Sistemas Operativos

Trabajo Práctico 2

Grupo 3

Ian Franco Tognetti - 61215 - itognetti@itba.edu.ar
Lautaro Farias - 60505 - lfarias@itba.edu.ar
Axel Castro Benza - 62358 - acastrobenza@itba.edu.ar

Índice

Introducción

El siguiente trabajo práctico tiene como objetivo implementar un memory manager, procesos, scheduling, mecanismos de IPC y sincronización, entre otros, en un kernel simple. Este se obtuvo utilizando como base el TPE de la materia Arquitectura de Computadoras.

En este informe se detallarán las instrucciones de compilación y ejecución; los comandos implementados junto con su funcionamiento; y las decisiones elegidas, modificaciones, problemas y limitaciones que surgieron a lo largo del desarrollo. Finalmente, se comentará una breve conclusión.

Decisiones tomadas durante el desarrollo

En el caso de los semáforos, con el fin de evitar inanición, se implementó una cola de procesos bloqueados. Esto garantiza que ningún proceso retenga el control del semáforo de manera prolongada, pues al regresar a la cola, se le permite a los otros procesos en espera tomar el semáforo. Si un semáforo está en 0 y un proceso ejecuta la operación signal, y existe al menos un proceso bloqueado esperando dicho signal, entonces el valor del semáforo no se modificará. En su lugar, se omitirá este paso y se despertará el proceso bloqueado.

En cuanto al scheduling, el algoritmo implementado por el scheduler es el "Priority-based Round Robin". La implementación viene acompañada de una variable estática que cuenta los ticks desde el último cambio de contexto. Cuando este valor coincide con la cantidad de ticks asignados al proceso en ejecución, se produce un context switch y el contador se reinicia. Las prioridades varían entre 1 y 5, representando la cantidad de timer-ticks asignados a cada proceso.

En cuanto a la clasificación de procesos, se distinguen entre mortales e inmortales (KILLABLE / NOT_KILLABLE). Los procesos mortales pueden ser interrumpidos mediante la combinación de teclas CTRL+C. Por otra parte, los procesos no matables, como el proceso "idle" o la "shell", no pueden ser detenidos ni pausados por el usuario. Se ha implementado la gestión de procesos hijos que están vinculados a sus padres a través de una tabla, permitiendo que el padre espere a que todos sus hijos terminen antes de continuar con su ejecución.

Por otra parte, se ha incorporado una funcionalidad para ejecutar procesos en segundo plano desde la terminal mediante la adición del carácter "&" después del comando deseado.

En la implementación del memory manager, se optó por un enfoque en el que cada bloque asignado incluye un encabezado de 8 bytes. Donde el bit menos significativo del

encabezado indica si el bloque está asignado o no, mientras que los bytes restantes almacenan el tamaño del bloque. Para recorrer la lista de bloques, se navega consultando cada encabezado, obteniendo el tamaño del bloque y saltando esa cantidad de bytes hacia adelante hasta encontrar otro encabezado. La lista finaliza al llegar a un bloque marcado denominado "EOL" (End Of List), que tiene un tamaño de valor 0 y se marca como asignado. Este "EOL" siempre se sitúa al final de la lista y por lo tanto debe actualizarse constantemente a medida que la lista crece. Además, al liberar un bloque de memoria que tiene otro bloque libre adyacente, se realiza una fusión de ambos bloques con el fin de reducir posible fragmentación interna.

En cuanto a la implementación del buddy system, se decantó por una implementación que utiliza un árbol binario que principalmente registra los bloques que están ocupados. Para lograr esto, se reserva una porción del heap donde la información de cada bloque se almacena en un nodo de tipo TNode. Cada uno de estos nodos contiene un dato que indica si el bloque está dividido y otro que indica si está ocupado. Con esta información, se realiza un mapeo con el heap y se devuelve el puntero correspondiente al usuario.

Instrucciones de compilación y ejecución

Dentro de una terminal de linux se deberán correr los siguientes comandos:

1- Para la compilación:

```
$ cd TPSO/
$ docker run -v "${PWD}:/root" --privileged --rm -ti
agodio/itba-so:2.0
$ cd root
$ cd Toolchain
$ make all
$ cd ..
$ make all
```

2- Para la ejecución:

```
$ ./run.sh
```

2- Para la limpieza de archivos binarios:

```
$ make clean
```

Aclaración: por defecto el trabajo se compila utilizando el Memory Manager propio. Si se desea compilar con el Buddy system, asegurarse de descomentar la línea 'MEMORY_MANAGER=-D USE_BUDDY' (línea 21) de la carpeta ~/TPSO/Kernel/Makefile. Por otra parte, puede llegar a suceder que al realizar la compilación o la ejecución del trabajo práctico ocurran errores en los archivos *mp.bin* y/o *./run.sh*. Estos inconvenientes se solucionan otorgándoles permisos a estos.

Comandos Implementados

Al iniciar la shell, los comandos disponibles son los siguientes:

- `help` *(n/a)*: Imprime los comandos y su funcionamiento.

- `divzero` *(n/a)*: Genera una excepción del estilo división por cero.

- `invopcode` *(n/a)*: Genera una excepción del estilo código de operación invalido.

- `time` *(n/a)*: Imprime la hora.

- `pong` *(n/a)*: Ejecuta el juego de pong para dos jugadores.

- `inforeg` *(n/a)*: Imprime los registros en el momento en que se haya tomado una screenshot con la combinación de teclas CTRL + R.

