Instituto Tecnológico de Buenos Aires

Informe del Trabajo Práctico Nº 2: Kernel
Sistemas Operativos
Grupo 12

Profesores:

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

Integrantes:

Berruti, Román (63533)
Kaneko, Tomás Ken (62297)
Kim, Hwa Pyoung (62129)

Introducción

Este proyecto implementa un kernel con las funcionalidades básicas de un sistema operativo del estilo UNIX. Entre ellas: multitasking preemptivo (context switching) con round-robin (scheduler) y prioridades, manejo de memoria física, sincronización mediante semáforos, comunicación entre procesos con pipes, y una shell interactiva que permite la ejecución de comandos, tanto en foreground como en background.

Instrucciones de compilación y ejecución

Para poder compilar y ejecutar el proyecto, primero debe crearse y ejecutarse un contenedor de docker con la imagen provista por la cátedra dentro de la carpeta x64BareBones-master. Luego, una vez inicializado el contenedor, deben escribirse las siguientes instrucciones:

cd root
cd Toolchain
make clean
make all
cd ..
make clean
Si se quiere usar el Buddy Memory Manager: **make buddy**
Si se quiere usar el Free Array Memory Manager: **make all**

Luego, desde otra terminal sin estar dentro del contenedor de docker y parados en la carpeta x64BareBones-master, debe ejecutarse ./run.sh para finalmente poder correr el proyecto.

Decisiones tomadas y limitaciones

- Al momento de correr el proyecto, pueden escribirse los comandos “SMALLER” o “LARGER” para modificar el tamaño de la fuente. También puede escribirse “COLOR” para que el color de fondo de la terminal sea distinto.
- Para el Memory Manager que no fuera el Buddy, decidimos implementar un Free Array, el cual va reservando memoria de a bloques de tamaño fijo.
- Cada proceso es creado con una prioridad determinada, dentro de 5 opciones que hay para definir la prioridad. A partir de esto, se setean las rondas que debe correr cada proceso según su prioridad para que lo tenga en cuenta el scheduler. Es decir, si un proceso tiene más prioridad, correrá más rondas, hasta que las mismas se terminen y el scheduler deba hacer un context switch con otro proceso.
- Decidimos utilizar estructuras sólidas para implementar nuestras funcionalidades para tener un código más limpio, tales como Memory blocks, sem_t, PCB, etc. Esto nos ayudó a poder mantener una mejor calidad comprensiva.
- Debido a limitaciones dentro de nuestro kernel, decidimos dejar al scheduler bloquear y matar procesos que terminan, permitiéndonos el correcto funcionamiento de la función waitPid.
- Cantidad máxima de:
    - Pipes: 128
    - Tamaño de bloque de memoria (Memory Manager Free Array): 4096
    - Cantidad de bloques (Memory Manager Free Array): 4096
    - Procesos: 128
    - Semáforos: 15
- Direcciones al inicio del Kernel:
    - stackStartAddress: 0x600000
    - startMemoryManager: 0x700000
- Nuestro proyecto no tiene un printf que funcione como el de <stdio.h> en el cual se pueda usar %d o %s para mostrar variables.

Por lo tanto, nuestras funciones con pipe no funcionan como corresponde, sino que utilizan nuevas líneas.

- Para poder correr procesos en background, debe escribirse en la terminal el nombre del proceso a correr, un espacio y luego un "&". Ejemplo: "TESTPRIO &"
- El comando MEM permite consultar el estado de la memoria. Decidimos que según el Memory Manager que se esté usando, imprima la información de una manera distinta para una mejor comprensión. Si se usa el Free Array Memory Manager, se visualizan la cantidad de bloques de memoria usados, libres, etc. En cambio, si se usa el Buddy Memory Manager, se visualizan directamente los bytes de memoria usados, libres, etc.
- Los comandos cat, filter y wc solo funcionan si se los pipea como segundo comando. Es decir, debe ejecutarse por ejemplo "PS|FILTER". El comando wc no funciona correctamente por limitaciones de nuestro trabajo de Arquitectura de Computadoras.
- El comando "PHYLO" que ejecuta el problema de los filósofos comensales no funciona correctamente. Creemos que es un problema de sincronización que no pudimos solucionar.
- Hay un bug que se presenta a veces luego de crear procesos donde no aparece el prompt "$>" al principio de la línea de la shell para escribir los comandos, pero se puede escribir igual el comando correctamente.
- Si la shell luego de crear varios procesos se pone muy lenta, se debe descomentar la línea 64 de "setTickFreq(1000)" del archivo kernel.c. Es la única solución que encontramos dado que el timer Tick interrumpe más veces.
- Cuando se corre "TESTMM", debe escribirse el comando "MEM" en distintas ocasiones para poder consultar el estado de la memoria mientras se ejecuta el test, donde se hacen varios alloc y free. Luego se puede hacer CTRL+C para matar el proceso del test.
- Cuando se corre "TESTPROCESS", debe escribirse el comando "PS" en distintas ocasiones para poder ver los procesos que se van creando y desbloqueando. Luego, debe hacerse KILL seguido del número de PID que tenga el proceso del test. Ejemplo: "KILL 3".

Problemas y soluciones

Durante la implementación de nuestro proyecto, nos encontramos con diferentes problemas.

- Como se mencionó anteriormente, el block y kill para procesos corriendo nos complicó el correcto funcionamiento de nuestra shell, y recurrimos a darle el funcionamiento a nuestro scheduler en la función schedule para cumplir esta función.
- El desarrollo del memory manager buddy nos trajo varias complicaciones. Nos costó entender la lógica para poder implementar correctamente el allocMemory y el free, teniendo que considerar los bloques que debían mergearse en caso de ser necesario. Otro problema fue que inicialmente estábamos usando un tamaño mínimo muy chico para cada bloque, por lo que había procesos que no tenían el espacio suficiente para poder ejecutarse correctamente. También tuvimos problemas para alinear correctamente las direcciones de memoria.
- Nuestro proyecto de Arquitectura de Computadoras no nos permitía hacer funcionalidades que escalaran de forma más simple, como nuestro bufferInterpreter en userShell, el cual ejecutaba los procesos mediante una cadena de ifs con strcmp. Decidimos implementar un array de comandos (el cual estaba implementado en el trabajo práctico de Arquitectura de uno de los integrantes del grupo) para poder facilitar la ejecución de los comandos, permitiéndonos mantener un código más simple y escalable.
- A la hora de implementar nuestro scheduler tuvimos varios problemas que nos llevó al debugueo de nuestro proyecto con gdb, llevándonos a modificar tanto archivos .c como .asm, las cuales no nos permitían correr la shell.

Citas de fragmentos de código reutilizados de otras fuentes

Para nuestro proyecto, indagamos en diferentes sitios web para entender el funcionamiento de cada funcionalidad pedida. Nos guiamos con las presentaciones que se dieron en las clases para desarrollar tanto el scheduler con el armado del stackframe para los procesos como para el desarrollo del memory manager y los filósofos comensales. También, nos guiamos bastante con el foro de la materia, el cual nos permitió solucionar varios problemas que nos surgieron a lo largo del desarrollo del proyecto.