Trabajo Práctico Especial
Sistemas Operativos

Diego Bruno Cilla - 57028
Oliver Balfour - 55177
Sebastian Favaron - 57044
Ximena Zuberbuhler - 57287

Introducción

El objetivo del trabajo práctico especial es continuar modificando el *kernel* del trabajo práctico especial 2, buscando implementar estructuras que le agreguen mejor administración de recursos al sistema. Para ello se decidió implementar los siguientes requerimientos: Pipes, Threads y mejoras al Physical Memory Management.

Diseño

System Calls:

Read	*0*
Write	*1*
Clear	*2*
SetTimeZone	*3*
GetTime	*4*
GetDate	*5*
Echo	*6*
Run	*7*
Malloc	*8*
DrawPixel	*9*
GetResolution	*10*
CreateProcess	*11*
Exit	*12*
Ps	*13*

FreeMutex	*19*
Receive	*20*
Send	*21*
AllocBlock	*22*
DeallocBlock	*23*
CreateSemaphore	*24*
OpenSemaphore	*25*
CloseSemaphore	*26*
WaitSemaphore	*27*
SignalSemaphore	*28*
CreateThread	*29*
RemoveThread	*30*
JoinThread	*31*
CreatePipe	*32*

GetPid	*14*

GetPipe	*33*

KillProcess	*15*
GetMutex	*16*
CloseMutex	*17*
LockMutex	*18*

ClosePipe	*34*
ReadPipe	*35*
WritePipe	*36*

Threads

Un hilo o *thread* es una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un SO. Los hilos representan código ejecutable y sus recursos son administrados por los procesos. Por lo tanto, se tuvieron que modificar las estructuras de los procesos al mismo tiempo que creamos una estructura nueva para definir a los threads.

Las estructuras quedaron definidas de la siguiente manera:

Procesos	**Thread**
• El ID del proceso • El ID del proceso padre • El estado del proceso (puede variar entre corriendo, listo, bloqueado y terminado) • Un flag que indica si corre en primer plano • La descripción del proceso • Un heap y su tamaño correspondiente • Un arreglo de threads • La cantidad de threads	• Un puntero al código • Un stack • Un flag que indica que llegó al final de su código • Una variable que indica el valor del thread que hay que esperar que llegue a su fin al hacer *join_thread*

• El thread que está activo	

La cantidad máxima de threads se decidió aleatoriamente que sea 5 y está definida con la constante MAX_THREADS. Se crearon llamadas a sistemas que cumplan las funcion de creación y eliminación de threads (*createthread* y *removethread*) y tambien una que espera a que otro thread termine (*join thread*).

Pipes:

Para el trabajo anterior ya habíamos trabajado con estructuras que trabajaban como named-pipes que cuentan con un buffer circular en el que si un proceso quiere escribir y está lleno se impide la escritura y se bloquea el proceso (escritura bloqueante), en tanto, para la lectura si se desea realizar la lectura y el buffer se encuentra vacío, se bloquea al proceso hasta que se reconozca algo para leer (lectura bloqueante). De esta manera se resuelve el problema del productor-consumidor para la sincronización de múltiples procesos ya que el productor no añade más mensajes que la capacidad del buffer y el consumidor no intenta tomar mensajes si el buffer está vacío. Se añadieron las llamadas a sistema *CreatePipe*, *WritePipe*, *ReadPipe*, *ClosePipe* para crear, escribir, leer y cerrar pipes respectivamente y GetPipe para obtener el pipe que se está utilizando.

Mejoras al Physical Memory Management:

En el trabajo práctico anterior ya se había incorporado un *buddy page allocator* con las funciones *allocBlock* y *deallocBlock* para poder reservar y liberar páginas respectivamente. Se resolvió a fines prácticos que el tamaño de las páginas sea de 4096 bytes y la cantidad máxima de bloques de 1024 indicados con las constantes *BLOCK_SIZE* y *NUMBER_OF_BLOCKS* respectivamente (total 4MB). Para guardar los bloques se utiliza un árbol almacenado en un array (heap) donde el nodo raíz representa el bloque de 4MB y los nodos más profundos representan los bloques mínimos de 4K. Los estados posibles para cada nodo son FULL, SPLITTED Y EMPTY.

Userland:

$> ps: Lista los procesos corriendo y su información.

$> kill [pids]: Remueve los procesos enviados como argumentos.

$> ipc [message]: Demo del ipc entre 2 procesos. Ofrece un comando para crear un proceso enviante y otro para crear un proceso receptor.

$> prodcons: Demo del problema del productor-consumidor resuelto utilizando semáforos. Ofrece comandos para agregar y remover procesos productores y/o consumidores.

$> while1: Crea un proceso con un ciclo sin final. Se puede utilizar para probar correr un proceso en background y el kill.

$> threads: Demo de la creación, remoción y la espera de threads dentro de un mismo proceso mediante comandos. Se puede ver el estado de cada thread dentro del mismo proceso.

$> pipes: Demo de la lectura y escritura de los named pipes mediante comandos que ofrecen crear un proceso para lectura/escritura señalando el nombre del pipe y el mensaje a enviar, en el caso de querer hacer una escritura. En el caso de la lectura se leen, por defecto, 5 bytes del pipe indicado.

Agregando & antes del nombre de cada módulo se puede correr el proceso en background.
Con ^C (ctrl + C) se llama al sysCall *Exit* que remueve al proceso actual.

Tests:

Se incluye un test básico del buddy memory allocator, en el que se prueba el alloc, 2 alloc consecutivos y un dealloc.

Compilación:

El repositorio cuenta con un archivo makefile para la compilación del código. Se recomienda ejecutar el código sobre un sistema operativo linux que cuente con gcc version 5 y qemu. Asimismo, se incluye un *dockerfile* para facilitar la compilación del mismo.

$> make all

Ejecución:

$> ./run
$> ./runtests