Alumnos: Beade, Gonzalo – Castagnino, Salvador – Hadad, Santiago
Profesores: Godio, Ariel – Aquili, Alejo – Mogni, Guido – Merovich, Horacio
Materia: Sistemas Operativos
Trabajo Práctico Especial: Pure64

Introducción

El objetivo del siguiente trabajo es concretar los temas vistos a lo largo de la materia Sistemas Operativos. Se implementó un sistema que simula el modo protegido de Intel, pues si bien el procesador se encuentra prendido en modo real, se produce una jerarquización de manera tal que pueden distinguirse dos niveles de privilegio : kernel y usuario. El principal objetivo es negarle al usuario un acceso directo al hardware, y esto se logra brindando una api para que el mismo pueda interactuar con los recursos, pero siempre con el kernel de por medio, logrando una protección del sistema.

La idea del proyecto es implementar funcionalidades de un Sistema Operativo, entre ellas manejo de memoria física, procesos, context switching, scheduling, sincronización y mecanismos de comunicación entre procesos.

Manejo de memoria física

La necesidad de que cada proceso pueda obtener sus recursos de manera dinámica justifica la implementación de un Memory manager en nuestro kernel. El Memory Manager brinda la posibilidad de que cada proceso “pida” memoria al sistema para guardar información, pero a su vez le brinda la posibilidad al kernel de crear los procesos , pues necesita guardar de ellos su información y asignarles un stack.

Se proveen dos implementaciones distintas: una que consta de una lista de bloques de memoria libres y una implementación muy básica del buddy Memory Manager. Ambas implementaciones tienen sus ventajas y desventajas.

Por una parte el buddy internamente al liberar un bloque de memoria llama a una función que se encarga de juntar (coalescing) los bloques que pertenecen a un mismo nivel del árbol y tienen al mismo padre (buddy blocks), por lo tanto si se reservan 2 kb, y luego 2kb , al liberar ambos bloques, se puede asignar 4kb. Dicha secuencia de operaciones en la otra implementación, produciría dos bloques de 2 kb que pueden alojarse, pero no se podría reservar 4kb en un solo pedido. Por otro lado, al contar con una implementación básica del buddy se cuenta con las siguientes desventajas: 1)Se ocupa una gran cantidad de memoria para las estructuras subyacentes que permiten la correcta operación del mismo. 2)Trae aparejado con su utilización una gran cantidad de fragmentación interna, pues al poderse reservar únicamente bytes en cada llamada, si lo que pide un proceso no cumple con dicho requerimiento se redondea para arriba. Esto puede verse en el sistema si se ejecuta el comando “testMM”: Si se ha compilado con el listfreeMM , entonces al ejecutar el comando “mem”, se verá que el sistema tiene todavía una gran cantidad de memoria disponible y no siente tanto el impacto de ejecutar dicho comando en background. Por otra parte cambiando únicamente la compilación para que se realice con el buddy, la misma secuencia de comandos dará como resultado un sistema notablemente reducido en cuanto a memoria disponible. Asignarle una mayor cantidad de bytes al buddy, por ejemplo 2Mb, podría pensarse que es la solución a dicho problema, sin embargo el malloc y el free son O(n) (temporal y espacial) donde n es la cantidad de memoria que maneja, por lo tanto no se trata de una implementación que escale a un sistema operativo que se encuentre en el orden de los GB, en ninguno de los dos casos.

