Trabajo Práctico N°2 - Entrega Final

Segundo Cuatrimestre 2024

Profesores:

- Godio, Ariel
- Aquili, Alejo Ezequiel
- Mogni, Guido Matías
- Gleiser Flores, Fernando
- Beade, Gonzalo

Integrantes:

- Bonesi, Franco - 64239
- Romanato, Matías - 62072
- Novillo, Valentina - 63156

Fecha de entrega: 11 de Noviembre de 2024

Introducción

Este trabajo aborda la construcción de un sistema operativo monolítico de 64 bits, partiendo de los conocimientos adquiridos sobre la API de sistemas UNIX. Se desarrollaron módulos clave como la administración de memoria mediante un **Buddy System**, multitarea preemptiva con cambio de contexto (**context switching**), y **scheduling** de procesos mediante **Round Robin** con prioridades. Además, se implementaron mecanismos de comunicación y sincronización de procesos con **pipes** y semáforos libres de **busy waiting**. Para facilitar la interacción, el sistema incluye una **shell** personalizada, que permite ejecutar comandos en **foreground** y **background**, y gestionar la comunicación entre procesos.

Este proyecto no solo pone a prueba los conceptos fundamentales de administración de recursos y procesos, sino que también refleja las prácticas de desarrollo de un kernel operativo básico, listo para ejecutar aplicaciones en un entorno controlado y eficiente.

Consideraciones:

- **Bloqueo de procesos aunque estén en estado “ready”**

  Para esta entrega decidimos adaptarnos al funcionamiento del test “test_processes” proporcionado por la cátedra, en el cual se considera como un caso correcto el de bloquear procesos que estén en estado “ready”. En un comienzo, nosotros no consideramos a este como un caso válido, basándonos en lo que se dió en las clases teóricas. Es decir, nosotros entendíamos que un proceso sólo podía pasar a estado “blocked” si estaba corriendo (es decir, en estado “running”). Por lo que pasar de “blocked” a “ready” era algo incorrecto.

- **Utilización de estructuras para distintas funcionalidades**

  Decidimos utilizar varias estructuras para consolidar la información que refiere a una funcionalidad, para lograr más claridad y para que acceder a la información sea más fácil. Esto lo implementamos para funcionalidades como el scheduler, PCB de un proceso y la información mostrada al hacer ps.

- **Stacks creados de forma constante**

  Decidimos crear los stacks de forma constante todos juntos al inicializar el kernel reservando una zona de memoria a partir de la dirección 0x600000, en vez de forma dinámica con mallocs, definiendo bloques de 4K para cada stack.

- **Cálculos de los PIDs**

  Como decidimos tener una cantidad fija de procesos máximos y tener todos los stacks definidos en una zona de memoria constante y hay la misma cantidad que procesos, decidimos usar la dirección de inicio de la zona de stacks y el base pointer de cada stack para calcular los PIDs, por lo tanto cada posible PID estaría asociado a un único stack, y al liberar el proceso, el PID estará nuevamente disponible.

- **Margen entre la memoria asignada para los stack y la memoria dinámica**

  Optamos por dejar un margen de memoria entre lo que ocupan los stacks y donde comienza la memoria dinámica. Es decir, hay un espacio de memoria no utilizada desde donde termina la memoria asignada para los stacks y donde comienza el espacio para la memoria dinámica.

- **Tamaño del bloque de memoria que devuelve nuestro malloc del memory manager**

  Fuimos modificando el tamaño de los bloques de memoria que retorna nuestro malloc a medida que íbamos viendo que las distintas estructuras que queríamos crear necesitaban tamaños de bloques más grandes. Es decir, había momentos donde el bloque que retornaba el malloc no era lo suficientemente grande como para contener todo lo que nuestras estructuras contenían.

- **Cantidad máxima de procesos**

  Definimos que la cantidad máxima de procesos sea 128 ya que es potencia de dos y permite que los cálculos sean redondos. Dicho número está definido en una constante “MAX_PROCESSES”.

- **Escala para definir las prioridades va del 1 al 5 y los quantums son de 55 ms, al igual que lo que tarda en interrumpir el timer tick**

- **Round Robin con prioridad**

  Para la implementación del algoritmo Round Robin con prioridad del scheduler decidimos que para la ejecución de un proceso, este se ejecute tantas veces seguidas como su prioridad más 1, lo que en total sería la longitud del quantum por su prioridad más uno.

- **Tomamos como convención que el vector de args finaliza en cero o NULL para poder reconocer dónde terminan los argumentos**

- **Proceso DEFAULT**

  Este sería el proceso que se corre cuando no hay ningún otro proceso para correr. Además, en cuanto a la adopción de procesos hijos, implementamos que a un proceso hijo lo adopte su padre y de no ser así su abuelo, etc. , de modo que si no hay ningún otro proceso padre que este antes para adoptar al hijo entonces el proceso que lo terminaría adoptando es el DEFAULT.

- **Consideraciones de sincronización**

  Tenemos un sem_create y un sem_open. El sem_create es quien crea al semáforo y el sem_open es quien lo mapea si lo estas abriendo desde otro proceso.

  Luego, tenemos el sem_destroy que solo debe ser llamado por el proceso que crea al semáforo para así liberar al mismo.
- **Consideraciones en test_sync**

  En slowInc cambiamos el uint64_t aux por un int64_t aux pues al debuggear nos dimos cuenta de que se le asignaba a aux un 184467… porque al ser unsigned pega la vuelta y le asigna el mayor número posible (es decir, 2^64 - 1) y consideramos que esto no era correcto para el test.

