Sistemas Operativos

TP3 - Estructuras de administración de recursos

Primer Cuatrimestre de 2019

Grupo 8

Integrantes

- Karpovich, Lucia 58131
- Martin, Fernando 57025
- Reyes, Santiago 58148
- Tarradellas, Manuel 58091

Introducción

El trabajo consiste en la implementación de estructuras sobre el trabajo práctico 2 para agregarle mejor administración de recursos al sistema.

Physical Memory Manager*

Para el *Memory Manager* se utilizó una implementación del algoritmo *Buddy Memory Allocation*. La memoria se divide en diferentes bloques o particiones las cuales llamaremos *ListNodes*.

La estructura que contiene la información de los nodos tiene punteros al siguiente y anterior para manejo de las listas, un puntero al *buddy* de la derecha si es que él es un nodo izquierdo, un puntero al padre (también si es que es nodo de izquierda), nivel en el que se encuentra, disponibilidad, dirección de memoria de su bloque, entre otras cosas.

Cada proceso se asigna de acuerdo con el requerimiento de espacio. A éste le asigna la primera partición lo más pequeña posible (que cumple con los requerimientos de espacio pedidos). Para poder cumplir esto, el sistema maneja un árbol binario con bloques de memoria donde la altura del árbol indica un nivel. El nivel más bajo (0 o representado en el código con él define BIGGEST_SIZE_LEVEL), tiene un único nodo que representa la partición más grande de la memoria, es decir, su total. El nivel más alto (13 o representado en el código con él define SMALLEST_SIZE_LEVEL), contiene los nodos de tamaño mínimo (2^7).

A su vez, el sistema posee un arreglo de 14 espacios donde cada uno apunta a un nivel del árbol para fácil acceso. El árbol contiene nodos en uso y nodos libres donde solo las hojas se encuentran disponibles para usar ya que si un padre tiene hijos, ellos representan las dos mitades del padre.

Si dos nodos hermanos (que llamaremos *buddies*) se encuentran libres simultáneamente, serán unidos en lo que resulta ser el padre. Es decir, se elimina a ambos hermanos y se libera al padre. De esta forma no queda tan fraccionada la memoria.

***Nota:** ver problemas encontrados.

Pipes

Se implementaron *unnamed pipes* en el sistema que operan de manera similar a como lo hace Linux.

Cada proceso se crea con un vector de *file descriptors* estático. En las posiciones 0 y 1 se encuentran los enteros ‘0’ y ‘1’ que representan entrada y salida estándar, el resto se encuentran inicializados en ‘-1’.

La creación de un pipe implica crear un vector de dos posiciones por el proceso. La función pipe recibe dicho vector y crea una estructura *tpipe,* cada pipe tiene un id único. El pipe

se agrega a una lista de pipes existentes y carga su id en las primeras dos posiciones libres del proceso que hizo la llamada a *pipe().* La función devuelve en el vector recibido por parámetro, los índices de donde se guardó el pipe en su vector de descriptores.

Las funciones de escritura y lectura reciben *file descriptors*. Estas utilizan estos índices para acceder al vector de *file descriptors* del proceso actual e imprimir o leer de ahí.

Para el correcto funcionamiento de pipes se implementó la función *dup().* Para esto se tuvo que modificar la forma en que se crean los procesos. Anteriormente un proceso era creado y ejecutado instantáneamente. Actualmente este funcionamiento se dividió en tres funciones separadas: la creación y ejecución inmediata, la creación y no ejecución (se obtiene su *pid* para ejecutarlo posteriormente), y la ejecución de un proceso ya creado mediante su *pid*).

Para hacer un pipe entre dos procesos, un proceso debe crear un pipe de la forma comentada anteriormente y hacer una llamada a *dup* pasándole el *pid* del proceso al quiere conectarse, un extremo del pipe y una posición del vector de descriptores del proceso a conectase donde se va a posicionar el extremo de pipe mencionado.

Para el intercambio de información, la estructura *tpipe* tiene una zona de memoria dinámica circular, un puntero de escritura (se actualiza cada vez que se escribe sobre la memoria) y un puntero de lectura (análogo). La función de lectura se bloquea si no hay nada para leer.

Mejoras al scheduler

Para habilitar multiprocesos se había implementado para el TP2 un sistema *Scheduler* que maneja una lista de procesos a ejecutar mediante un sistema de lotería, el cual ya permitía manejo de prioridades. Los procesos se encuentran representados en una lista de nodos que llamamos *tPList,* los cuales contienen información sobre cada proceso como su estatus, prioridad, stack pointer, etc.

Ante la creación de un proceso se le asigna una cantidad de “tickets” para la lotería y la cantidad dependerá de su prioridad (un proceso con alta prioridad recibirá muchos tickets lo cual significa más chances de ser seleccionado para ejecutarse ante un sorteo). El proceso ganador del sorteo es el que se ejecutará, eligiéndolo de la lista de procesos listos para correr.

Por definición del algoritmo se evita la inanición dado que todo proceso recibe tickets para el sorteo (por más baja cantidad que sea), y como el sorteo se realiza de manera aleatoria, dicho proceso siempre tiene la posibilidad de correr eventualmente.