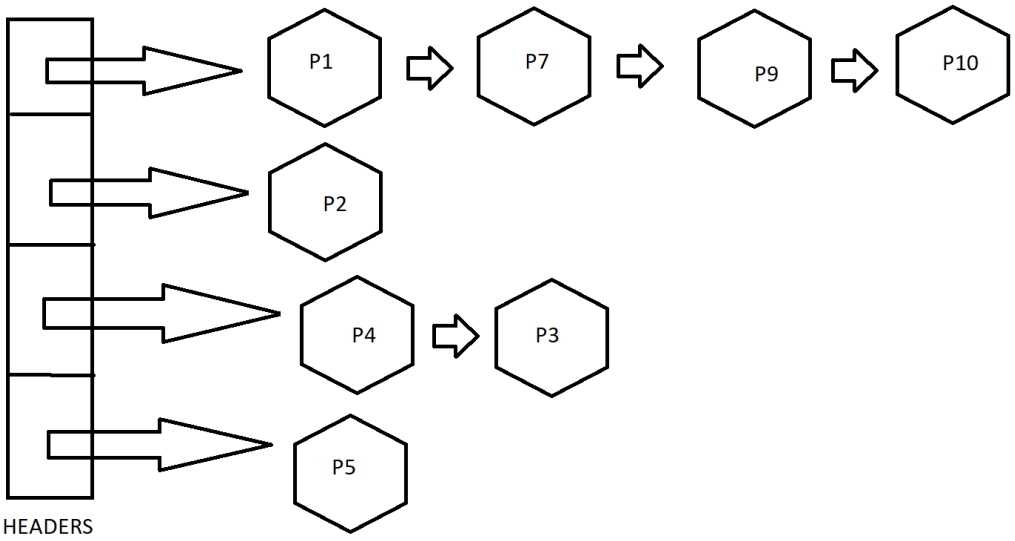
La implementación de ListFreeMemoryManager fue basada en el repositorio: https://github.com/Infineon/freertos/blob/master/Source/portable/MemMang/heap_2.c y su integracion en el sistama no represento un gran problema. Mientras que la implementación from scratch del buddy sí

representó un desafío. El principal problema fue en el free, al notar que se estaba pisando memoria en el sistema, se agregó a testMM un for que recorre los punteros entregados viendo si en algún momento se entregaban dos punteros iguales. Sin embargo el problema no era ese, si no que el free funcionaba mal, pues en nuestra implementación un puntero no está relacionado con un solo bloque, entonces surgió la necesidad de que el malloc haga esa asociación para evitar problemas en el free.

Procesos, cambio de contexto y planificación

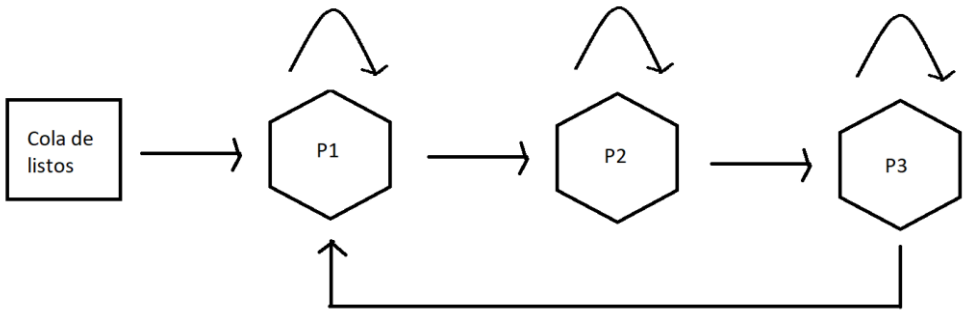
El manejo de procesos está implementado en dos módulos, process.h y roundRobin.h.

Process.h contiene métodos de creación de procesos nuevos en memoria, obtención de data de los procesos, exit y espera. Lo interesante de la implementación se encuentra, sin embargo, en el archivo de Round Robin. El sistema trabaja con una colección de listas simplemente encadenadas. Cada lista simplemente encadenada está asociada a una posición en un vector headers.



El primer elemento del vector (posición 0) es la cola de listos. La función nextTask() del roundRobin es llamada en cada interrupción del timerTick, momento en el que se decide, según la prioridad del proceso, si se debe avanzar al próximo en la lista o no.

El siguiente gráfico muestra la esencia sobre cómo decide nextTask() el próximo proceso a ejecutar. nextTask() puede elegir no avanzar al próximo proceso, entrando en un lazo. La cantidad de veces que nextTask() entra en un lazo para un proceso P es WORSTPRIORITY - P.Priority + 1, donde WORSTPRIORITY es la peor prioridad que puede asignar nuestro SO. Notemos como si P.Priority = 0, nextTask() selecciona al mismo proceso la máxima cantidad de veces, simbolizando la máxima prioridad. Si P.Priority = WORSTPRIORITY, nextTask() solo corre el proceso una sola vez antes de pasar al próximo.