- **Consideraciones con PVS**

  Nos aparecen los siguientes errores:

1) /root/Userland/SampleCodeModule/Shell/eliminator.c 342 err V779 Unreachable code detected. It is possible that an error is present.

2) /root/Userland/SampleCodeModule/Shell/shell.c 173 err V609 Divide by zero.

En cuanto al error 1), consideramos que es un falso positivo y por ende no lo modificamos.

En cuanto al error 2), este es un “error necesario” por decirlo de una forma pues se necesita hacer una división por cero para que se pueda evaluar la excepción de división por cero (lo cual viene del TP de Arquitectura de Computadoras).

Con respecto a las notas y warnings, consideramos que los que aparecen son falsos positivos o no representan verdaderos inconvenientes en el sistema.

Problemas en el desarrollo:

- **Incorporación del GDB**

  Uno de los problemas que nos encontramos fue el de la incorporación del GDB como herramienta para debuggear el TP. Si bien al final lo pudimos incorporar y utilizar correctamente, fue una de las cosas que nos llevó bastante tiempo.

Instrucciones:

Para compilar el kernel hay que entrar en la carpeta x64barebones y ejecutar el programa **compile.sh**, este asume el nombre del contenedor con la imagen proporcionada por la cátedra, por lo que seguramente haya que modificarlo. Para ejecutarlo solo es necesario correr el programa **run.sh.**

Además, incorporamos la opción de poder compilar poniendo **compile.sh buddy** para que se utilice el buddy manager en lugar del memory manager que implementamos para la primer pre-entrega.

Para ejecutar junto con el GDB, hay que partir desde dos terminales distintas. En una se va a ejecutar el programa run.sh pasandole como argumento gdb **(./run.sh gdb)**. En la otra terminal, es necesario correr el gdb. En nuestro caso, utilizamos el que ya viene en la imagen de docker proporcionada por la cátedra, por lo que solo hay que inicializar y ejecutar el contenedor, posicionarse dentro de root y ejecutar el gdb. Además, se utiliza el ip del guest en el gdbinit, el cual no está incluído en el repositorio.

Para ejecutar los tests desde la shell, se puede usar el comando “help” para ver los nombres de los comandos específicos, se puede usar anteriormente el comando dec para que se pueda visualizar todo su contenido en pantalla al mismo tiempo.

Limitaciones:

1. **Carga del Sistema Operativo con Procesos Elevados**: Al ejecutar el `test_processes` con un número elevado de procesos (por ejemplo, 120), el sistema operativo experimenta una significativa ralentización. Esto se debe a la carga de trabajo excesiva que estos procesos imponen.
2. **Limitaciones de la Shell**: La implementación de nuestra shell se basa en la desarrollada para el proyecto de Arquitectura de Computadoras, donde no se incluyó la funcionalidad de scroll. Esto puede dificultar la verificación de resultados de comandos como `test_priority`, ya que al no poder desplazarse hacia arriba en la terminal, es necesario tomar una captura de pantalla para comparar resultados.
3. **Warning de División por Cero**: Uno de los requisitos en el TP de Arquitectura de Computadoras fue la implementación de un manejo de excepciones para divisiones por cero, lo cual requiere una excepción real de división por cero para testearse. Por ello, este warning se mantiene en el proyecto actual.
4. **Nomenclatura de Funciones y Variables**: En el TP de Arquitectura de Computadoras no se adoptó una convención uniforme para nombrar funciones y variables. Esto ha resultado en una mezcla de estilos, incluyendo

el uso de `nombreFunción` o `NombreVariable`, en lugar de la convención sugerida `nombre_función`.

5. **Visualización de Pipes en Pantalla**: En nuestra terminal, los pipes (|) no se visualizan correctamente, ya que la línea vertical aparece parcialmente fragmentada. Este comportamiento es un efecto heredado del TP de Arquitectura de Computadoras, debido al funcionamiento de la función `putChar`.
6. **Limitación en el Uso de Pipes**: La terminal no permite realizar pipes en los que el comando a la derecha del pipe no tenga una función de lectura, lo cual restringe la funcionalidad de algunos comandos en esta configuración.Además, los pipes permiten conectar solo hasta 2 procesos.
7. **Visualización del help**:

   Al ejecutar el comando **`help`**, la lista de comandos es tan extensa que no todos se muestran en pantalla con el tamaño de letra predeterminado. Para visualizar todos los comandos de manera completa, basta con ingresar **`dec`**, lo cual ajusta la visualización adecuadamente.

8. **Phylo**

   Luego de correrlo por primera vez, el programa muestra bugs.

Conclusión

El desarrollo de este sistema operativo monolítico de 64 bits nos permitió aplicar y profundizar en conceptos clave de administración de memoria, multitarea y comunicación entre procesos. La implementación de un **Buddy System** para la gestión eficiente de memoria y el uso de **Round Robin** con prioridades en el **scheduler** permitieron optimizar el manejo de recursos, mientras que el uso de **pipes** y semáforos libres de **busy waiting** facilitó la sincronización de procesos. La adaptación de funcionalidades como la asignación de **PIDs** y la creación de stacks constantes consolidaron un sistema robusto, y nuestra shell personalizada ofreció una interfaz controlada y funcional. A través de este proyecto, hemos enfrentado y resuelto desafíos técnicos que enriquecieron nuestra comprensión del diseño y

desarrollo de sistemas operativos, culminando en una implementación funcional y ajustada a los requisitos.