Instituto Tecnológico de Buenos Aires

TP2: Construcción del Núcleo de un Sistema Operativo y estructuras de administración de recursos.

72.11 - Sistemas Operativos

GRUPO N°4

Integrantes:

Aramburu, Paz - 62556
Neimark, Luciano - 62086
Tordomar, Nicolás - 62250

Docentes:

Aquili, Alejo Ezequiel
Godio, Ariel
Mogni, Guido Matías

Fecha de entrega: 5 de Junio 2023

Introducción

En el siguiente informe se detalla sobre la implementación del segundo trabajo práctico de la materia. Este trabajo es una continuación del trabajo realizado en la materia Arquitectura de Computadoras durante el segundo cuatrimestre del 2022. Vamos a profundizar en cuestiones como problemas que tuvimos durante el desarrollo, decisiones de diseño, y funcionamiento específico de cada área que compone al kernel y al userspace.

Por último se detalla cómo compilar y ejecutar el proyecto.

Scheduler y procesos

Para el scheduler implementamos un round robin con prioridades como se pedía en la consigna. Este scheduler consta de 5 listas donde se colocan los procesos y dependiendo de qué lista están, la prioridad que tienen. Al crear un proceso, todos arrancan con la máxima prioridad y si al momento de correr un quantum largo, lo puede correr entero, se decrementa la prioridad del proceso. Por otra parte, implementamos una estrategia para que los procesos puedan volver a escalar en prioridades si no corren durante todo su quantum para evitar inanición.
También, se implementó un método para poder setear el valor de la prioridad de un proceso durante el tiempo de ejecución y la posibilidad de un proceso de poder esperar a su hijo haciendo waitpid.
En cuanto a la ejecución de los procesos se distingue entre un proceso que está corriendo en background y uno que está corriendo en foreground colocando un ; después del comando. Los estados posibles de un proceso son bloqueado, ready, zombie y running,

Semáforos

Para el correcto funcionamiento de un sistema operativo es necesario poder soportar semáforos para evitar race conditions. Para implementar semáforos lo que hicimos fue guiarnos con la implementación de POSIX y también con el ejercicio de la guía donde se pide una implementación de semáforos usando semáforos.
Durante un momento del desarrollo nos encontramos con el problema de que necesitábamos poder tener exclusión mutua sobre la variable que incrementa y decrementa el semáforo así que tuvimos que implementar el caso particular del mutex en assembler con xchg.

Pipes

La implementación de pipes consta de un buffer al que se le puede escribir o leer del mismo.
En una primera instancia, guiándonos con la problemática de *producer and consumer*, estuvimos mucho tiempo decidiendo cuál era la mejor forma de impedir que dos procesos escriban o lean del buffer al mismo tiempo. Al consultarlo en el

foro, se nos recordó que las interrupciones son trapgates, por lo que con solo hacer down de un semáforo podemos garantizar que vamos a estar solos trabajando en el buffer. La implementación de la api de pipes tiene un comportamiento similar a los pipes de POSIX.
Otra forma de crear pipes, a parte de utilizando la api en algún proceso, es a través de la shell escribiendo proc1/proc2 logrando que la entrada estándar del proc1 sea la shell, la salida el pipe, la entrada de proc2 el pipe y por último, la salida de proc2 el STDOUT.
Se decidió que el identificador de los pipes fueran números en vez de letras y se agregó una opción de crear pipes anónimos para facilitar el desarrollo en varias ocasiones.

Memory Manager

El desarrollo del memory manager nos requirió bastante tiempo del TP.
Comenzamos intentando de implementar las iteraciones más básicas que ofrecía la página FreeRTOS para poder incrementalmente entender cómo funcionaba un memory manager. Luego de varias pruebas decidimos implementar una versión parecida al Heap 4. Este heap consta de una lista de nodos de memoria que a medida que se hace malloc, se va fragmentando un gran bloque inicial. La implementación que elegimos cuenta con merge de bloques libres adyacentes, mejorando significativamente la experiencia y la eficiencia del manager. Por otro lado, la segunda versión del memory manager que tuvimos que implementar fue un Buddy Memory Manager. Lo que nos benefició mucho al implementar el manager fue que en varias páginas de internet y en varios repositorios de github se encontraban versiones que pudimos usar de inspiración y para poder terminar de entender cómo funcionaba. Otro recurso que nos sirvió fue este video de youtube donde pudimos ver gráficamente cómo era el buddy para orientarnos en el desarrollo.
Otro desafío por el cual pasamos fue permitir en tiempo de compilación, decidir qué memory manager vamos a usar para correr el TP.

Aclaraciones PVS

- Las primeras 9 notas/errores corresponden al archivo BMFS.c ubicado dentro del bootloader. Esos archivos fueron provistos por la cátedra por lo que no se modificaron.
- Las notas restantes corresponden a un detalle que se hizo en el archivo drawings.c. A pesar de que no se haya reutilizado al máximo el código, refleja únicamente una cuestión estética y no de calidad de código.

Instructivo de compilacion y ejecucion

1. Entrar al contenedor de docker provisto por la cátedra.
2. Ir a la carpeta Toolchain
3. Ejecutar make clean
4. Ejecutar make all
5. Ir a la root del proyecto
6. Ejecutar “make clean”
7. Ejecutar “make all”
    a. Si se ejecuta “make all MM=BUDDY” compila con el buddy memory manager
    b. Si se ejecuta el “make all MM=STANDARD” compila con el heap que utiliza first fit.
8. Salir del contenedor y correr ./run.sh.