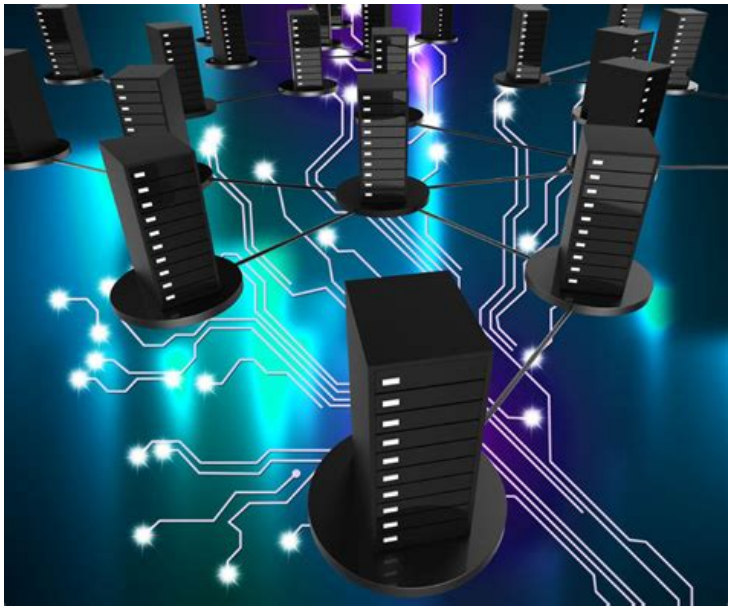
Sistemas Operativos
TP2: Construcción del Núcleo de un Sistema Operativo y estructuras de administración de recursos.

Informe - TP2 Sistemas Operativos



Barmasch, Juan Martín (61033),

Bellver, Ezequiel (61268),

Lo Coco, Santiago (61301)

01/11/2021

ITBA

TABLA DE CONTENIDOS

INTRODUCCIÓN

El presente informe detalla las decisiones tomadas por el grupo a la hora de resolver el Trabajo Práctico n° 2 de la materia 72.11 Sistemas Operativos. El mismo consistió en crear un kernel simple basándose en el TP final de la materia 72.08 Arquitectura de Computadoras. Para ello, se implementaron Memory Management, procesos, scheduling, mecanismos de IPC y sincronización.

DESARROLLO

Memory Management

Acorde a lo solicitado por la cátedra, se implementaron 2 tipos de Memory Manager distintos (ambos con 512 MiB de memoria total): uno que use el algoritmo “Buddy” de alocación de memoria y uno a elección del grupo. Para este último, se optó por una implementación que emplea un “first fit algorithm”, es decir que busca el primer bloque de memoria lo suficientemente grande para alocar la memoria solicitada y lo utiliza. A su vez, esta implementación incluye un algoritmo de coalescencia, que agrupa los bloques de memoria libre contiguos que pudieran generarse tras la liberación de memoria en un solo bloque de mayor tamaño.

En cuanto a la implementación que utiliza el algoritmo “Buddy”, cabe aclarar que el grupo optó por un bloque de memoria con tamaño mínimo de 4096 (0x1000) bytes, correspondiente al llamado *level* 17. Se optó por este valor dado que se consideró lo suficientemente grande como para no producir overheading, pero aún siendo lo suficientemente pequeño para minimizar el desperdicio de memoria en las alocaciones de los bloques mínimos. Dada la simplicidad de este Kernel, tal vez ese tamaño pudo haber sido menor. De todas formas, en nuestro caso la memoria no es un recurso escaso.

Cabe aclarar que estos dos tipos de Memory Manager son seleccionados al momento de compilación, no pudiéndose usar ambos al mismo tiempo (dado que no tendría sentido). Para

más información, ver instrucciones de compilación y ejecución.

Scheduling, Procesos y Context Switching

El algoritmo de Scheduling emplea un Round Robin que admite prioridades. El mismo fue implementado con un único array para todos los procesos, a los cuales se les consulta el estado (bloqueado, listo, etc.) antes de seleccionar un nuevo proceso para ejecutar. Las prioridades fueron implementadas almacenando la cantidad de ejecuciones consecutivas que tuvo un proceso y estableciendo un máximo para este valor que depende de la prioridad del mismo (siendo este límite mayor para los procesos de con más prioridad).

