2º Cuatrimestre 2016

Sistemas Operativos

Segundo TPE: Construcción del Núcleo de un Sistema Operativo

Carla Barruffaldi, 55421
Luciano Bianchi, 56398
Tomás Cerdá, 56281

Introducción

En este trabajo práctico se extendió el kernel monolítico realizado en la materia Arquitectura de Computadoras a un kernel simple que soporta multiprocesos.

Se implementó un Scheduler, una abstracción de un proceso, una tabla de procesos, un memory allocator, fifos como mecanismo de IPC, mutexes globales y variables de condición. Para demostrar el funcionamiento de estas implementaciones se realizó una resolución del Problema de los Filósofos y dos resoluciones del problema del Productor y Consumidor: una por IPC y la otra con mutexes y variables de condición.

Kernel Space

Memory Allocator

Se presentan dos memory allocator, uno que reserva y libera páginas de 4 KiB para el uso de datos y el otro páginas de 2 MiB para el uso del stack de los procesos. Se toma la asunción de que ningún proceso requerirá más de 2 MiB de stack.

En cuanto a sus implementaciones, ambos se llevaron a cabo mediante una estructura de pila. Se quita (pop) una página de la pila al reservar y se guarda (push) una página al liberar.

Para el uso de datos, tanto el Kernel Space como el User Space utilizan el mismo memory allocator, es decir que se liberan y reservan páginas del mismo Stack.

System Calls

- **Malloc**
- **Free**

Procesos y Scheduler

Se realizó una abstracción de un proceso donde a través de esta se realizan todas las operaciones que conciernen a un proceso. Entre ellas las más importantes son:

- Crear, destruir y matar a un proceso.
- Guardar y obtener Stack Pointer.
- Bloquear o desbloquear un proceso.
- Obtener proceso dado un PID.
- Asignar foreground.
- Obtener información como PID, PPID, File Descriptors abiertos, páginas reservadas, etc.

En cuanto a la información que guarda un proceso se encuentra:

- Estado: ready, blocked o deleted. No se guarda running pues no se consideró necesario, el Scheduler conoce qué proceso se encuentra corriendo. El estado delete representa que se debe borrar el proceso.
- PID, PPID y nombre.
- Páginas de memoria obtenidas mediante la system call malloc.
- Stack Pointer y dirección de comienzo de stack.
- File descriptors abiertos.

Todos los procesos se guardan en una tabla de procesos implementada como un arreglo, donde el índice del arreglo representa el PID del proceso guardado en dicho lugar. Esto permite acceder rápidamente a un proceso dado un PID.

Al crear un proceso se le puede pasar un único argumento, el cual se guarda en el registro RDI del nuevo proceso.

Para darle foreground a un proceso, el proceso que lo está dando si o si debe ser el que tiene el foreground actualmente. Esto es para evitar que cualquier proceso pueda tomar el foreground sin restricción.

Para obtener la información de todos los procesos activos se requiere el uso de dos System Calls. Una con la cual se obtiene un arreglo de los PIDs de los procesos activos, y otra con la cual se obtiene información de un proceso dado un PID. Se tomó esta decisión porque implementar una única System Call que devuelva la información de todos los procesos puede ser costosa en términos de espacio.

Se implementó un Scheduler mediante el algoritmo Round Robin que internamente utiliza una lista circular. En cada nodo se guarda la abstracción a un proceso (puntero), un quantum y el siguiente proceso a correr. El quantum por default de cada proceso es de 1. Este puede aumentar dadas dos operaciones:

- Yield. Se aumenta en 1 el quantum del siguiente proceso a correr, pues el proceso actual que hace yield “robó tiempo” al siguiente proceso. Sería una compensación.
- Finalizar un proceso. Mismo razonamiento al anterior.

Dado que la lista circular puede ser accedida de forma concurrente y así corromperse, por ejemplo, se agrega un proceso a la lista pero hay una interrupción del timer tick antes de terminar de agregar el proceso, se implementó un lock denominado “superlock” .El Scheduler al elegir el siguiente proceso a correr verifica y obtiene el superlock mediante una operación TSL. Si ya se encontraba tomado, simplemente retorna y permite correr al proceso que ya se encontraba corriendo.

El superlock se utiliza al agregar o quitar un proceso de la lista, elegir el siguiente proceso a correr y en la implementación de Mutexes. Esto último se explica más adelante.

