Trabajo Práctico 2

Sistemas Operativos

Segundo Cuatrimestre 2008

Implementación de un sistema operativo multitarea

Abramowicz Goñi Pan

Resumen

Se pide realizar un programa que conmute diferentes procesos, asignandoles a cada uno un tiempo de ejecucion. El multitasker no corre sobre ningún Sistema Operativo,
se ubica en memoria utilizando un bootloader GRUB. El mismo debe ser implementado para plataformas Intel de 32 bits, en modo protegido.
El multitasker debe ser preemptivo y tomar como base la interrupción de hardware correspondiente al timer tick.

Introducción

Para esta entrega se parte del Trabajo especial presentado para la materia “Arquitecturas de computadoras” y se procede a implementar el multitasker desde ahi.
En cuanto al multitasker, este presenta 2 tipos de funcionamiento: round robin y round robin con prioridades, los cuales serán explicados con mas detalle en la sección correspondiente.
En cuanto al desalojo de procesos, se decidio no utilizar el método provisto por el procesador (recurriendo a TSS) sino que se implementó uno desde cero.
La preemptividad del multitasker se asegura dado que los procesos que están corriendo no pueden hacer nada para evitar ser sacados del procesador ya que no participan en esto sino que se utiliza la interrupcion INT08 para determinar cuando un proceso debe desalojar el mismo y se llama a un scheduler de procesos para realizar el traspaso.

Organización de la entrega

El código para el trabajo práctico se divide en 2 grandes partes, la carpeta src, donde se encuentra tanto el codigo fuente C como el codigo fuente de assembler, y la carpeta include donde se guardan los archivos de headers. Tanto include como src comparten una subdirectorio llamado drivers donde se encuentra el código que maneja los distintos dispositivos (serial, video, tty, teclado). Por otro lado el directorio src provee un subdirectorio de nombre apps donde se aloja tanto el codigo fuente de las aplicaciones del sistema operativo como los header de las mismas.
Ademas, en el raiz se ubican el Makefile y los subdirectorios de la Imagen del Sistema operativo y el codigo binario del mismo.

Contexto de Tareas

Se optó por implementar una versión propia del switcher de contextos de tareas. Es interesante que debido a que cada proceso tiene su propia zona de memoria no es necesario hacer un backup del stack de los mismos sino que alcanza con hacer backup de los registros en el momento en que se cambia de proceso y restaurarlo despues. Ya que el scheduler guarda en cada registro asignado a los procesos cual era su ESP en el momento en que estos abandonaron el microprocesador.

Generador de contextos

Cuando se crea un nuevo proceso se genera el stack frame del mismo, esto se implementa por medio de una funcion de assembler de nombre createStackFrame, la misma recibe como parametro la direccion del nuevo ESP y un puntero a funcion que corresponde a la nueva funcion a ejecutar.
Lo que hace es reemplazar el registro ESP por el valor que recibe como parametro y luego pushea el estado de los registros, CS y ESP. Esto se hace para que cuando el scheduler le de el procesador a este proceso, el mismo encuentre su contexto sin inconsistencias. Ademas, pushea los parametros que reciben todas las aplicaciones del sistema operativo humix (pid, ppid, parametros) para luego pushear la direccion de retorno de una funcion de mantenimiento del kernel que se encarga de eliminar al proceso de la cola de procesos del scheduler. Finalmente retorna en su nombre el valor de ESP a la función createProcess.

Generador de procesos

Los procesos se crean llamando a CreateProcess, la cual se encarga de: por un lado le asigna una pagina de memoria al stack y otra al heap, luego inicializa la misma y finalmente llama a createStackFrame para guardar toda esta informacion en arreglo de estructuras de tipo process_t, las mismas serán utilizadas luego por el scheduler y la funcion top para realizar las tareas de multitasking. CreateStackFrame se encarga de dejar armado el stack del nuevo proceso como para que cuando empieze a ejecutarse lo haga sin problemas. Vale aclarar que es aquí donde se asignan las prioridades a los procesos para que luego el scheduler decida cuanto tiempo viviran en el procesador. En caso de que no se asignen prioridades el scheduler mismo asignara la minima si el numero seteado no fuera valido dentro de los parametros de este sistema operativo.

Kill

La conocida función UNIX kill fue implementada como un system call que recibe 2 parámetros, actualmente solo esta soportada SIGKILL, haciendose posible en el futuro implementar una versión mas completa y con mas funcionalidad.