El resto de los elementos del vector representan colas de bloqueados. Trabajamos con contrato tácito de bloqueos por contraseña. En otras palabras, un proceso puede bloquearse por “clase” y cada “clase” de bloqueo tiene asociadas posiciones en el vector. Para bloquear un proceso, se le debe dar una contraseña.

Esa contraseña se traduce en una posición del vector. Si un proceso está bloqueado por una contraseña en particular, no puede ser desbloqueado por otra.

- La posición 1 del vector representa el bloqueo por IO.
- Las posición 2 del vector representa el bloqueo por syscall. Es decir, un bloqueo que es realizado voluntariamente desde userland utilizando el comando kill.
- Las posiciones 3-29 representan los bloqueos por semáforo.
- Las posiciones 30-MAXBLOCKTYPES representan los bloqueos por FIFOs.

Un proceso puede ser desbloqueado explícitamente según su PID. Un caso de esto es utilizando el comando kill desde la consola, en el cual se introduce un proceso bloqueado específico y ese mismo proceso es liberado. Sin embargo, a veces esto no tiene sentido. Un ejemplo de esto es en las colas de procesos bloqueados por un semáforo en particular. En ese caso, el desbloqueo no se realiza por PID, sino que se consulta al banco procesos bloqueados por esa contraseña y se desbloquea al primero disponible según FIFO.

En una primera implementación, usamos desbloqueo LIFO, pero esto podría generar inanición. Nada garantiza que el proceso en el fondo de la lista alguna vez salga de ahí.

Cada vez que se bloquea un proceso, se analiza si la cola de listos está vacía. Si la cola de los listos está vacía, se inyecta un proceso IDLE, el cual hace hlt eternamente. Al agregar o desbloquear un proceso, este proceso IDLE se resguarda para próximo uso, pero no queda corriendo.

Al desarrollar el problema de los filósofos, nos dimos cuenta de un problema. Existe la posibilidad de que un proceso acapare un semáforo mutex, y que se lo pueda matar en ese preciso instante, antes de devolver el semáforo. En este caso, todos los otros procesos que estaban esperando a ese semáforo quedan bloqueados para siempre, esperando a un semáforo que nunca se va a habilitar. Deberíamos revisar que cuando se mate a un proceso, este no haya acaparado un semáforo, pero por tema de tiempo y preservar la integridad del código decidimos no hacerlo.

Sincronización

Debido a que nuestro sistema es un multitarea apropiativo, resulta fundamental brindarle a los procesos la posibilidad de colaborar entre ellos. Al igual que en POSIX nuestro sistema cuenta con la opción de que dos procesos distintos se comuniquen a través de “pipes”. Esta colaboración es imposible en un entorno concurrente de ejecución sin un mecanismo de sincronización.

Es justamente por ello que nuestro sistema provee una API para que los procesos puedan sincronizarse a través del uso de semáforos. Cada semáforo en concreto se accede con una clave del tipo char * , y aquel proceso que tenga dicha clave a disposición puede abrir el semáforo para su utilización , y luego cerrarlo para permitir la liberación de recursos.Cada semáforo tiene la capacidad de bloquear y desbloquear procesos, operaciones muy sensibles de cualquier sistema , es por eso que cada operación se realiza a través de una syscall, protegiendo la integridad del kernel.

Otro aspecto a destacar es que se implementaron la operaciones wait y post de manera atómica , pese a que no exista la posibilidad de que al estar ejecutando en nombre del kernel un proceso sea interrumpido (en esta arquitectura), es correcto hacerlo de esta manera pensando que en algún momento podría ejecutarse en un entorno de multiprocesamiento simétrico. Dicha atomicidad se logró mediante la implementación de spinlocks antes del acceso a los valores actuales del semáforo.

Una limitación de nuestra implementación es que con tener la clave del semáforo se pueden realizar cualquiera de las operaciones permitidas, incluso sin haber abierto el semáforo. Esto puede solucionarse

trivialmente, pues cada semáforo podría guardar la información de los procesos que lo han abierto. Por otra parte, si un proceso por error de quien lo programa no cierra el semáforo que ha abierto, el sistema nunca podrá liberar dicha información, lo cual lleva a una pérdida de performance.Dicho problema no presenta un gran desafío tampoco en cuanto a la implementación, pues periódicamente se podría ver cuáles procesos que abrieron el semáforo no han finalizado aún.

