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

2ndo Trabajo Práctico – Grupo 12

72.11 Sistemas Operativos

Segundo cuatrimestre 2023

Integrantes:

Deyheralde, Ben (Legajo: 63559)

Gonzalez Rouco, Lucas (Legajo: 63366)

Mutz, Matias (Legajo: 63590)

Profesores:

Aquili, Alejo Ezequiel

Godio, Ariel

Mogni, Guido Matias

Índice

1. Introducción

El objetivo del trabajo práctico nº2 consiste en aprender e implementar las funcionalidades básicas que debe ofrecer el Kernel de un Sistema Operativo para gestionar y administrar recursos. Para ello, utilizamos la base de un proyecto final de la materia correlativa anterior “Arquitectura de Computadoras” y le añadimos lo visto en esta materia: memory manager, scheduler, mecanismos de sincronización y mecanismos de IPC (Inter Process Communication). En este informe vamos a desarrollar las decisiones de diseño tomadas, cómo compilar y ejecutar el TP y cómo probar que las implementaciones realizadas funcionen correctamente.

2. Decisiones tomadas durante el desarrollo

En esta sección detallaremos las decisiones de diseño que tomamos en cada "feature" implementada. Fueron implementadas en el orden sugerido en la consigna.

2.1. Memory Manager

Al principio utilizamos el ejemplo "dummy" que vimos en la clase práctica para luego ir agregándole funciones y hacer que sea cada vez más completo. Luego, nos inclinamos por una "Free List" donde guardamos los nodos de memoria libre y también los de memoria ocupada. Tomamos esta decisión debido a que consideramos que la búsqueda de bloques libres iba a ser más eficiente que en un "Bitmap" y esto fue lo que priorizamos.

2.2. Scheduler

Para el Scheduler implementamos un Round-Robin con prioridades, como solicita la consigna. Para esto utilizamos 7 colas de procesos donde cada una representa una prioridad y dentro de cada prioridad funciona un Round-Robin. Hemos declarado 7 prioridades, de las cuales solamente 5 son prioridades reales y 2 son utilizadas internamente por nosotros:

1. Prioridad 0: utilizada solamente para el proceso "Idle". Ningún otro proceso va a tener esta prioridad y es la prioridad más baja de todas, solamente se corre el "Idle" cuando no hay otro proceso Ready.

2. Prioridades 1 a 5: siendo 1 la prioridad más baja y 5 la más alta, en estas prioridades vamos encolando los procesos Ready.

3. Prioridad 6: utilizada para procesos bloqueados únicamente.

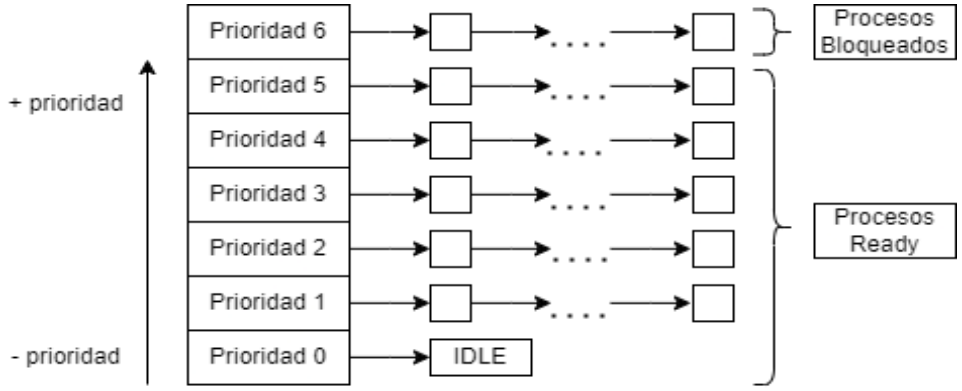

Figura 1: Esquema de cola de prioridades

La prioridad 0 y 6 no son prioridades reales sino que nosotros internamente lo manejamos de esta manera porque nos facilitó el manejo de los procesos, pero no son prioridades aceptadas en el comando “nice” por ejemplo.

2.3. Sincronización

A la hora de implementar los semáforos, nos basamos en cómo funcionan los semáforos en Linux, ofreciendo las funciones de semInit, semPost y semWait y semClose.

2.4. Pipes

Para el caso de los pipes, implementamos una struct que tiene un buffer circular, un PID del proceso de entrada y uno para el proceso de salida y luego variables que utilizamos para manejar la escritura y lectura y el bloqueo en caso de que sea necesario. De la misma manera que con los semáforos, brindamos la funcionalidad estándar: openPipe, closePipe, writePipe y readPipe.

3. Instrucciones de compilación y ejecución

Es requisito para poder ejecutar el programa tener Docker corriendo.