En el caso de recibir SIGKILL, kill fue implementada como una funcion que busca en el vector de procesos al pid deseado, setea el estado del mismo como NONE y luego recorre recursivamente este arreglo buscando hijos y demás descendientes del proceso buscado.

Entrada Salida

Cuando se trata de entrada/salida se mantuvo el mismo modelo que se tenía anteriormente realizandole las modificaciones necesarias para que funcionara en esta nueva versión del sistema operativo. Para brindar mayor escalabilidad y facilidad al realizar programas se maneja la entrada/salida por medio de file descriptors, que hacen que la lectura o escritura sean transparentes para el usuario, solo basta indicar el file descriptor que se quiere utilizar y el sistema operativo se encarga de llamar a las funciones que hagan falta por lo que la sintaxis para escribir en el puerto serial y la de un printf a stdout no varían mas alla de indicar el file descriptor destino.

Una vez que el kernel cargó, este le asigna un FD de lectura y un FD de video a la TTY activa, se hablará mas de TTYs en la sección correspondiente.

Ademas vale destacar que la librería de entrada/salida que se le ofrece al programador del sistema operativo (se encuentra en io.c) es bastante completa e implementa las funciones de la conocida stdio de UNIX, lo cual ayuda tambien a facilitar al programador las herramientas para desarrollar al no tener que aprender nuevas funciones.

TTY

Las multiples TTYs fueron implementadas como un vector de TTYs, donde la estructura que define a cada una indica si las mismas están activas y mantienen datos sobre los File Descriptors que estas tienen asignados. Actualmente se soportan hasta 7 TTYs, lo cual es modificable en el archivo tty.h.

Las TTYs funcionan independientemente unas de otras y tienen como agregados muy util la posibilidad de hacer scroll dentro de las mismas para ver comandos anteriores y sus respuestas por medio de la tecla pg up y pg down, esto es posible porque guardan un buffer de video mas alla de lo que la pantalla llega a mostrar. Ademas las TTYs implementan versiones de getByte y setByte propias a las mismas para controlar el flujo de entrada/salida de los procesos asociados a ellas, esto decidio resolverse asi ya que debia controlarse la

como los procesos que corren consumen caracteres de la entrada o escriben en la salida.

Scheduler

El Scheduler funciona de dos maneras: en round-robin y en round-robin con prioridades, el funcionamiento de las dos es similar, con la salvedad de que el primero siempre le asigna un tiempo de procesador de 1 timer tick a cada proeso, mientras que el segundo asigna tiempos que varian entre 1 y 4 dependiendo de la prioridad de cada proceso. La implementación del scheduler va de la mano del cambio de contextos ya que es el scheduler el que le dicta cual es la direccion del nuevo ESP. La version entregada solo ofrece soporte para 64 aplicaciones facilmente expandible a mas modificando la constante simbolica correspondiente.
El scheduler esta implementado como una cola circular que cada vez que debe dar el procesador a otro proceso actua de la siguiente manera:

- Guarda el ESP actual en la estructura del proceso actual.
- Lee el estado del siguiente proceso en la lista, si el estado del mismo no es PROC_READY entonces lee el del proximo.
- Repite el segundo paso hasta encontrar un proceso con estado PROC_READY
- Retorna el subindice en la cola del nuevo proceso

En el caso del scheduler con prioridad la implementacion es muy similar

- Guarda el ESP actual en la estructura del proceso actual.
- Si el proceso actual estuvo menos tiempo en el procesador del tiempo que debe entonces lo deja y le suma 1 a la cantidad de time slots que vivio
- Por el contrario, si ya cumplio su tiempo de vida:
  Lee el estado del siguiente proceso en la lista, si el estado del mismo no es PROC_READY entonces lee el del proximo.
- Repite el tercer paso hasta encontrar un proceso con estado PROC_READY
- En la estructura que define a este proceso setea su tiempo vivido con 0
- Retorna el subindice en la cola del nuevo proceso

De esta manera, el scheduler mantiene a los procesos bloqueados fuera del procesador hasta que estos se desbloqueen. Vale aclarar que los procesos bloqueados por tratar de leer de una entrada sin datos son desbloqueados cuando alguien escribe en esa entrada por el controlador de la misma.

