Sistemas Operativos
Trabajo Práctico 2
4 de noviembre de 2019

Construcción del núcleo de un sistema operativo y estructuras de administración de recursos

Nicolas Britos - 59529
Agustín Roca - 59160
Ignacio Grasso - 57610

Índice

1. Introducción

Como indica el título del trabajo práctico, llevamos a cabo la construcción del núcleo de un sistema operativo, implementando un Memory Management, Scheduling y mecanismos de IPC. Como punto de partida utilizamos la entrega final de la materia Arquitectura de Computadoras. Una vez creado el kernel, las aplicaciones muestran el correcto funcionamiento de lo anteriormente mencionado.

2. Implementaciones

2.1. Procesos

Un proceso, dentro de este Sistema Operativo, está definido por los siguientes atributos:

- Pid(Process id): número identificatorio, no existen dos procesos con el mismo pid.
- Ppid(Parent process id): número identificatorio del padre del proceso.
- Name: nombre del proceso.
- State: estado del proceso. Puede ser Ready, Running, Dead, Locked
- StackPointer: puntero al stack propio del proceso.
- ProcessMemoryLowerAddress: puntero al inicio del bloque de memoria alocado para el proceso.

2.1.1. Instancia de un proceso

Al instanciar un proceso, se crea una estructura donde se almacenarán los atributos mencionados anteriormente propios de dicho proceso. Se reserva

espacio y se inicializa el stack con lo que luego serán los valores que tomarán los registros al correrse el proceso.

2.1.2. Manejador de procesos

Para evitar mucho acoplamiento entre los procesos y sus funciones y el scheduler se decidió realizar un intermediario entre ellos. De esta manera los procesos no dependen del scheduler ni viceversa. Aun así se decidió que usaran la misma definición de la estructura de procesos para simplificar el código. Básicamente, al llamar a un proceso a través de una syscall primero se pasa por el manejador de procesos y este le pasa toda la información necesaria y ejecuta las funciones correspondientes de procesos y scheduler.

2.1.3. Context Switching

Este sistema operativo permite ejecutar varios procesos al mismo tiempo. Cuando el scheduler decide que es el momento de cambiar de proceso, se guarda el contexto del proceso que está siendo desalojado en su propio stack. Al obtener el siguiente proceso a ejecutarse, se desapilan los registros y se restaura el contexto para que este pueda ser ejecutado. Cuando la interrupción finaliza, el RIP apunta a la próxima instrucción a ejecutarse del próximo proceso.

2.1.4. Scheduler

Como mencionamos anteriormente, el scheduler tiene la tarea de seleccionar quién será el próximo proceso a ser ejecutado. Para llevar a cabo esta selección se implementó una cola, donde el próximo proceso a ejecutarse, es el próximo en la cola. El scheduler

devuelve el puntero al stack del siguiente proceso así es posible realizar el context switching. Con esto evitamos la monopolización del uso de la CPU y lograr un equitativo reparto de esta. Cada proceso tiene definida una prioridad: LOW o HIGH. Dependiendo del valor, es por cuanto será multiplicado el quantum y una vez que paso el tiempo, la cambia.

2.2. Manejo de memoria

El manejo de memoria fue implementado de dos maneras distintas. Según como sea compilado el programa, va a ser la opción seleccionada; por default está el Buddy.
Para la implementación del Buddy, definimos una estructura Node, que tendrá entre otros atributos importantes, un puntero al nodo inferior derecho y otro al nodo inferior izquierdo. Con esta estructura definida, podemos manejar la memoria como si fuese un árbol y así poder ir partiendo a la mitad los bloques de memoria y obtener el tamaño correspondiente. Justamente por esto de ir dividiendo los bloques a la mitad, cada bloque será siempre del tamaño de una potencia de 2.

Para la implementación de la Free List, diseñamos una lista encabezada con un puntero al primer nodo y otro puntero al último. Cada nodo tiene un puntero al anterior, al próximo y a su información. Se definió una tabla donde cada campo tiene un puntero a la dirección de memoria y un boolean indicando si está reservada o no.

2.3. IPC

Con respecto a la sincronización y comunicación entre procesos implementamos semáforos y pipes, ambos nombrados. Los semáforos se implementaron de manera

similar a como se encuentra especificado en POSIX: cuando su valor es 0 (o negativo) se bloquean todos los procesos que realizan un sem_wait utilizando una cola (FIFO) como estructura para almacenarlos. Cuando este valor es negativo el mismo equivale a la cantidad de procesos en espera.