Durante la implementación se encontró principalmente una dificultad: El código que sirvió como fuente para la implementación de este mecanismo no contemplaba una posibilidad. La implementación original era la siguiente:

```
while(loop){
    acquire(&currentSem->lock);
    if(currentSem->value > 0 ){
        currentSem->value--;
        loop=0;
    }else{
        release(&currentSem->lock);
        blockProcess(getCurrentPid(),SEMAPHORE_PASSWORD + semIndex);
        acquire(&currentSem->lock);
        currentSem->value--;
        release(&currentSem->lock);
        }
    }
```

```
while(loop){
    acquire(&currentSem->lock);
    if(currentSem->value > 0 ){
        currentSem->value--;
        loop=0;
    }else{
        release(&currentSem->lock);
        blockProcess(getCurrentPid(),SEMAPHORE_PASSWORD + semIndex);
        acquire(&currentSem->lock);
        if(currentSem -> value > 0 ){
            currentSem->value--;
            loop=0;
        }else{
            release(&currentSem->lock);
        }
    }
```

Como puede observarse, la implementación original no contemplaba que el código puede ejecutarse concurrentemente por dos procesos diferentes. Supongamos que un proceso realiza un wait y el valor del semáforo es cero, entonces el mismo se bloquea luego de hacer el release del spinlock. Si el proceso que tiene el acceso ahora, realiza un post , se desbloquea el proceso, lo cual en nuestro sistema significa que el mismo se agrega a la cola de listos. Sin embargo, el mismo no se ejecuta instantáneamente, si no cuando el scheduler lo determina. Por tanto, para cuando el proceso vuelva a tener el control de la cpu, puede haberse realizado un wait por otro proceso, lo cual nos advierte la necesidad de agregar ese if nuevamente. Cabe destacar que el PVS afirma lo siguiente sobre dicho:

if "/root/Kernel/semaphore.c 79 err V547 Expression 'currentSem->value > 0' is always false."

Por lo justificado anteriormente, se trata de un falso positivo.

Comunicación entre procesos

La implementación de la comunicación entre procesos se realizó a través de la implementación de un pequeño "file system". En este caso, el concepto de file o archivo refiere a un objeto el cual encapsula una fuente o sumidero de información junto con las operaciones requeridas para acceder a esta. A su vez, el concepto de file system hace referencia al sistema que permite administrar los archivos existentes y la interacción del usuario o del mismo kernel con estos. Todo elemento, mejor dicho, archivo del file system tiene un nombre que lo diferencia de los demás y con el cual puede ser accedido por cualquier proceso, asumiendo que este último conozca el nombre.

Operaciones del file system

El file system cuenta con varias operaciones que permiten a los procesos interactuar con él. En primer lugar, la operación "create" permite a un proceso crear un nuevo archivo que vivirá en el file system y la operación "unlink" permite eliminarlo de este, estas operaciones trabajan con nombres o cadenas de caracteres. En segundo lugar, las operaciones "open" y "close" permiten a un proceso abrir un archivo para poder interactuar con este y una vez terminada esta cerrarlo, estas operaciones trabajan con valores numéricos llamados file descriptors los cuales son propios de cada proceso. Para trabajar con file descriptors tenemos también las operaciones dup y dup2 que permiten duplicar file descriptors dando al proceso un nuevo valor numérico que refiere a la misma apertura. Por último, están las operaciones "read" y "write" que permiten al proceso escribir y leer de un archivo, esto se hace a través aperturas previamente existentes y se utiliza un file descriptor que refiere a estas.

Estructura subyacente del file system

En este file system uno puede encontrar tres tipos de estructuras o descriptores que permiten la administración de los archivos, de mayor a menor nivel de abstracción tenemos los file descriptors, los open files o aperturas y los inodos. Cada file descriptor refiere a un open file y cada open file refiere a un inodoro.

