Instituto Tecnológico de Buenos Aires

Sistemas Operativos

Ingeniería en Informática

Sistema Operativo Multiproceso

Segundo Trabajo Práctico Especial

Titulares: Rearden, Rodrigo; Godio, Ariel

Semestre: 2018A

Grupo: 3

Repositorio: https://github.com/juangod1/SO-TP2

Fecha: 16 de mayo de 2018

Entrega: 16 de mayo de 2018 a las 23:59hs

Autores: Delgado, Francisco (#57101)

Radnic, Pablo (#57013)

Godfird, Juan (#56609)

Ormachea, Joaquin (#57034)

Resumen

Se presenta un trabajo práctico de sistemas operativos con el fin de implementar un sistema operativo multiproceso. La implementación se realiza sobre el trabajo práctico de Arquitectura de computadoras hecho por parte de los integrantes del grupo. Para la evaluación de la calidad del trabajo práctico se implementaran programas de usuario que demuestran el correcto funcionamiento del sistema operativo.

Índice

1. Introducción

1.1. Procesos

Para este Sistema operativo un proceso es un hilo de ejecución de uno o varios programas. Este tiene asociado a él varias estructuras que harán posible el manejo del mismo, estas son: Un número identificador de proceso, este número no puede ser el mismo para dos procesos distintos. Un puntero al stack que maneja el proceso.

El sistema implementado es de carácter preemptivo, esto significa que los recursos asignados al proceso pueden ser expropiados del mismo. Esto se hace mediante la rutina de atención a la interrupción del timer tick. Cada vez que el timer tick interrumpe el procesador este genera un cambio de contexto al siguiente proceso que se debe correr.

La rutina que determina qué proceso debe correr es el scheduler, este se encarga de mantener una cola de procesos listos y procesos bloqueados.

1.2. Manejo de memoria

Para llevar a cabo el manejo de la memoria del sistema utilizamos un "memory manager" del lado del kernel que se encarga de administrar las páginas otorgadas a cada proceso, ya sea para su uso como stack o como heap. Este administrador utiliza 2 tipos de estructuras para almacenar la información correspondiente a cada proceso. Se definio que la implementacion de las funciones malloc y free y el tipo de estructura para manejar los bloques de memoria quedan a criterio del usuario. De todas formas se realizo una implementacion identica a la que utiliza el memory manager para administrar la memoria del kernel.

1.3. Comunicación entre procesos

Para la comunicación entre procesos se implementaron casillas de mensajes para poder mandar mensajes entre los processos, ademas para manejar la concurrencia entre procesos se implementan mutexes mediante semaforos.

2. Implementación

2.1. Procesos

2.1.1. Cambios de contexto

El cambio de contexto se implementa aprovechando el mecanismo de interrupcion de timer tick. Para cambiar al contexto del próximo proceso se guarda el contexto del proceso que se está desalojando en el stack del mismo. Luego se pide al scheduler cual es el puntero del stack del próximo proceso mientras que se le provee por parámetro el puntero del stack del proceso que se desalojo. La rutina de scheduler se encarga de guardar el puntero del stack del proceso que se desaloja en la estructura del mismo. Luego con el puntero a stack del proceso que sigue se cambia al próximo stack, se desapilan los registros para restaurar su contexto y termina la interrupción así volviendo al instruction pointer del proceso que sigue ya que estamos en el stack del proceso que sigue.

2.1.2. Instanciación de un proceso

Para instanciar un proceso se debe armar la estructura dentro de su stack como si estuviese en el medio de una interrupción de timer tick. para esto se pushea el instruction pointer y luego se pushean selectores de segmento y flags tal y como lo hace el procesador cuando entra en una interrupción. Finalmente se pushean registros dummy así cuando el scheduler levante el stack de este proceso que nunca corrió este es análogo al stack de un proceso que ya estaba corriendo y el scheduling puede funcionar sin problema.

2.1.3. Bloquear un proceso

Para bloquear un proceso este ejecuta un wait. El proceso cambia a bloqueado y se queda loopeando en un while agotando su quantum dentro de kernel space para quedar esperando con el puntero a instrucción donde debería. Es importante notar que no se vuelve a correr el proceso hasta que este pase a listo y el while solo se usa para agotar el quantum que se le aloco antes de bloquearse.

2.1.4. Terminar un proceso

Cuando un proceso ejecuta el syscall kill este le comunica al kernel que terminó su ejecución y debe ser terminado. El kernel se encarga de destruir todas las estructuras relacionadas con el proceso y se lo remueve de la cola del round robin para que no se vuelva a correr ya que este fue agregado cuando se ejecutó por el scheduler

2.1.5. Algoritmo de scheduling

El algoritmo de scheduling corre cada timer tick y este se basa en un round robin implementado como una lista simplemente encadenada. Al correr un proceso se vuelve a colocar el mismo al final de la cola así puede volver a ejecutarse cuando se acabe su quantum, además todo proceso que se instancie irá al final de la cola. Para determinar qué proceso se recorre la lista desde el primer elemento de la lista buscando el primer elemento que no se encuentre bloqueado.