Los procesos fueron implementados como una estructura que almacena toda su información relevante (nombre, pid, estado, file descriptors de entrada y salida, prioridad, etc.). Se implementaron 5 estados posibles para un proceso: READY, DEAD, BLOCKED, WAITING, BLOCKEDIO. Si un proceso se encuentra bloqueado, no podrá contar con tiempo de CPU. Si está muerto será liberado y eliminado de la cola de listos cuando sea su "turno" de correr. Si un proceso está esperando, no correrá hasta que todos sus hijos hayan terminado de correr (mueran).

En cuanto a las prioridades de los procesos, existen 40 niveles para las mismas (tal como lo implementa Linux para los procesos que no son de Real Time). Las mismas pueden ser modificadas mediante el comando *nice*, el cual recibe un offset con respecto a la prioridad media, pudiendo este offset tomar valores enteros entre -20 y 19.

DEMOSTRACIÓN DEL FUNCIONAMIENTO DE LOS REQUERIMIENTOS

El funcionamiento de las aplicaciones de User Space puede ser comprobado ejecutándolas, para lo cual basta con escribir su nombre desde la shell y presionar enter.

Se cuenta con el comando *help* el cual recibe el número de página empezando en 1 para luego imprimir los comandos disponibles y una breve descripción de cada uno. Entre los comandos disponibles se encuentran los *test* provistos por la cátedra.

Cabe aclarar que el comando *loop* no sirve para hacer un *test* del estilo de *test_prio*

debido a que el mismo está implementado con un *sleep* que manda a dormir el proceso y lo despierta cuando pasó el tiempo predefinido.

A la hora de utilizar comandos como el *cat*, *wc* o *filter* si se corren sin pipe, se quedan leyendo de entrada estándar hasta que se le mande un EOF, el mismo se manda usando la combinación de teclas Alt+F1.

CAMBIOS REALIZADOS A LOS TESTS

En el *test_sync* se hizo que el semáforo fuera inicializado fuera de la función *inc* para poder realizar luego un *semClose* debido a que esta función libera la posición de memoria y lo borra de la lista de semáforos.

Por otro lado, en el *test_processes*, agregamos un *quitCpu* al principio del *while* para que termine de limpiar los recursos correctamente. Esto es pues en nuestra implementación no liberamos todos los recursos cuando se elimina un proceso sino cuando el *scheduler* le asigne tiempo de *CPU* como se mencionó previamente.

Los demás cambios a los tests fueron meramente de portabilidad.

INSTRUCCIONES DE COMPILACIÓN Y EJECUCIÓN

Se encuentran detalladas en el archivo *README.md*

CÓDIGO REUTILIZADO DE OTRAS FUENTES

Se utilizó código de

https://github.com/Infineon/freertos/blob/master/Source/portable/MemMang/heap_4.c

como base para implementar el Memory Manager elegido por el grupo y de

https://github.com/cloudwu/buddy/blob/master/buddy.c

como base para implementar el Buddy. También hay métodos aislados que fueron

tomados de repositorios open source, como es el caso de la función *ftoa()* definida en el archivo *libc.c* de Userland. Para todos estos casos, el link al repositorio se encuentra comentado en el código. Todo el código restante es de nuestra autoría, dejando de lado el aportado por la cátedra de 72.08 Arquitectura de Computadoras como base del proyecto final de esa materia.

NOTA

Respecto a la entrega del 1/11:

- Se agregó PVS-Studio y cppcheck en la rama master (pese a que antes se había hecho pero en otra rama que quedó desactualizada).
- Se agregaron los dump en los dos memory manager.
- Se corrigieron los warnings de compilación (+ los de PVS-Studio y cppcheck).
- Se agregaron 2 líneas en el sem.c de Kernel inicializando *sem->entering* y *sem->last* cómo NULL. La falta de estas instrucciones provocaba fallas a la hora de ejecutar comandos del tipo "*cat* | *cat*".
- Se agregaron las nuevas funcionalidades al comando *help* debido a que este estaba desactualizado (con los comandos de Arquitectura de las Computadoras).
- Se especificó que el 2do comando corrido después de un *pipe* en la *shell* se ejecutara en *background*.
- Se guardó el puntero obtenido mediante un malloc en la estructura del proceso. Esto se hizo con el propósito de realizar un correcto free luego (antes realizábamos una aritmética con una macro pero no devolvía valores correctos).
- Se modificó el *test_processes* agregándole un *quitCpu* para su correcto funcionamiento.