TP2 - Construcción del Núcleo de un Sistema Operativo y estructuras de administración de recursos

Sistemas Operativos 72.11

2021-2Q

Grupo: 3

Integrantes:

Baliarda Gonzalo, 61490

Pérez Ezequiel, 61475

Ye Li Valentín, 61011

Decisiones

-MM: se decidió adaptar la implementación del heap2 de FreeRTOS.

-Buddy: se implementó el sistema buddy basado en heap2.

-Scheduler: algoritmo Round Robin basado en prioridades, con sistema de "envejecimiento" para evitar inanición.

-Procesos: los procesos con prioridad 1 son considerados "foreground", mientras que los de prioridades 2 a 10 serán "background". Al llamar a un proceso desde la shell, puede pasarse como último parámetro el carácter '&', lo cual le dará al proceso prioridad 3 por defecto.
Si no se pasa el '&', el proceso tendrá prioridad 1.
Si se intenta hacer un proceso interactivo en bg, el mismo retornará 0 inmediatamente en cada read() que haga.
La shell no correrá cuando haya otro proceso con prioridad 1 corriendo, y no consideramos la posibilidad de que haya más de 1 proceso en foreground además de la shell.
Al hacer el exit de cada proceso, no se cierran los pipes (a menos que sean pipes que el proceso usa como stdin o stdout) o semáforos que él mismo tuviera abiertos automáticamente, pero si se lo elimina de las listas de espera.

-Semáforos: se simula la implementación de POSIX en la que los sem_open() devuelven un puntero a la estructura que representa el semáforo.

-Teclado: se decidió utilizar la tecla F2 para simular el Ctrl+C (Exit de un proceso) y utilizar la tecla F3 para simular el Ctrl+D (EOF).

-Pipes: se decidió que un proceso que utiliza un pipe solo se pueda bloquear en el mismo en caso de que el otro extremo esté abierto.

-Procesos CAT, WC, FILTER: retornan al hacer un enter con la línea vacía o al presionar F3.

-Tests: el test del memory manager provisto por la cátedra fue adaptado para que funcione. Ver más detalles en la sección de problemas y soluciones.

Instrucciones de compilacion y ejecucion

1. *cd Toolchain; make all*. **Este paso solo se debe realizar la primera vez que se descarga el repositorio.**
2. Para la compilación del proyecto, desde la carpeta inicial se puede compilar para que utilice el memory manager o el buddy de las siguientes formas:
    a. *make MM=BUDDY* (para utilizar el buddy).
    b. *make all* (para utilizar el otro)
3. Por último resta correr el sistema utilizando qemu con el comando: *./run.bat*

Limitaciones

Memory Manager

El memory manager puede asignar hasta 128MB de memoria.

Procesos

No hay limitaciones en la cantidad máxima de procesos. Por defecto, a cada proceso se le asignan 4KB de memoria.

Shell

La shell no es del todo responsiva al leer del teclado, puede que algunas teclas no se registren bien.

Semáforos

Se pueden crear hasta 20 semáforos, y cada uno puede tener hasta 10 procesos esperando al mismo tiempo.

Pipes

Se pueden crear hasta 20 pipes y no se pueden pipear comandos built-in. Los pipes tienen solo 1 proceso que lee y otro que escribe.

Phylo

Se pueden crear hasta 10 filósofos en el comando phylo.

Argumentos

Los argumentos que pueden recibir los procesos y los comandos built-in son de máximo 20 caracteres, y cada uno puede recibir hasta 5 argumentos (si se pipean procesos, la cantidad de argumentos para cada uno baja, máximo 3 en total).

Problemas y soluciones

Memory Manager

Al ejecutar el test del memory manager provisto por la cátedra dentro del SO, el mismo arroja algunos errores. Sin embargo, el test funciona correctamente por fuera del TP.

Tampoco tuvimos ningún problema en todos los testeos que realizamos por nuestra cuenta ni a la hora de realizar implementaciones.
Además, ambos memory manager están basados en heap2, implementación que ya está testeada y fue recomendada por la cátedra.
Probamos diversas soluciones y consideramos todo tipo de posible error, analizando meticulosamente el código tanto del testeo como del memory manager, pero no logramos encontrar ningún error.

Pipes

Al pipear dos procesos en la consola (ej: cat | filter), el proceso que debía leer del pipe no lo hacía correctamente porque la variable “readable” dentro de su fdPipe (la cual determina si lee o no de este), estaba cargada con 0 (cuando debería estar en 1, valor que se le pone al crear el pipe).
No encontramos el motivo del cambio en el valor de readable a lo largo de la ejecución, pero descubrimos que agregando un print justo antes de leer esa variable su contenido volvía a ser 1, y con eso quedó “solucionado”.

Sleep

Al igual que en el pipe, necesitamos agregar un print en esta función, dado que sin el mismo el código tira excepción. Luego cambiamos el print por un “fakePrint” que recibe un string pero retorna inmediatamente sin hacer nada, y el código siguió funcionando.

Argumentos

Al pasar argumentos a cada proceso, nos encontramos con muchos problemas.
Inicialmente intentamos crear un char **argv utilizando alloc, pero los argumentos se terminaban pisando en algún momento de la ejecución.
Para “solucionarlo”, decidimos que cada proceso reciba una matriz constante de argumentos con un tamaño preestablecido, que es el mismo que se usa en las funciones de Userland (5 argumentos, 20 caracteres cada uno). Notar que esto no es una limitación pues ningún comando usa tanta cantidad de argumentos ni tan largos.

Testeos adicionales

Además de los testeos provistos por la cátedra (que están listados en el comando “help”), se implementaron una serie de procesos para realizar algunos testeos básicos de ciertas implementaciones. Entre estos se incluyen:

- p1: imprime los argumentos que recibe y su cantidad.
- p2: se ‘attachea’ a un pipe fijo y escribe un mensaje sobre el mismo. Este pipe luego es leído por p3.
- p3: crea y se ‘attachea’ al mismo pipe que p2 y lee su contenido. El proceso espera 5s a que p2 deje un mensaje, luego del cual lo imprime (si existe).
- p4: crea y se ‘attachea’ a un semáforo determinado y realiza un wait sobre el mismo, quedando bloqueado.

- p5: se ‘attachea’ al mismo semáforo que p4 y realiza un post para liberarlo.

Instrucciones para testeos básicos:

1. Argumentos: correr *p1* con hasta 5 argumentos.
2. Semáforos: correr *p4 &* cuantas veces desee. Luego, chequear el estado de los procesos creados usando *ps* y el estado del semáforo usando *sem*.
   Liberar los procesos de a uno ejecutando *p5.*
3. Pipes: correr *p3 &* y dentro de los siguientes 5s correr *p2.*
4. Pipear procesos: correr *cat* | *filter.*

Biografía

- https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/
- https://www.felixcloutier.com/x86/xchg
- https://github.com/sifive/FreeRTOS-metal/tree/master/FreeRTOS-Kernel/portable/MemMang