Se implementó el comando *nice* para el usuario para que este pueda modificar la prioridad de un proceso. La *shell* deberá recibir el siguiente comando para hacer uso de esta función:

$> nice PID PRIORITY

Donde:

- PID es un número válido correspondiente al proceso a modificar.
- PRIORITY es una de las siguientes opciones: HIGHP, MIDP, LOWP.

Filósofos comensales

Para realizar la simulación de los filósofos comensales incorporamos un semáforo que representa la cantidad de filósofos que pueden acceder a la mesa donde se come.

En cualquier momento dado, hay la misma cantidad de filósofos que pares de palitos chinos, si restringimos el acceso a la mesa para que haya la cantidad total de filósofos menos uno, nunca se producirá un *deadlock* ya que siempre algún filósofo podrá comer.

Cuando finalice, dejará los dos palitos que utilizó y se irá de la mesa para ponerse a pensar (el sushi es bueno para filosofar según la gente). Esto permite que si un filósofo quedó afuera de la mesa, pueda ingresar para esperar por su turno para utilizar unos palitos.

Cada vez que nace un filósofo, se aumenta la cantidad de palitos chinos (presentes en forma de semáforos) y los lugares en la mesa. Al morir, estas cantidades disminuyen.

Aplicaciones de User space

pipes: el comando *producer* (imprime del 0 al 9 en líneas separadas) y el comando *consumer* (consume strings y los imprime) pueden ser usados en conjunto para demostrar el funcionamiento de pipes de la siguiente manera:

$> producer \ consumer

ps: muestra la lista de procesos con sus propiedades, PID, nombre, estado, foreground, memoria reservada, prioridad, etc.

prodcons: muestra una resolución para el problema productor consumidor de buffer acotado, se debe poder incrementar / decrementar en runtime la cantidad de consumidores y productores (hasta 5). Muestra el funcionamiento de mutex y semáforos.

memtest: Se implementó una función que pide y libera memoria para demostrar el funcionamiento del manejo de memoria.

ptest: muestra el funcionamiento del scheduler mostrando procesos en simultáneo que pueden ser terminados con el comando killtest.

nice: (ver seccion del scheduler).

philosophers: simula el problema de los filósofos comensales de manera interactiva.

Instrucciones de compilación y ejecución

Requisitos: Tener instalado docker (versión 1.8 o posterior) y quemu.

Para el correcto compilado y ejecución del programa, se recomienda ingresar el comando dado por terminal, con todos los archivos en el mismo directorio:

$> cd src

$> sudo ./docker.sh

Limitaciones

El sistema tiene un límite de memoria a manejar (dado por una variable MEM_SIZE) y una cantidad máxima de particiones de la memoria (dado por la variable MAX_NUM_NODES).

Hay un máximo para la longitud del id de un mutex dado por una variable llamada MAX_MUTEX_ID y lo mismo para el nombre de los semáforos.

El driver de teclado no tenía implementado el carácter '|' así que fue reemplazado con el carácter '\'.

El vector de *file descriptors* de cada proceso tiene un máximo de MAX_FD posiciones.

Problemas Encontrados

El scheduler no generó grandes problemas dado que se implementaron cambios no muy significativos desde el sistema anterior generado para el TP2.

Nuevamente resultó complejo la utilización de tests dado que en el TP todo se encuentra muy entrelazado, de modo que probar y arreglar el funcionamiento de una función o parte del proyecto de manera unitaria resultaba imposible porque algunas cosas solo funcionan cuando son ejecutadas en el contexto del trabajo práctico. En otras palabras, se hizo muy poco unit testing.

El testeo de la memoria se hizo dificultoso dado que si bien se podía ir probando de a unidades, una vez que se incorpora al resto del TP, se vuelve todo muy empalmado y resulta complicado separar los errores por partes. El manejo de memoria sufrió cambios significativos y estos fueron implementados en simultáneo con cambios como pipes, de manera que se tuvo que unir todo hacia el final. Los tests que se hicieron anteriores al merge tuvieron el error de no reflejar la situación final de modo que con poco tiempo para la entrega surgieron errores que no se llegaron a corregir.

El principal error de el sistema es un fallo de memoria. Bajo ciertas circumstancias, el recorrido de una de las listas (en la funcion levelHasAvailable del archivo memoryManager.c) de nodos se vuelve cíclica y nunca finaliza. Debido a que llegamos sin tiempo a esta conclusión

decidimos hacer la entrega de todas formas. El correcto funcionamiento de las demás aplicaciones del sistema se puede comprobar simplemente usando el manejo de memoria del TP2*, ya que ese no genera fallas. De todas formas, el funcionamiento parcial del manejo de memoria nuevo se puede ver corriendo el tp con la memoria nueva (se podra observar el error llamando a ps) o en vez de ejecutar la shell, ejecutando la funcion testMem()* del kernel.c.

* Para facilitar este cambio, dejamos ambos códigos en el .c y .h del memory manager con el codigo del TP2 comentado.

Fuentes

Algoritmo Buddy Memory Allocation

https://www.memorymanagement.org/mmref/alloc.html

https://www.geeksforgeeks.org/operating-system-buddy-system-memory-allocation-technique/

Filosofos

https://www.studytonight.com/operating-system/dining-philosophers-problem

https://pages.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html

https://medium.com/science-journal/the-dining-philosophers-problem-fded861c37ed