Sistemas Operativos

Trabajo Práctico N°2

Construcción del Núcleo de un Sistema Operativo

26 de Septiembre de 2018

Agustina Osimani 57526

Nicolás Barrera 57694

Ezequiel Keimel 58325

Marcos Lund 57159

Physical Memory Management

Para la implementación del mismo se utilizó la técnica Buddy Allocation. En este algoritmo se divide la memoria en particiones para satisfacer los pedidos de memoria de la manera más adecuada. Cada bloque de memoria en el sistema tiene un orden. Este orden es un entero de 0 a un límite que depende de la cantidad de memoria disponible y el tamaño de las páginas (bloque mínimo).

El tamaño de un bloque de orden n es proporcional a 2^n para que los bloques sean exactamente del doble de tamaño de los bloques que están en un orden menor.

Cuando un bloque es dividido a la mitad, cada uno de los nuevos bloques (sub bloque) se convierte en el único buddy del otro. Un sub bloque puede unirse a su buddy cuando los dos están libres para usar y se convertirán en el bloque del cual fueron divididos.

Se definió la siguiente estructura

```
typedef struct {
  uint64_t memoryAmount; // cantidad de RAM disponible
 void * memoryBaseAddress; // dirección a partir de la cual se asigna memoria
  uint32_t maxOrder;
 uint8_t * pages; // array con el estado de cada página (FREE - USED - SPLIT)
} memoryManager_t;
```

Procesos, Context Switching y Scheduling

Para habilitar multiprogramación se optó por implementar un planificador que maneja una lista de colas, ordenada por prioridad, cada una de las cuales implementa un algoritmo de planificación potencialmente distinto, aunque en el código que se entrega se escribió sólo la implementación de colas Round Robin sin prioridad.

El planificador se ejecuta por cada interrupción del timer tick. En cada una de sus ejecuciones pregunta a la cola que corresponde al thread actualmente en ejecución si su condición de ejecución expiró. Por ejemplo, en el caso de una cola Round Robin esta función de verificación retornará verdadero sólo si el thread actual completó su quantum y posteriormente reiniciará este valor.

Si la función de verificación retorna falso, se incrementará el valor de quantum según corresponda y finalizará la ejecución del planificador.

SI la función de verificación retorna verdadero, se procederá a escanear las colas por orden de prioridades, eligiéndose siempre un proceso de la cola más prioritaria que no esté vacía. Es decir, que si la primera cola no está vacía se ejecutarán procesos de la misma hasta que lo esté, y sólo entonces se explorará la cola siguiente.

Esta implementación se sustenta en la observación del comportamiento del sistema operativo a lo largo de la etapa de desarrollo, eligiéndose este comportamiento por considerárselo el más óptimo y flexible.

En cuanto al concepto de proceso mismo, se lo definió como una estructura compuesta por:

- Un bloque de memoria llamado **heap** que está reservado para uso del usuario por medio, por ejemplo de la función malloc provista por la librería estándar en userland.
- Una lista de los **file descriptors** abiertos y disponibles para el proceso.

- Una lista de los **threads** que engendró, siendo el thread cero el correspondiente a la rutina principal del proceso.
- Un puntero a archivo en el file system que corresponde al **directorio de trabajo** del proceso.
- Un entero en donde se almacena el **valor de retorno** para su posterior recuperación por medio de la función waitpid.
- Un **semáforo** que facilita la implementación de waitpid, congelando al thread en espera hasta que la ejecución de la rutina principal finalice.
- Un **nombre**.
- Un identificador único numérico del proceso (**PID**) que corresponde a la posición del mismo en la tabla global de procesos.

La inicialización de un proceso contempla la adquisición de segmentos de memoria necesarios a fin de definir la estructura anteriormente descrita y la creación de un primer hilo en la posición cero de su lista de threads. A su vez, la ejecución del código de este primer hilo se encuentra englobado por una función wrapper que garantiza la recuperación del valor de retorno así como la llamada a la función *killProcess* para la apropiada liberación de los recursos reservados.

En lo que refiere a los hilos como tal, se los define con un número *tid* correspondiente a su posición en la tabla de threads de su proceso y se define además una propiedad *status* que existe a fin de permitir un seguimiento de la posición del thread dentro de las estructuras del sistema operativo, incluyendo pero no limitándose a colas de scheduling, la cola de procesos dormidos, y colas de espera asociadas a los distintos procesos de sincronización.

Cuando el planificador define efectivamente cambiar el hilo en ejecución, por otro que haya sido adecuadamente elegido según el procedimiento descrito con anterioridad, se procede a generar una copia de respaldo del stack pointer en el instante inmediatamente anterior a la ejecución del código del planificador en su estructura. A continuación, se devuelve el stack pointer correspondiente al hilo que comenzará, o continuará, su ejecución para de esta forma garantizar que la instrucción iretq levantará la información de su contexto a fin de continuar con la ejecución del código de manera transparente para el thread en cuestión.

A fin de poder reutilizar este comportamiento para manejar la inicialización de un thread, se recrea manualmente el stack frame de interrupciones cargándose con los valores apropiados que permitan que la efectiva ejecución de su código no se diferencie de un cambio de contexto de un thread ya en ejecución.

Finalmente, el driver de teclado adquiere el concepto de proceso en primer plano. Solo el proceso en primer plano puede leer de la entrada estándar. Procesos en segundo plano que lo intenten quedarán congelados hasta adquirir el primer plano y continuar con su ejecución.

IPCs

Se decidió implementar un file system para manipular los archivos ya que se deseaba manejar pipes con file descriptors.

