Trabajo Práctico N°2 - 3er Entrega

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: 14 de Octubre de 2024

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 (osea 2^64 - 1) y consideramos que esto no era correcto para el test.

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:

El repositorio tiene 2 carpetas principales, x64barebones donde se encuentra el kernel y test_apps dónde está la aplicación del memory manager y los archivos necesarios para el test.

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.**

Para compilar la aplicación con el memory manager primero hay que posicionarse en la carpeta test_apps, 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 ejecutar el test basta con correr ./**mem**.

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.

Para ejecutar los tests desde la shell, se puede usar el comando “help” para ver los nombres de los comandos específicos para correr cada test. Estos se ejecutan en background.

Limitaciones:

Al ejecutar el test_processes, si se le pasa un número de procesos máximos, por ejemplo 120, muy alto ralentiza mucho todo el sistema operativo.

Al compilar el sistema operativo con el flag -Wall se reciben varios warnings debidos a casteos y conversión de tipos de datos, todos relacionados con el dispatcher y llamado de syscalls, esto porque el llamado espera solo argumentos de tipo uint64_t pero las funciones reales que se llaman a partir de las syscalls esperan otros tipos de datos, consideramos que por ser un sistema operativo y para evitar usar un switch, esto es aceptable.

Creamos la implementación de la shell a partir de la shell de Arquitectura de Computadoras donde en esta NO se puede scrollear y por ende puede ser un poco molesto dado que por ejemplo para poder correr el test_priority y poder verificar si

funciona no se puede scrollear para arriba para comparar los resultados del ps. Por lo tanto, nosotros les sacamos una foto a la pantalla y comparamos.