Estado de Procesos

El estado de los procesos en cualquier momento dado es uno de los siguientes:

- PROC_BLOQUED
- PROC_SLEEP_BLOQUED – Para procesos que estan durmiendo
- PROC_STDIN_BLOQUED – Para procesos que se bloquearon leyendo una entrada vacia
- PROC_CHILD_BLOQUED – Para procesos que están esperando que un hijo termine de ejecutarse
- PROC_SEM_BLOQUED - Para procesos que están esperando un semáforo para acceder a una shared memory
- PROC_READY – Para procesos que no estan bloqueados ni ejecutandose
- PROC_EXECUTING – Para procesos que estan ejecutandose
- NONE – No es un estado en sí sino que es el estado que figura en las entradas del arreglo de procesos del scheduler cuando ese elemento está vacio.

Ademas, a cada proceso le corresponde una prioridad entre 0 y 4 que indica la importancia del mismo, siendo 0 la minima y 4 la máxima, cualquier intento de cambiar la misma por un valor no comprendido entre estos dos hará que el proceso automáticamente tenga prioridad 0.
La prioridad de los procesos se utiliza principalmente para determinar cuantos time slots se le asigna a cada proceso (recordar que 1 time slot = 1 timer tick), los procesos de prioridad 0 permanecen 1 time slot en ejecución, los de prioridad 1 permacen 2 time slots, etc.
El Scheduler ofrece además una interface para saber que procesos ocupan el procesador en un momento dado y en que porcentaje, estado de los mismos, etc. La misma se llama con la funcion top(), y como ejemplo del uso de la misma se encuentra la funcion topApp() que muestra en formato tabular esta información.
Para la eliminacion de procesos se cuenta con el comando kill {pid} que simplemente recorre la cola de procesos buscando aquel proceso cuyo pid sea igual al indicado, ademas, ejecuta la misma funcion sobre los hijos de este proceso, es decir, sobre los procesos cuyo campo parent en la estructura que lo define sea igual al pid indicado. Esta funcion se ejecuta recursivamente hasta que no encuentre mas procesos descendientes del proceso inicial con pid indicado.
Se considera que a menos que un proceso este en estado PROC_READY, el mismo no ocupa timeslots del procesador y cuando se ejecute la funcion top esto es lo que se vera reflejado en el output.

Excepciones

El sistema operativo maneja ciertas excepciones basicas utilizando la IDT. Estas excepciones son:

- Divide by Zero
- General Protection Fault
- Page Fault
- Out of Bounds
- Invalid OP Code

Lo que se hace para capturarlas es definir un handler en libasm.asm el cual se registra en la IDT en la posición correspondiente. En todos los casos el handler simplemente llama a una función en C que imprime un mensaje informativo sobre que tipo de excepción ocurrio, acto seguido procede a matar al proceso que lanzo la excepción y a desbloquear a su padre en caso de que este lo estuviera esperando.

IPC

Para permitir la comunicación entre procesos se implementó el IPC de UNIX shared memory, acompañado por supuesto de una implementación de semáforos.
Los shared memory se identifican por una key que debe ser única y son guardados en un arreglo de registros de tipo s_ipc_block_shm los cuales identifican univocamente a los mismos.
La implementacion de shared memory utiliza memoria del kernel en su creacion y aprovecha la librería de paginacion para conseguir que se le asigne memoria libre. La librería de shared memory se utiliza de manera similar a la de UNIX, contando con las funcion shm_open, shm_close y mmap.
Por otro lado los semáforos tambien son guardados en un arreglo de estructuras, con la diferencia de que el tipo de las mismas es s_ipc_block_sem. Cuando un proceso pide un semáforo lo que se hace es guardar la información de este en una estructura de tipo s_ipc_block_proc_sem, la cual se usa para determinar que procesos pidieron ese semáforo y que procesos estan bloqueados esperandolo. De nuevo, la interface de esta librería es similar a la de UNIX, ya que cuenta con funciones sem_set, sem_up, sem_down y sem_close.

Intérprete de comandos

El interprete de comandos funciona de manera similar al de bash y cuenta con historial de comandos ingresados recientemente, obviamente el historial no es presistente una vez que se apaga la computadora y se vuelve a encender. Ademas, al igual que en UNIX, si se escribe el carácter "&" al final del nombre de un comando, este se ejecutara en background, lo que se detalla a continuación.

El shell funciona en un ciclo infinito que lee comandos de STDIN y los ejecuta, la forma de utilizarlo es similar a la de UNIX y acepta comandos de hasta 50 caracteres (facilmente modificable). Cuando el shell detecta que se quiere ejecutar un programa, este se forkea y su hijo ejecuta al mismo, a cada programa se le pasa como parametros su pid, ppid y una lista de los parámetros ingresados por linea de comandos.

