72.11 Sistemas Operativos

TP2 - INFORME

Grupo 22

Santiago Devesa (64223)

Tiziano Fuchinecco (64191)

Tomas Balboa (64237)

Profesores

Alejo Ezequiel Aquili

Ariel Godio

Fernando Gleiser Flores

Guido Matías Mogni

Índice

Resumen y Objetivos

A desarrollar en la versión final del informe.

Compilación del trabajo práctico

1. Ejecutar el comando:

   docker pull agodio/itba-so-multi-platform:3.0
2. Clonar el repositorio:

   git clone git@github.com:smdevesa/TP2_SO.git
3. En el directorio del repositorio, ejecutar el comando:

   docker run -v ${PWD}:/root --security-opt seccomp:unconfined -ti agodio/itba-so-multi-platform:3.0
4. Dentro del contenedor, ejecutar los siguientes comandos:

   cd /root/Toolchain

   make all

   cd ..

   make all
5. Salir del contenedor con el comando:

   exit

Compilación del testeo en aplicación externa

1. Dirigirse a la carpeta *memory_tests* con el comando:

   cd Tests/memory_tests
2. Ejecutar los comandos:

   make all

Ejecución del trabajo práctico

Ejecutar el archivo *run.sh* con el comando: *'./run.sh'* en Linux o macOS.

Si se desea ejecutar con sonido, ejecutar el archivo runSound.sh con el comando: '*./runSound.sh'* en Linux o macOS.

De esta forma es posible ejecutar los testeos del manejo de memoria en el kernel del trabajo práctico.

Una vez ejecutado el comando ./run.sh, escribiendo *'help'*, se listan todos los comandos a ejecutar de la shell.

Test de memory manager

Se puede ejecutar el comando *'tmm X &'* para correr el test de memory manager en background siendo X la cantidad de direcciones de memoria a utilizar para el test. Los tests corren en foreground por defecto, por lo que este test debe ser corrido en background para no perder el procesador para siempre.

Test de context switching y prioridades

Para correr el test_processes se debe ejecutar *'ts X &'* siendo X la cantidad de procesos a crear. Por limitaciones del sistema la cantidad máxima de procesos por default es 25. Por otro lado, para ejecutar el test_priority hay que ejecutar *'tp'*. Los tests corren en foreground por defecto, por lo que test_processes debe ser corrido en background para no perder el procesador para siempre.

Test de sincronización y no sincronización

Para correr el test_sync se ejecuta *'tsy [n] [use_sem]*', siendo n la cantidad de veces que se incrementará o decrementará la variable global *'global'* en el test de sincronización. Por otra parte, el valor de [use_sem] será 1 cuando se desee el uso de semáforos para sincronizar el manejo de la variable global, y 0 en caso contrario. Por defecto el test corre en foreground, aunque se tiene la opción de correrlo en background con el comando *'tsy [n] [use_sem] &*'.

Ejecución del testeo en aplicación externa

Memory_Test

Si se desea ejecutar el testeo de memoria en una aplicación externa al trabajo práctico se debe ejecutar el comando:

./mmTest

Decisiones tomadas durante el desarrollo

Physical Memory Management

En primer lugar, el naive_mm.c actúa como un administrador de memoria muy básico. El código utiliza bloques de tamaño fijo (denominados BLOCK_SIZE), a diferencia de un sistema variable, esto simplifica la administración de la memoria. Más adelante, se implementó la función my_free() para gestionar la liberación de la memoria alocada. Sin embargo, esta implementación no hace un manejo de la fragmentación de la memoria.

Cabe destacar que con el objetivo de no modificar mucho el test provisto por la cátedra, al momento de ejecutar el testeo, no modificamos los parámetros que pide la función Test_mm() pero los ignoramos, ya que nuestra implementación del naive_mm, siempre maneja una cantidad de memoria fija de 256 MB.

Procesos, Context Switching y Scheduling

Para esta segunda entrega parcial, desarrollamos dos archivos en la parte del kernel, processes.c y scheduler.c. Se implementó un Round-Robin con prioridades haciendo que las aplicaciones que requieran mayor prioridad tengan un quantum más largo (para evitar hacer colas multinivel) aunque a posteriori podría mejorarse este sistema. Además, por cuestiones de simplicidad se tomó la decisión de tener una cantidad máxima fija de procesos que por defecto es 32. De esta forma evitamos utilizar memoria dinámica excesiva.

Sincronización

En esta tercera entrega parcial, se creó el archivo semaphore.c dentro de la sección kernel. Se crearon los *spinlocks* con la instrucción de assembler XCHG como se vió en la clase práctica de sincronización con el fin de cuidar las estructuras que se encargan de los semáforos: el arreglo que contiene los structs de semáforos tiene un lock y además cada uno de los semáforos tiene su propio lock interno para asegurar atomicidad. Luego, cada semáforo tiene un nombre y una cola de procesos que están esperando a que se desbloquee. Utilizando este método evitamos la inanición ya que los procesos que se pusieron antes en la “fila” del semáforo recibirán el control del procesador antes que los demás.

Limitaciones

Physical Memory Management

Una de las limitaciones de este primer administrador de memoria corresponde a que el usuario no puede decidir sobre la cantidad de memoria a administrar. Siempre se reservan 131.072 direcciones de memoria.

Por otro lado, el tamaño del bloque de memoria es fijo por lo que si se le demanda un bloque más grande, devuelve NULL y si se le pide uno más chico, devuelve el tamaño de bloque establecido (1024 direcciones de memoria).

Finalmente, este memory manager no chequea si hay dobles liberaciones y tampoco se fija si la dirección enviada es una dirección válida de un bloque, que el memory manager le haya devuelto al usuario.

Procesos, Context Switching y Scheduling

La cantidad máxima de procesos es fija y es por defecto 32. Esta puede cambiarse en tiempo de compilación.

Sincronización

La cantidad máxima de semáforos es fija y por defecto es 64. A su vez, el nombre de los semáforos también está fijo en un tamaño máximo de 64 caracteres. Otra limitación viene relacionada a la cantidad máxima de procesos que se pueden encolar para el uso del semáforo: siguen siendo 32, la máxima cantidad de procesos en ejecución del programa.

Citas de fragmentos de código

Para el armado del naive memory manager, se utilizó como referencia el código hecho en la clase práctica por Ariel Godio, el lunes 16 de septiembre 2024.