Al elegir el siguiente proceso a correr, se busca hasta encontrar alguno cuyo estado sea ready. En la clase se planteó la idea de que un proceso bloqueado haga yield() y verifique si

seguía bloqueado. Esto significaba que el Scheduler ponía a correr un proceso y éste veía si su estado seguía siendo bloqueado. Se llegó a la conclusión de que esto era erróneo: el Scheduler simplemente elige el siguiente proceso a correr, ignorando los bloqueados. Un proceso sale de este estado mediante alguna interrupción, como un mutex liberado o un write a un buffer previamente vacío, por ejemplo.

Cuando un proceso termina normalmente, este lo indica mediante la system call end. Se liberan sus recursos, se quita de la lista de forma inmediata y se pone a correr el siguiente proceso.

Esto no es así cuando se mata un proceso ajeno mediante la system call kill. En este caso el proceso entra en un estado delete, y el Scheduler al encontrarse con un proceso en este estado cuando busca el siguiente proceso a correr lo quita de su lista. Se tomó esta decisión para no tener que navegar a través de una lista circular para quitar un elemento específico. Navegar a través de la lista no es una operación segura por lo que requeriría del superlock, cuyo uso debe ser el menor posible.

System Calls

Se presentan únicamente las system calls más importantes.

- **Exec**. Crea un proceso.
- **End**. Termina un proceso.
- **Kill.**
- **Yield.** Permite correr al siguiente proceso.
- **Process Info.** Obtiene toda la información de un proceso.
- **Processes Pids.** Obtiene un arreglo de pids correspondientes a los procesos en la tabla de procesos.
- **Set foreground.**

Sleep

Se modificó la implementación del Sleep del TP anterior para que funcione en forma de multiprocesos. El Sleep recibe los milisegundos a dormir el proceso y este es bloqueado.

Al ejecutar la System Call Sleep el proceso es guardado en una Lista Simplemente Encadenada y bloqueado. Se guardan los Timer Ticks necesarios para despertarse. Cuando se da una interrupción de Timer Tick se recorre la lista y se resta en 1 los Timer Ticks almacenados de cada proceso. Si llegó a 0 se quita el proceso de la lista y se desbloquea.

System Calls

- **Sleep.** Recibe los milisegundos a dormirse.

Input Output Estándar

Un proceso que quiera leer de entrada estándar puede que se bloquee dada alguna de las siguientes dos condiciones:

- No es el proceso en foreground. Se desbloqueará cuando se le sea cedido.
- No hay nada para leer. Se desbloqueará dada una interrupción del teclado.

System Calls

- **Read.** Con File Descriptor 0.
- **Write.** Con File Descriptor 1.

Mutexes

Se realizaron mutexes globales. Se abren a partir de una string y se obtiene una key. Con esta key se realizan las operaciones de bloquear, desbloquear y cerrar mutex.

Se implementaron a partir de un arreglo de una estructura mutex, donde el índice del arreglo representa la key de un mutex.

La estructura mutex está comprendida por:

- El lock. Puede estar LOCKED o UNLOCKED.
- Una cola de procesos bloqueados esperando el lock.
- El nombre que lo identifica para que otros procesos puedan obtener su key una vez abierto.

Al tomar un lock este se testea y setea mediante una operación TSL. Esta puede implementarse con XCHG en el caso de Intel. Si el testeo falla se agrega el proceso en la cola y se bloquea. Dado que se requiere que esta operación de bloquear y agregar en la cola sea atómica se toma el superlock. Explicación:

```
int mutex_lock(int key) {
    set_superlock();
    mutex * m = &open_mutexes[key];
    if (!_unlocked(&m->locked)) { /* Operación TSL */
        /* Si se encola y no bloquea, puede que haya una interrupción y sea desencolado,
        sea desbloquado y luego se bloqueara cuando deba correr nuevamente.
        ** Si se bloquea y no encola, como está bloqueado no se encolará, entonces no se
        desencolará y nunca desbloqueara. */
        queue_process(m, p);
        block_process(p);
        unset_superlock();
    }
    else
```

```
        unset_superlock();
}
```

Al desbloquear un mutex, se libera el lock. Si hay un proceso en la cola, se desencola, se lo desbloquea y se cierra nuevamente el lock.

Al cerrar un mutex se desencolan y desbloquean todos los procesos y se libera el espacio en el arreglo de mutexes. Esta operación debe hacerse cuando se está totalmente seguro de que el mutex no está siendo utilizado por ningún proceso.

Por último, se puede obtener información de los mutexes activos para el comando ipcs.