- `clear` *(n/a)*: Limpia la shell.

- `testMM` *(size)*: Ejecuta el test de memoria provisto por la cátedra. El argumento size indica la cantidad de memoria a utilizar en MB,

- `testProcesses` *(quantity)*: Ejecuta el test de procesos provisto por la cátedra. El argumento quantity indica la cantidad máxima de procesos a crear.

- `testPriority` *(n/a)*: Ejecuta el test de prioridades provisto por la cátedra.

- `testSynchro` *(increments, quantity, flag)*: Ejecuta el test de sincronización provisto por la cátedra. El argumento increments determina la cantidad de incrementos a realizar, quantity la cantidad de procesos que realizarán los incrementos y flag determina si se utilizarán semáforos o no (1 = si, 0 = no)

- `cat` *(n/a)*: Imprime el stdin tal como lo recibe.

- `loop` *(n/a)*: Imprime un saludo junto con el pid del proceso tras una determinada cantidad de segundos.

- `wc` *(n/a)*: Cuenta las líneas del stdin e imprime su resultado al recibir la señal EOF.

- `filter` *(n/a)*: Imprime el stdin filtrando las vocales.

- `kill` *(pid)*: Mata el proceso vinculado con dicho pid.

- `ps` *(n/a)*: Imprime todos los procesos que se están ejecutando seguido de su información correspondiente.

- `phylo` *(n/a)*: Ejecuta el problema de los filósofos. Se pueden agregar filósofos al apretar la tecla A y reducir los mismos al seleccionar la tecla R.

- `nice` *(pid, priority)*: Asigna un valor de prioridad a un proceso vinculado con el pid.

- `block` *(pid)*: Cambia el estado de un proceso vinculado a ese pid.

- `mem` *(n/a)*: Imprime el estado de la memoria.

Aclaración: `(n/a)` representa los parámetros que debería recibir cada uno de ellos, que en estos casos es “not applicable”. Por otra parte, cabe destacar que, al presionar las teclas CTRL + D se envía la señal EOF y para matar directamente al proceso y volver a la shell se utiliza CTRL + C.

Limitaciones

En cuanto al scheduler, se restringe la creación de procesos a un máximo de 20, asignando a cada uno un stack de 4096 bytes. Respecto a los semáforos, se limitaron a 50, con la capacidad de bloquear hasta 20 procesos cada uno. Siguiendo con los pipes, se estableció un límite de 20, con una capacidad máxima de memoria de 1024 bytes por pipe. Por último, respecto al comando phylo, se definió un límite de 15 filósofos, teniendo en cuenta la restricción de hasta 20 procesos en ejecución simultánea, como se mencionó anteriormente.

Problemas encontrados durante el desarrollo

Uno de los mayores problemas que tuvimos fue la falta de feedback por parte de nuestro sistema operativo. Un ejemplo claro de esto fue al realizar el armado del stackframe, el sistema operativo no ejecutaba ningún comando, ni siquiera los pertenecientes a la materia de Arquitectura de Computadoras y claramente tampoco dejaba indicios de la causa. Por lo tanto, tras mucho tiempo de relectura de código y pensamiento, logramos dar con el problema, una variable tenía un valor indebido.

Por otra parte, otro desafío fue la dificultad de trasladar los parámetros desde la shell hacía las funciones de los comandos. Por ejemplo, al ejecutar block junto con el valor de un pid, este valor no se transmitía correctamente. Esto se daba puesto que se estaba obteniendo una posición incorrecta del stack. En otra ocasión, al ejecutar testSynchro, el proceso daba como resultado final cero siempre y esto se debía a que se estaban trasladando mal los valores de los parámetros a la función myProcessInc propia del archivo.

Asimismo, un problema más bien general fue la falta de conocimiento acerca de cómo funcionaban algunos comandos. Específicamente nice y block pero esto fue solucionado rápidamente gracias al grandioso manual de usuarios de Linux.

Modificaciones realizadas a las aplicaciones de test provistas

1. testMM: Se ha cambiado el valor de retorno de la función para que sea de tipo void. Además, se ha eliminado el parámetro “int argc” pues este ya es validado previamente en otra función.

2. testProcesses: Se ha cambiado el valor de retorno de la función para que sea de tipo void. Además, se ha eliminado el parámetro “int argc” pues este ya es validado previamente en otra función.

3. testPriority: Se ha cambiado el valor de retorno de la función para que sea de tipo void. Además se han ajustado valores de algunos defines para que sean pertinentes con el sistema operativo.

4. testSynchro: Se ha cambiado el valor de retorno de la función para que sea de tipo void. Además, se ha eliminado el parámetro “int argc” pues este ya es validado previamente en otra función. Por otra parte se han reordenado los parámetros que recibe la función. En este caso, la misma recibe en primer lugar la cantidad de iteraciones a realizar, seguido de la cantidad de procesos que realizan esta iteración y por último el flag que determina el uso o no de un semáforo. Además se añadió al final una cadena de caracteres que muestra el valor esperado (en caso de haber usado semáforos) y el valor final. Este último es el que varía en caso de utilizar o no semáforos.

Conclusión

En resumen, el desarrollo de este trabajo práctico ha reafirmado los conceptos obtenidos en la materia Sistemas Operativos. Desde la implementación de un kernel básico hasta la gestión de recursos y la sincronización de procesos. A su vez, este proyecto ha develado la complejidad y dificultad que lleva el desarrollo de un sistema operativo.