Los programas que pueden ser llamados desde el shell se registran (en el archivo shell.c) como handlers que llaman a los mismos, de esta manera se mantiene la uniformidad en la llamada.

Una de las debilidades del mismo es que la historia de comandos solo llega a abarcar como maximo 10, aunque esto es simplemente modificable, ya que la constante simbolica que lo define es la unica limitacion en este aspecto.

Fork

Este método para la duplicación de procesos fue implementado mediante un syscall.

Procesos en background

Se dispone de varios métodos para controlar si los procesos se ejecutan en background o foreground. Por un lado se cuenta con la posibilidad de anexar "&" al final de un comando para que el proceso corra en background desde el inicio, por otro lado se cuenta con el comando fg para traer a foreground al ultimo proceso que se ejecutó en background, asi como tambien su contraparte bg, que envia a background el ultimo proceso detenido en foreground. Para detener un proceso se optó por implementar una combinación de teclas ala linux, para hacerlo basta con presionar CTRL+Z. Cuando un proceso esta en background se considera que no puede leer de la entrada estandar ni escribir en la salida estandar, lograr esto no fue facil al principio pero se soluciono

Administración de memoria

Se eligio utilizar un modelo flat en cuanto a la unidad de segmentación, ya que al no poder desactivarla, lo mas parecido a eso es definir 4 segmentos diferentes, los cuales son:

- 2 Segmentos de codigo que ocupan toda la memoria cada uno, uno de 16bits y otro de 32 bits.
- 2 Segmentos de datos que ocupan toda la memoria cada uno, uno de 16 bits y otro de 32 bits.

Lo interesante es el modo en que se realiza paginación, porque ya que se ofrece multitasking debe tambien ofrecerse la posibilidad a cada proceso de tener una zona de memoria propia a la que los demas no puedan acceder. Esto se resolvio de la siguiente manera:

- Se inicializa el modulo de paginación indicandole cuanto espacio ocupa el kernel del sistema operativo para asignarle a este la cantidad de paginas que necesite, tambien se informa de la zona de usuario para asignarle las paginas necesarias, por ultimo, se guarda el directorio de paginas en CR3 y se setea el bit de paginacion.
- Cada vez que se pida una nueva pagina, la librería de paginacion busca la primer pagina vacia y la reserva, asi como tambien otras paginas si ese fuera el caso.

- Para dar de baja o de alta una pagina basta con setear en el registro de atributos de la misma el ultimo bit en 0 o 1 dependiendo de si se la quiere dar de baja o de alta.

Por otro lado, a diferencia del trabajo presentado en “Arquitecturas de computadoras”, se ofrece la posibilidad de pedir memoria dinamicamente, utilizando el ya conocido malloc(). Este funciona de la siguiente manera:

- Se inicializa el memory manager cuando arranca el sistema operativo, el mismo contiene información sobre todos los bloques ocupados y libres en cada momento. La memoria se representa como una lista doble enlazada donde cada bloque apunta al proximo y al anterior. Ademas, para evitar procesamiento inutil, se mantiene en el header de esta lista la ubicación del primer bloque vacio, lo cual acelera la alocacion de memoria nueva.
- Cuando se pide un bloque de memoria el memory manager se fija si tiene suficiente memoria disponible, si es asi le asigna un bloque al proceso que lo pidio y actualiza el puntero al primer bloque libre.
- Cuando se quiere liberar un bloque de memoria basta con setear el flag de free en el header del mismo. Ademas, para mantener consistente la lista se une el nuevo bloque libre a sus vecinos si estos estuvieran libres.

Programas de prueba

El programa de prueba malloc se encarga de testear la función del sistema operativo del mismo nombre, basicamente es un ciclo que mientras logre que se le asigne un bloque de 400 bytes sigue pidiendo memoria e imprimiendo donde empieza el bloque que se le asignó, hasta que finalmente la función malloc le devuelve NULL por falta de recursos y el programa termina avisando de lo sucedido.

El programa de prueba div0 simplemente realiza una division por cero para demostrar el funcionamiento de las excepciones. Cuando se ejecuta lo unico que se verá sera el mensaje “Dividiendo por cero”, seguido por el mensaje impreso por el handler de la excepcion Divide by Zero.