Con los comandos **chmod u+x ./docker.sh** y **chmod u+x ./run.sh** se le da permisos de ejecución a los archivos docker.sh y run.sh.

Con el comando **./docker.sh se** descarga la imagen de **agodio/itba-so:2.0** de Docker que es donde se compilará el proyecto. A su vez, abrirá el contenedor y luego se ejecutará el programa.

Para utilizar el TP:

- Con el Memory Manager estándar es con el comando ./docker.sh
- Con el Buddy Memory Manager es con el comando ./docker.sh -b

4. Pasos a seguir para demostrar el funcionamiento de cada uno de los requerimientos

4.1. Memory Manager

Se puede testear el funcionamiento del Memory Manager con el comando “tm” desde la Shell seguido por un número. Este número va a ser la cantidad máxima de bytes que asigne el Memory Manager. Luego el MM va a asignar bloques de memoria de X bytes, siendo X un número entre 1 y esta cantidad máxima.

tm <cantidad máxima de bytes a asignar>

4.2. Procesos y Scheduler

Utilizando el comando “tp” desde la Shell seguido por un número X y un ampersand (para correrlo en background) se crean X procesos y se les cambia el estado aleatoriamente. Al correrlo en background, se puede ir poniendo el comando “ps” para ver cómo se van creando, finalizando y cambiando el estado de los procesos.

tp <número de procesos> &

También se puede probar el cambio de prioridades utilizando el comando “nice” seguido por el PID del proceso y un número entre 1 y 5. Luego, utilizar el comando “ps” para ver que efectivamente se modificó su prioridad. No es obligatorio correr el testeo en background pero haciéndolo de esta manera es más fácil entender el correcto funcionamiento de nuestra implementación.

4.3. Sincronización

Para testear el funcionamiento de los semáforos, se puede usar el comando “ts” de la siguiente manera:

ts <num1> <num2>

num1 puede ser cualquier número positivo e indica la cantidad de sumas y restas que va a realizar el test. En el caso de num2, este indica si queremos que se usen o no semáforos durante el testeo.

En caso de ser 0, no se activan los semáforos y el resultado del test puede ser cualquiera.

En caso de ser 1, se activan los semáforos y el resultado del test debe ser 0, que es el comportamiento esperado.

Otra manera de probar que la sincronización funciona correctamente, es mediante el comando “philo” ya que internamente utiliza semáforos para solucionar el problema de los filósofos.

4.4. Pipes

Para ver el correcto funcionamiento de los pipes se puede usar el símbolo “|” en la Shell. Por ejemplo: “help | wc” o “help | filter”.

5. Citas de fragmentos de código reutilizados de otras fuentes

Para resolver el problema de los filósofos utilizamos como guía una implementación de “GeeksForGeeks”:

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

6. Errores de PVS-Studio

Al utilizar PVS-studio, los errores encontrados son todos “falsos positivos”. El archivo “reports.tasks” se encuentra en la root del proyecto. Muchos de estos errores eran de “bfms.c”, un archivo inicial del Kernel que nunca fue modificado. Había además gran cantidad de comentarios acerca de printf y scanf, pero esto se debe a la lista de argumentos variables que se pasa como argumentos al llamar a la función. Hay comentarios acerca del while(1) de la shell que indica que podría ser un loop infinito pero es la funcionalidad esperada. Este tipo de errores se repiten por lo que los consideramos insignificantes y falsos positivos.

7. Problemas encontrados durante el desarrollo

Tuvimos una gran cantidad de problemas con el scheduler. Consideramos que fue la parte más tediosa y la que nos llevó más tiempo. Sin embargo se pudieron completar todas las consignas del trabajo.

8. Limitaciones

Las limitaciones del trabajo son pocas. La mayor limitación que se puede encontrar es que al matar un proceso utilizando el comando “kill *pid*”, si el proceso utilizaba un semáforo, ese semáforo no es destruido, por lo que podemos encontrar un memory leak.

También hay una máxima cantidad de procesos que se pueden tener a la vez, estando este valor definido por MAX_PROCESSES, al igual que de semáforos y pipes.

Además, en el memory manager, reservamos 1 MB entero para la lista con los nodos y su información. Consideramos que es espacio suficiente para mostrar el uso del trabajo, sin embargo sabemos que existe la posibilidad de que no haya espacio para más nodos.

Finalmente, no consideramos que esto sea una limitación pero lo comentamos porque quizás funciona de manera un poco anti-intuitiva ya que no es como en Linux: al matar un proceso con CTRL+C, no aparece la prompt de la Shell de manera inmediata sino que hay que apretar nuevamente una tecla.