System Calls

- **Mutex open.**
- **Mutex close.**
- **Mutex lock.**
- **Mutex unlock.**
- **Mutexes info.** Devuelve información de los muetexes abiertos.

Variables de Condición

Se implementaron obviamente con el uso de mutexes y una estructura análoga a la de los mutexes: un arreglo de colas básicamente. Se abren y cierran igual que los mutexes.

La operación ‘Conditional Wait’ recibe una variable de condición y un mutex. Este es desbloqueado y el proceso se bloquea y encola hasta que otro proceso haga Signal o Broadcast sobre la variable de condición. Cuando se hagan algunas de estas operaciones, se desencola y desbloquea uno (Signal) o todos (Broadcast) los procesos. Acto seguido, intentan tomar el mutex con el que inicialmente habían entrado.

Así se asegura que al salir del Conditional Wait, posean el mutex con el que habían entrado.

System Calls

- **Cond open.**
- **Cond close.**
- **Cond wait.** Se bloquea el proceso y desbloquea el mutex pasado por parámetro.
- **Cond signal.** Se desbloquea un único proceso y toma el mutex pasado por parámetro en Cond Wait.
- **Cond broadcast.** Se desbloquean todos los procesos y cada uno intenta tomar el mutex pasado por parámetro en Cond Wait.
- **Cond info.** Devuelve información de las variables de condición abiertas.

Mecanismo de IPC: FIFOs

Como mecanismo de IPC se realizaron FIFOs. Se implementaron mediante un arreglo de estructuras FIFO.

Estas estructuras están compuestas por un string que identifica al FIFO, una key (la cual es traducida a File Descriptor para los procesos) que es el índice del arreglo, un arreglo circular que cumple el rol de buffer, una cola para los procesos bloqueados por hacer Read con el buffer vacío y un mutex para proteger dicho buffer.

Los fifos se abren mediante un String y se obtiene un File Descriptor. Al cerrar un FIFO se liberan los recursos asociados y se desencolan y desbloquean todos los procesos. Al escribir se desencolan y desbloquean procesos bloqueados por el read mientras haya algo para leer. Tener en cuenta que puede escribirse más de lo que deseaba leer un proceso.

Los fifos son siempre de read/write, y no bloquean en el momento de su creación o en la escritura.

System Calls

- **Fifo open.**
- **Fifo close.**
- **Read.** Con el File Descriptor de algún FIFO.
- **Write.** Con el File Descriptor de algún FIFO.
- **Fifos info.** Devuelve información de los FIFOs abiertos.

User Space

Criterio Foreground y Background

Se tomó como criterio que todo proceso ejecutado desde la Shell de comandos tomará el foreground siempre y cuando no se pase como último argumento un '&'. A su vez, para dar manualmente el foreground desde la Shell existe el comando **fg**, que recibe como argumento el PID del proceso al cual dar el foreground.

Por último, para devolver el foreground a la Shell, mientras hay un proceso corriendo que lo posee, se puede hacer uso de la tecla ESC.

API Creación de Procesos

Se desarrolló una pequeña API para la creación de procesos para abstraer este procedimiento del uso de la System Call `exec()`. Los problemas que puede traer el uso de esta System Call a secas es olvidarse del uso de la system call `end()` al finalizar el proceso y devolver a algún proceso el foreground.

Se provee una función `execp()` la cual se asegura que el nuevo proceso siempre terminará con la System Call `end()` y siempre devuelve el foreground al proceso padre. Recordemos que la System Call Set Foreground únicamente otorgará el foreground si el proceso que llama a esta System Call posee el foreground.

Finalmente, `execp()` permite que se le pase un string como parámetro, dando al nuevo proceso los parámetros `argc` y `argv`, análogos a los que ya conocemos.

De esta forma, se puede escribir un nuevo programa de la misma manera que estamos acostumbrados, recibiendo `argc` y `argv` y finalizando con un `return`.

Comandos ipcs y ps

Para estos dos comandos se definieron estructuras cuya definición son conocidas tanto por el Kernel Space como el User Space. Estas estructuras representan la información de un elemento en particular. Estos elementos pueden ser un proceso, un FIFO, un mutex o una variable de condición.

Se definieron System Calls que reciben un puntero a la estructura de interés y se llena el espacio de memoria con la información. Luego un proceso en el User Space imprime la información a pantalla.

Problema de los Filósofos

El problema está resuelto con un diseño muy similar al visto en clase y el presentado en el libro de la cátedra. Difiere en lo que concierne a la variante pedida: añadir y quitar filósofos preservando el estado de éstos.

