al realizar test unitarios de estructuras y funciones previo a su implementación en el trabajo.

Instrucciones para ejecución

Se compiló sobre ubuntu 16.04 LTS.

Instalación

En linux se debe contar con nasm, qemu, gcc-5 y make.

1. Extraer en una carpeta el trabajo.
2. Abrir la carpeta en la terminal.
3. Moverse al directorio **TP-Arqui/RowDaBoat-x64barebones-d4e1c147f975/Toolchain/**
4. Ejecutar comando **make all**
5. Luego en **TP-Arqui/RowDaBoat-x64barebones-d4e1c147f975/** ejecutar comando **make all**

En caso de que no funcione probar hacer **make clean** antes de **make all.**

Utilización sobre QEMU

Desde la terminal en la carpeta **TP-Arqui/RowDaBoat-x64barebones-d4e1c147f975/** ejecutar el comando:

make fromzero

Este comando, hace make clean, make all y ejecuta el tpe.

Problemas

Ante cualquier dificultad, nos pueden contactar en el correo institucional del ITBA a cualquiera de los integrantes del grupo.

Bibliografía

A continuación se encuentra la bibliografía utilizada para realizar el trabajo práctico. Se debe tener en cuenta que también fueron usados elementos teóricos y prácticos enseñados en clase y proveídos por la cátedra como parte del material que se encuentra en el campus.

- Base del trabajo práctico: https://bitbucket.org/RowDaBoat/x64barebones/wiki/Home
- Referencia para driver de teclado: http://www.osdever.net/bkerndev/Docs/keyboard.htm
- Referencia para manejo de excepciones: https://os.phil-opp.com/better-exception-messages/
- Referencia para el manejo de la estructura con la información del modo video: http://www.delorie.com/djgpp/doc/ug/graphics/vesa.html

- Referencia para el driver del real time clock: http://wiki.osdev.org/CMOS
- Referencia para el armado del stack del proceso nuevo: https://gitlab.com/RowDaBoat/Wyrm
- Message passing: https://wiki.osdev.org/Message_Passing
- Estándar LLVM: https://llvm.org/docs/CodingStandards.html