La estructura más básica es el inodo. Esta es una estructura de datos genérica que encapsula la información propia de un archivo particular en el file system, esta es creada y eliminada con las operaciones "create" y "unlink". Estas estructuras pueden ser aprovechadas por los diferentes tipos de archivos al momento de definir sus funcionalidades. Un inodo permite a un archivo reservar semáforos y colas de bloqueados, llevar un índice de lectura y uno de escritura, llevar la cuenta de la cantidad de aperturas hechas sobre este, tener una referencia a la memoria utilizada por este, etc.

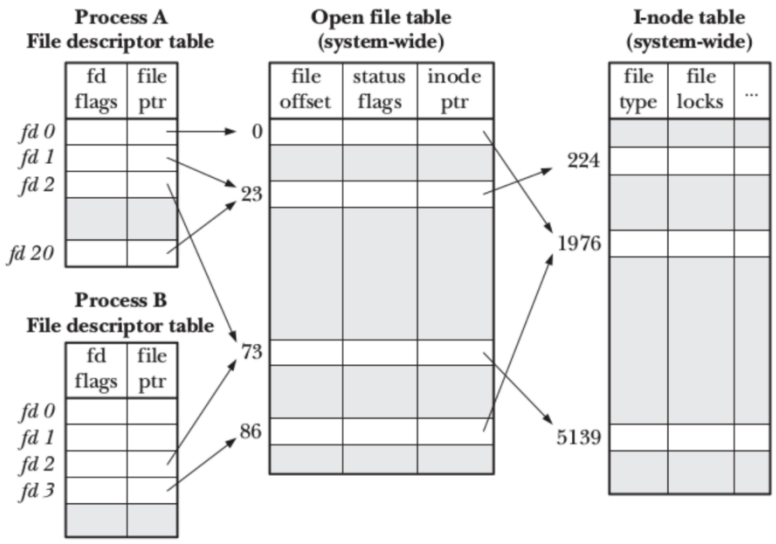


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

En segundo lugar tenemos los open files, estas son estructuras de datos creadas con la operación “open” en base a un código preexistente y eliminados con “close” cuando se cierra el último file descriptor que apunta a este. Un open sirve como intermediario entre los procesos y los archivos, tienen los permisos propios de esa apertura y llevan la cuenta de los file descriptors que refieren a él.

Por último están los file descriptors, estos son números enteros propios de cada proceso que le permiten a este interactuar con un archivo particular a través de la apertura de ese archivo. Estos son creados con las operaciones “open”, “dup” y “dup2” y eliminados con la operación “close”. Los file descriptors son virtuales en el sentido que los file descritpros de un proceso no tienen por que tener ningún tipo de relación con los de otro proceso.

Tipos de Archivos

El sistema acepta cuatro tipos de archivos diferentes, los teclados, las consolas, los archivos regulares y los fifos.

En primer lugar, un teclado o “keyboard” funciona como un intermediario entre procesos y el handler del teclado. Este brinda la funcionalidad necesaria para poder leer cadenas de caracteres de un teclado de manera atómica y sin busy waiting, esto se garantiza con el uso de semáforos y colas de bloqueados, además cuenta con un mecanismo de EOF.

En segundo lugar, una consola o “console” funciona como un intermediario entre procesos y el panel de pantalla. Este brinda la funcionalidad necesaria para poder leer cadenas de caracteres de un teclado de manera atómica, esto se garantiza con el uso de un semáforo.

En tercer lugar, un “fifo” o “pipe con nombre” es un archivo que puede ser abierto únicamente para leer o para escribir, pero no ambas. Este cuenta con mecanismos de sincronización que permiten a diferentes procesos trabajar sobre él simultáneamente sin la posibilidad de anomalías. Cada fifo cuenta con escritura y lectura atómica, lectura y escritura bloqueante, un mecanismo de EOF para lectura y otro para escritura y apertura bloqueante en caso que el fifo sea abierto únicamente por escritores.

En cuarto lugar, un archivo regular puede ser abierto tanto para escribir como para leer como para leer y escribir. Este cuenta con índices de lectura y escritura, un mecanismo de EOF para la lectura y no cuenta con bloqueo en sus operaciones. Los archivos regulares cuentan también con una mayor cantidad de memoria reservada que por ejemplo los fifos.