Los buffers y los archivos regulares pueden ser abiertos o cerrados para habilitar o deshabilitar las operaciones de escritura y de lectura. Por otro lado, los semáforos y los mutexes tienen operaciones para interactuar con ellos (wait, signal, lock y unlock).

Se definen system calls para realizar todas las operaciones anteriormente mencionadas.

Cada proceso tiene definido un directorio actual (cwd). Para modificar el mismo se debe utilizar la system call *chdir* que recibe el path del directorio al cual se desea cambiar. Los paths pueden ser absolutos (comenzando con /) o relativos. Además, se utiliza "." para referir al directorio actual y ".." para referir al directorio anterior.

La lectura de los buffers es bloqueante. Es decir que los procesos que leen de un buffer vacío duermen hasta que haya información para leer. Por otro lado, si todos los escritores de un buffer lo cierran, se abortan las operaciones pendientes de lectura del mismo. De igual manera, la escritura de los buffers es bloqueante si éste se encuentra lleno.

En cuanto a semáforos se comenzó implementando mutex garantizando exclusión mutua mediante la instrucción atómica XCHG. A partir de ellos, se implementaron semáforos enteros, y posteriormente semáforos binarios como un caso especial de estos primeros. El correcto funcionamiento de los mutex pretende demostrarse por medio del programa *updown* (véase abajo). Dado que los semáforos son fundamentales para el correcto funcionamiento de las rutinas *thread_join* y *waitpid*, y que las mismas responden adecuadamente, se considera que su implementación responde a las necesidades que pretende cubrir sin exponer fallas que el equipo haya podido detectar.

Aplicaciones de User space

Ps

Se implementó el comando *ps* para listar los procesos que se encuentran ejecutándose, mostrando su PID, nombre, estado, cantidad de threads y tamaño de heap y correspondientes y utilizando un buffer que es llenado desde el kernel para imprimir la información correspondiente desde el user space, procurando proteger la información de dichos procesos.

Se implementaron distintos programas que pueden ser ejecutados con comandos de la terminal para probar el correcto funcionamiento de distintos componentes del proyecto.

Test Memory Manager

Para testear el memory manager se implementó un programa (*test-memory-manager*) que pide insertar una cantidad de bytes y te muestra la dirección de memoria donde los alocó. Al final del programa se liberan los bloques que se reservaron a modo de testeo.

El mínimo bloque que puede alocarse es de 4096 bytes. Si se supera ese tamaño, se devuelven bloques con tamaño de potencias de 2.

De esta manera se puede observar que los bloques reservados en memoria no se solapen y que las direcciones de memoria tengan sentido.

Up Down

La funcionalidad de los mutex se verificó en un programa (*updown*) que posee una variable global inicializada en 0 y crea 5 hilos up y 5 hilos down. Los primeros incrementan la variable en 1 1000 veces dentro de un ciclo for, mientras que los segundos la

decrementan. Para modificar la variable, los procesos deben acceder a un mutex y liberarlo después de realizar la modificación.

Una vez que todos los hilos terminaron, se imprime el valor de la variable.

Prod Cons

El programa *prodcons* simula el funcionamiento de un restaurante donde solo hay 20 platos. Los chefs deben esperar a que haya platos vacíos para seguir produciendo, mientras que los camareros deben esperar a que haya platos preparados para servirlos.

Para lograr esto, se utiliza un mutex que permite entrar a la zona crítica, un semáforo entero para manejar el estado de buffer lleno y otro para manejar el estado de buffer vacío.

Arcoiris

El comando *arcoiris* ejecuta un programa que modifica el color de la letra cada cierto intervalo de tiempo. Esto permite demostrar el correcto funcionamiento del scheduler ya que la terminal y éste programa corren simultáneamente.

To Uppercase

El comportamiento de los pipes se puede observar en el programa que se ejecuta con el comando *toUppercase*. Dicho programa abre dos procesos: uno que recibe un texto que introduce el usuario y otro que lo convierte en mayúsculas. El texto introducido se transfiere de un proceso a otro mediante dos buffers, pudiéndose observar de esta manera el bloqueo de pipes para la lectura.

Problemas encontrados durante el desarrollo

- Resultó particularmente desafiante la implementación del *waitpid* y del *thread_join* pues requirieron la utilización de mecanismos de sincronización que se desarrollaron en paralelo con las funciones mencionadas.
- Al desarrollar el programa *prodcons* se utilizaron dos semáforos enteros para limitar el acceso al buffer cuando éste estuviera vacío o lleno. En el caso en que un consumidor estuviera esperando por el semáforo vacío (ya que no puede consumir si el buffer está vacío), si un productor hiciera un signal a ese semáforo y se desbloqueara el consumidor pero el usuario lo matara antes de llegar a ejecutar su código, el semáforo quedaría bloqueado y nunca se consumiría esa posición del buffer. Este problema se hubiera dado de la misma manera en Linux ya que los semáforos no tienen dueño.

  Para solucionar esto, se decidió guardar la cantidad de productores o consumidores que deben morir para que, al correr su código, primero verifiquen que esa variable no es mayor a cero para realizar su función. Si llegara a ser cero, se suicidaría.

Instrucciones de compilación y ejecución

Se debe navegar al directorio que contiene el código fuente y ejecutar el siguiente comando por consola:

```
user@linux:$ cd Toolchain
user@linux:$ make all
```

A continuación se debe regresar al directorio fuente y correr el mismo comando:

```
user@linux:$ cd ..
user@linux:$ make all
```

Finalmente, se debe correr el programa ejecutando:

```
user@linux:$ ./run.sh
```

Bibliografía

http://bitsquid.blogspot.com/2015/08/allocation-adventures-3-buddy-allocator.html