Respecto a los pipes, estos son unidireccionales y bloquean tanto en escritura como en lectura si es que no hay nada para leer o espacio en el buffer para escribir.

3. Instrucciones de compilacion y ejecucion:

Dentro de la imagen de Docker otorgada por la Cátedra, entrando usando socat y xquartz cómo se nos fue explicado para usar el display, ubicarse en el directorio del proyecto y ejecutar el comando: “make all” o “make buddy” (estas dos son sinónimos y al ejecutarse se usará buddy) o “make freeList” (para usar freeList) para compilar el proyecto.
Luego para ejecutar se deberá realizar el comando “./run.sh” (o “docker.sh”. Hacen lo mismo).

4. Limitaciones

Se pueden ejecutar hasta 20 procesos y cada uno de estos puede tener asociados hasta 20 file descriptors (un pipe se representa con un file descriptor).
No se pudo llegar a implementar el pipe via shell con el ‘|’. Aun así cuando se corre procesos entre ‘|’ se corren de izquierda a derecha en orden (lo que sería equivalente al ‘;’ en Linux). Dejamos esto para mostrar que logramos hacer el parseo correctamente aunque no se pudo implementar los pipes. El ‘&’ si está implementado y es para forzar a que un proceso corra en background salvo por “nice”, “block” y “clear” que siempre corren en foreground. También dejamos que “loop” siempre se corra en background ya que correrlo en foreground bloquearía la terminal.

Si se deseara hacer que loop bloquee la terminal si se corre sin ‘&’ solo habria que sacar el “= S_M_BACKGROUND” de la línea 88 de TPE/Userland/SampleCoreModule/shell.c .

5. Problemas encontrados

Los dos primeros problemas que tuvimos cuyo origen exceden este trabajo fueron que la actualización de Mac OS Catalina trajo muchos problemas con qemu. Y el otro se debió a que dos de los tres integrantes del grupo no habían realizado la modalidad del final de Arquitectura de Computadoras del sistema operativo sino que la de Galileo, entonces se tuvieron que tomar algunos días para entender cómo funcionaba.
Uno de los problemas que tuvimos al poder encarar por primera vez este trabajo ya resuelto lo anterior fue como organizar las syscalls. Como la base de este TP fue la entrega final de Arquitectura de Computadoras, el manejo de las syscall no existía, ya que al ser tan pocas, no era necesario tener un handler para ellas. Así que decidimos implementar un handler, asignandole a cada syscall un ID, que representa una posición específica en un vector de punteros a Syscalls.
Otro problema fue en la creación de procesos. Inicialmente le asignabamos la dirección donde arrancaba la función al RIP. Al hablar con el profesor y entender los problemas que esto ocasionaba, decidimos implementar una wrapper.
Con respecto al scheduler y el dispatcher (encargado de intercambiar procesos) tuvimos muchos problemas ya que no siempre cargaba bien los registros y ejecutaba cualquier instrucción. Esto fue arreglado cambiando el orden en que se guardan los registros y tomando en cuenta el manual de Intel de x86-64 para ver cuales son los registros que se pushean al stack y en qué orden al llegar una interrupción.

Como último paso antes de la entrega de este TP, utilizamos CPPCHECK y PVS STUDIO para encontrar bugs que

normalmente el compilador no detecta. Ambas herramientas nos ayudaron a arreglar varias errores, pero tambien nos arrojaron warnings que nos exceden y no podemos resolver (warnings y estilos sobre código que no escribimos nosotros).
Por un lado, CPPCHECK nos indico de varias funciones que nunca son llamadas pero preferimos dejarlas igualmente, además de que hay algunas que son llamadas por código Assembler que cppcheck no tiene en cuenta. También señaló una parte del codigo de memManager como que nunca sera ejecutada y es porque este no tiene en cuenta los IFNDEF. Hay otros warnings también que vienen desde Pure64 en sí que preferimos no tocar.

Por el otro lado, PVS STUDIO indico algunas cosas más, como por ejemplo, conversiones de un entero a un puntero, que esto debe ser resuelto así porque son posiciones de memoria fijas por Pure64; una asignación hecha dos veces seguidas pero que tiene un sleep en el medio (para phylo). Además, hay warnings de que no se realiza “unlockProcess” en ipc.c, pero eso está hecho de manera intencional y por último dos loops infinitos que también están hechos de manera intencional (en phylo y en shell). Hay otros warnings también que vienen desde Pure64 en sí que preferimos no tocar.