Manual del usuario:

El proyecto puede compilarse usando un manejador de memoria Buddy o un List Free. List Free tiene la ventaja de que no tiene fragmentación interna, pero no tiene coalescencia. Para compilar con uno o con otro, se debe acceder al archivo Kernel/Makefile y setear la variable MEMMANAGER por buddy o por listfree.

El listado de los comandos disponibles es el siguiente.

A tener en cuenta, el parámetro opcional [-p] permite a un proceso interactuar con el file system "en nombre de otro proceso", esto quiere decir que al ejecutarse un comando con este parámetro, siendo p el pid deseado, el file system interpretará que lo está accediendo el proceso de pid p. Es importante aclarar que esta funcionalidad fue agregada exclusivamente para demostrar las funcionalidades y no es utilizada en userland para nada más que esto ya que creemos que un proceso no debe tener acceso a este tipo de información.

Para marcar el EOF de la entrada estándar desde el teclado se debe presionar la tecla con el símbolo '~' del teclado inglés.

Para utilizar flags, se debe escribir el signo =. Por ejemplo, **kill –p=1 -k=0**

- mkreg <filename> : crea un archivo regular en el file system con el nombre <filename>
- mkfifo <filename> : crea un pipe con nombre o fifo en el file system con el nombre <filename>
- unlink <filename> : elimina el archivo de nombre <filename> del filesystem una vez que se hayan cerrado todas sus aperturas
- open [-p] <mode> <filename> : abre un archivo preexistente del file system con el nombre <filename> con los permisos asignados en <mode>, estos son 0 para solo lectura, 1 para sólo escritura y 2 para lectura y escritura
- close [-p] <fd> :cierra un file descriptor previamente creado, una vez cerrados todos los file descriptors se cierra la apertura de la que estos dependen
- write [-p] <fd> : permite escribir en un archivo abierto por el file descriptorf <fd>, si los permisos indicados en la apertura no lo permiten devuelve un mensaje de error
- read [-p] <fd> <count>: permite leer una cantidad de bytes <count> de un archivo abierto por el file descriptor <fd>, si los permisos indicados en la apertura no lo permiten devuelve un mensaje de error
- dup [-p] <oldfd> : dado un file descriptor preexistente <oldfd> crea un nuevo file descriptor que apunta a la misma apuerta
- dup2 [-p] <oldfd> <newfd> : dado un file descriptor preexistente <oldfd> toma al valor <newfd> como una referencia a la misma apertura que <oldfd>
- printFileContent <filename> : imprime sin tener en cuenta los índices des escritura ni lectura una porción de la información que se encuentra en un archivo
- prinFileInfo <filename> : imprime la información almacenada por el inode de una archivo en el file system con el nombre <filename>

- printFdTable [-p] : imprime las primeras posiciones de la tabla de file descriptors de un proceso particular

- kill [-p] [-k]: el flag -p recibe el pid del proceso activo. El flag -k recibe un código de señal. Con k=0, se está matando a un proceso. Con k=1, se está bloqueando a un proceso. Con k=2, se está desbloqueando a un proceso. Un proceso bloqueado por kill solo puede ser desbloqueado por kill. Notar como esta implementación distinta de kill es más parecida a la original de Linux, y absorbe al comando block que pide la cátedra.

- help: muestra una lista con todos los comandos disponibles.

- mem: Imprime el estado de la memoria

- ps: Imprime la lista de todos los procesos con sus propiedades: nombre, ID, prioridad, stack y base pointer, foreground

- loop: Imprime su ID con un saludo cada una determinada cantidad de segundos.

- nice [-p] [-b]: Cambia la prioridad de un proceso. El flag -p determina el proceso. El flag -b determina el cambio en el niceness del proceso. Puede valer entre -20 y 19.

- sem: Imprime la lista de todos los semáforos con sus propiedades: estado, los procesos bloqueados en cada uno y cualquier otra variable que consideren necesaria.

- cat: Imprime el stdin tal como lo recibe.

- wc: Cuenta la cantidad de líneas del input.

- filter: Filtra las vocales del input.

- pipe: Imprime la lista de todos los pipes con sus propiedades: estado, los procesos bloqueados en cada uno.

- phylo: Implementa el problema de los filósofos comensales.