Para realizar estas dos operaciones se toma el mutex crítico que todos los filósofos comparten. Es por esto que las operaciones puede que no se lleven a cabo de forma inmediata. A su vez, no es posible realizar una nueva operación hasta que termine la operación anteriormente pedida.

A la hora de remover un filósofo se tomó el siguiente criterio: el filósofo a remover entra en un estado DYING. Cuando este filósofo “termina de pensar” verificará si se encuentra en un estado DYING. Si se encuentra en este estado procederá a ver si ambos filósofos a su costado están comiendo. Si es así se bloqueará hasta que alguno termine de comer. Finalmente, el filósofo termina su proceso y actualiza el contador de filósofos.

Para llevar a cabo esto se utilizaron dos categorías de variables de condición. La primera sirve para bloquear al proceso que escucha comandos hasta que se remueva al filósofo en cuestión. Así se asegura lo antes descrito de no poder recibir comandos hasta terminar de ejecutar el comando actual, pues esto puede corromper el el problema de los filósofos. El filósofo al ver que su estado es DYING y debe removerse, hace Signal sobre esta variable de condición despertando al proceso que escucha comandos.

La segunda categoría corresponde a una variable de condición por cada filósofo que sirve para bloquearse en caso de que al removerse ambos filósofos a su costado están comiendo. Por esta razón, al terminar de comer cada filósofo hace Signal sobre esta variable de condición correspondiente a los filósofos a su costado.

Obviamente todas estas operaciones requieren de la posesión del mutex crítico para llevarse a cabo. Este mutex es pasado como parámetro en los Wait de condición, además de la variable de condición que corresponda.

Se realiza un sleep de duración aleatoria cuando un filósofo entra en estado THINKING o EATING.

Finalmente se realizó una implementación en modo texto que realiza un output muy similar al visto en clase y también otra implementación en modo gráfico donde los filósofos están en ronda representados por círculos de colores según estado. En el Manual de Usuario se explica el significado de los colores.

Problema del Productor y Consumidor

Se presentan dos implementaciones de este problema.

Variante 1: FIFOs

En esta variante se crean dos fifos, uno por el cual se envía un mensaje por cada slot vacío y otro en el cual se envía un mensaje por cada ítem producido. Se inicia la actividad llenando el fifo de vacíos con una cantidad determinada de mensajes (que sería conceptualmente como el tamaño del buffer), y se ponen a correr el consumidor y el productor.

El consumidor lee del fifo de ítems producidos (bloquea si no hay ninguno) y cuando lee un ítem, lo consume y manda un mensaje por el fifo de slots vacíos. El productor hace lo opuesto, lee los espacios vacíos (bloquea si no queda ninguno), y al leer crea un ítem y lo envía en el fifo de ítems.

Debido a que tanto consumidor como productor sólo escriben en un fifo si antes leyeron del otro, la cantidad de *slots vacíos* + *slots llenos* es siempre la misma.

Variante 2: Mutexes y Variables de Condición

En esta versión se presenta un buffer de tamaño fijo compartido por los dos procesos. También se comparte el tamaño del mismo, para saber desde donde consumir y extraer.

Se hace uso de un mutex para proteger estas dos variables. Sin embargo, es también necesaria una variable de condición para que el productor pueda bloquearse cuando el buffer está lleno, y el consumidor pueda bloquearse cuando el buffer está vacío. Siempre al producir o consumir se hace una Señal de Condición sobre la variable.

Código Externo

Se tomó código externo para la implementación del Hook a la hora de crear un nuevo proceso y también para las funciones matemáticas necesarias para pintar los círculos de los filósofos. La fuente se presentará a continuación pero se también se muestra en el código en cuestión. Fuente:

- https://bitbucket.org/RowDaBoat/wyrm/wiki/Home
- http://stackoverflow.com/questions/2284860/how-does-c-compute-sin-and-other-math-functions

Por último, si bien el código para pintar los filósofos en círculos que cambian de color fue razonado y escrito por nosotros, la inspiración de hacerlo en dicha modalidad fue dada por el grupo de Alejo Saqués.

Ambas implementaciones del problema de Productor/Consumir son las propuestas por Tanenbaum en su libro Operating Systems: Design And Implementation.

Conclusión

Se buscó que las implementaciones sean lo más simples y modulares posibles, como también razonar y desarrollar todas las implementaciones nosotros, buscando y tomando la menor cantidad de código externo en lo posible.