2.2. Administrador de memoria

La implementación del mismo se definió con la idea de que sea sencillo de entender sin priorizar la eficiencia espacial. Se utilizan 2 estructuras que almacenan toda la información necesaria para reservar y liberar bloques y asignar o liberar páginas. Como estructura principal se creó unas lista encadenada de bookBlocks. Esta funciona como un libro donde se registra un bloque por proceso donde se guardan las direcciones de las páginas para el stack y para el heap de cada uno. También se guarda el indicador del final del espacio disponible en el heap(brk).

```
typedef struct bookBlockStruct *
bookBlock;

struct bookBlockStruct {
        int owner; //Este es el  pid
        void* base;
        void* stack;
        int brk; //Este puede ser otro tipo de dato para
ahorrar memoria
        bookBlock prev;
        bookBlock next;
};
```

Definimos de manera estática el tamaño deseado por página ya que al estar mapeado 1:1, nos resulta indistinto cuando empieza y termina una página. A partir de este número calculamos cuántas páginas tenemos disponibles en nuestro sector designado para memoria. Cargamos las direcciones iniciales de cada página en un array y mediante los métodos popNewPage y popReverseNewPage, vamos marcando las páginas como utilizadas. A medida que los proceso son eliminados las páginas se marcan como libre, quedando listas para un nuevo proceso.

Para diferenciar la memoria del memory manager y del kernel de la de la de los demás procesos, inicializamos páginas del libro al bootear el sistema. el "pid"/"owner" 0 es del memory manager y el 1 del kernel. Dado que los procesos arrancan con pid = 0, se decidió que todos tengan owner = pid+2. En la página del memory-manager se almacenan los bloques del libro, mientras que en la página del kernel se utiliza una segunda estructura que permite manejar la página fragmentada

en bloques. Esta estructura consiste de un bloque header con la información del bloque contiguo. Este permite saber cuánto espacio tiene, si está libre o no y conecta con el header del bloque siguiente. De esta forma es muy sencillo ir creando nuevos espacios y definir si están libres o no.

```
typedef struct dataBlockStruct *
dataBlock;

struct
dataBlockStruct {
size_t
size;

dataBlock next;
int free;
};
```

2.3. IPCs

2.3.1. Casillas de mensajes

El sistema cuenta con una estructura header llamada Post Office que encabeza una lista de message boxes el cual es similar al message queue de posix. La estructura message box tiene un int que define el tamaño en bytes de los mensajes, un byte que define si es bloqueante o no la recepcion y una cadena de caracteres que funciona como llave. A la vez cada message box es header de su cola de mensajes. El usuario accede al sistema de mensajeria creando una estructura mbdt Struct message box descriptor type y enviando su direccion por parametro a las syscalls sendMessage, recieveMessage y finalizeMessageBox. Las llaves de los message boxes tienen un tamaño de 5 bytes ya que estos proveen suficientes combinaciones como para que no se agoten en una situacion practica. El tamaño es fijo por message box. Permitiendo que el usuario receptor sepa el tamanio del mensaje que espera, y a la vez dando flexibilidad de tamanio del sistema de mensajeria a los usuarios. En un caso ideal la estructura de Message boxes deberia ser un mapa con identificador mbd t o key. Pero la implementacion era muy dificil y no mejoraria a grandes razgos la eficiencia del algoritmo de busqueda de message boxes.

2.3.2. Mutexes

Se decidio implementar semaforos de valor maximo 1 Simplificaba el desarollo y garantizaba la funcionalidad pedida. En un futuro se podria extender para permitir valores mayores a 1 en inicializacion. El sistema de semaforos cuenta con una estructura header llamada Traffic Control que encabeza una lista de list entries. Cada list entry tiene un semaforo el cual tiene una llave, un valor y una cola de procesos bloqueado. El usuario accede al sistema de trafico enviando un numero entero identificador a las syscalls (semStart, semWait, semSignal, semStop) que funciona como identificador de semaforos. En un caso ideal la estructura de Semaforos deberia ser un mapa con identificador int key. Pero la implementacion resulto dificultosa y no mejoraria a grandes razgos la eficiencia del algoritmo de busqueda de semaforos.

3. Programas de usuario

El trabajo práctico cuenta con programas a nivel de usuario para demostrar el funcionamiento. Para ver la lista de de programas de usuario uno puede ejecutar el comando help desde la terminal, este comando muestra todos los programas de usuario disponibles junto a una muy breve explicación de su funcionamiento. A continuación explicaremos con un poco más de detalle los programas de usuario más relevantes al trabajo práctico.

3.1. Demo productor consumidor

El programa, el cual se puede ejecutar con el comando de shell prodConsDemo, posee una variable que solo puede tomar el valor 1 o 0, esta comienza inicializada en 0. El productor incrementa el valor de la variable mientras que el consumidor la decrementa. Si un consumidor se genera pero la variable esta en 0 entonces este espera hasta que un productor incremente la variable y asi puede consumir. En el caso de que se genere un productor pero la variable este ya en 1 este se bloqueara esperando que un consumidor decremente la variable para que este la incremente denuevo. Asi podemos tener una cantidad variable de consumidores y productores en tiempo de ejecucion y se puede manejar perfectamente un buffer limitado.

3.2. Memory manager demo

El objetivo de este programa es mostrar la alocacion y liberacion de memoria por proceso. Este muestra paso a paso como se aloca y libera memoria para un proceso. Ademas muestra como se libera la memoria cuando un proceso termina.

3.3. Demo processos en background

para ver la demcostración de procesos en background se puede ejecutar el comando 'testBackgroundProcess.[es]te corre un proceso en background el cual puede ser llevado a fg ejecutando 'fg pid' donde pid es el pid del processo que se corrio en bg. Si se quiere correr directamente en foreground se puede ejecutar 'runfg testBackgroundProcess'.

3.4. Tests

El trabajo tambien contiene una suite de tests para garantizar el correcto funcionamiento del sistema. Los tests incluyen un test general el cual testea semaforos y mensajes. Tambien se tienen tests de memorymanager e IPCs. Para poder ver todos los tests se pude ejecutar el comando 'help test' para ver las suites que hay.