Trabajo Práctico 2:

Mecanismos de administración de recursos

Integrantes:

- Mateo Reino
- Jonatan Blankleder
- Agustin German Ramirez Donoso

1 Introducción

El objetivo del tp se puede dividir en 4 objetivos principales: Crear un manejador de memoria, un manejador de procesos, una sincronización dentro de dichos procesos y diferentes aplicaciones de UserSpace pedidas por la cátedra. Fuimos capaces de implementar las 4 funcionalidades aunque tuvimos un problema principal el cual vamos a explicar más adelante

2. Manejo de memoria

2.1 Implementación

Hemos implementado dos formas de manejar la memoria, la primera es la memoria con capacidad de reservar la memoria y liberarla, la segunda forma de manejarla es con el Buddy allocator que consiste en tener listas de bloques de tamaño igual para ser más eficiente.

make usa bump allocator con free (lista simple).

2.1 Limitaciones

make buddy compila un buddy system de 2^k páginas.

Hemos utilizado un bloque de 1 MiB para el heap.

3. Procesos

3.1 Implementación

PCB: Array fijo de 128 entradas (struct pcb); campos : PID, estado, prioridad, punteros de stack, regs salvados.

Scheduling basado en el algoritmo **Round-Robin** con prioridades.

Además, concluimos que la prioridad máxima en nuestro proyecto iba a ser de 7 ya que consideramos que la diferencia entre prioridades 1 y 7 es lo suficiente. Esta diferencia se puede ver corriendo test_priority.

3.2 Limitaciones

Puede haber hasta 128 procesos al mismo tiempo.

4. Semáforos

4.1 Implementación

Buscamos hacer que la implementación de los semáforos sea lo más parecida a la vida real viéndolo desde el lado del usuario. Creamos syscalls para sem_init, sem_post,

sem_wait y sem_destroy. 2 procesos se pueden conectar al mismo semáforo sabiendo su nombre.

Cuando un proceso se bloquea pasa a estar en una cola. Siempre que se hace un sem_post se liberan todos los procesos en la cola de ese semáforo.

4.2 Limitaciones

Puede haber hasta 128 procesos esperando en un semáforo.
Puede haber hasta 32 semáforos.
La longitud máxima del nombre de un semáforo es de 32 caracteres.

5. Pipes

5.1 Implementación

La implementación de los pipes también se buscó que sea lo más parecida a la vida real posible (vista desde el usuario). Además, 2 procesos se pueden conectar a una misma pipe sabiendo su nombre.

5.2 Limitaciones

Tamaño máximo del buffer de la pipe: 1924.
Longitud máxima del nombre: 32 caracteres.
Máxima cantidad de pipes: 32.

6. Aplicaciones

6.1 Implementación

Buscamos diseñar todas las aplicaciones y se las pueden ver corriendo el comando help. Aun así, no se logró implementar todas las aplicaciones mencionadas en la consigna:

- cat: imprime el stdin y no está hecho para conectarse con otros procesos
- wc: lo mismo, siempre termina imprimiendo 1
- sh: No cumple los requisitos relacionados al pipe entre 2 aplicaciones, procesos en bg, ni kill de los procesos mediante ctrl+z.
- phylo.
- ps.
- mem.

7. El problema

Tuvimos 2 grandes problemas: El primero fue que nos costó mucho tiempo ser capaces de poner un proceso a correr. Y el segundo, que **no lo solucionamos**, el cual es simplemente que el proyecto no tiene la suficiente memoria cuando se pone en ejecución, lo que causa que una variable global pierda su valor y no podamos llamar a las diferentes aplicaciones.

Por mail fue mandado un instructivo de cómo replicar el error en una versión vieja del proyecto, pero el error es el mismo y la solución debería ser la misma.

Aun así, a pesar del error seguimos realizando el tp aunque nuestra única posibilidad de debugeo es la compilación y no correr el proyecto y probar los comandos. Esperamos que la cátedra nos de la posibilidad de solucionar cualquier problema que haya debido a la falta de pruebas en el tp debido a que es una solución rápida

2.3 IPC - Pipes Unidireccionales

Se crean con un buffer máximo de 1024, con lecturas y escrituras bloqueantes, idénticas a los semáforos. El nombre de los pipes al igual que los semáforos tienen como longitud máxima 32.

2.4 Semáforos

Diseño inspirado en los semáforos de Linux. Con down/up sin busy wait. sem_wait/post con spinlock + wait-queue. Bloquea al hilo y planifica.

Con una cantidad máxima de semáforos de 32, con un nombre de longitud máxima de 32 caracteres. La de waiters en un semáforo es de 128.

3 Compilación y ejecución

Para compilar y ejecutar, se encuentran hechos archivos bash para crear el contenedor (create-container.sh), compilar (compile.sh), limpiar los archivos .o (clean.sh) y ejecutar los binarios/imágenes (run.sh). Si está usando WSL probablemente sea necesario hacer dos2unix a los mismos para que se ejecuten correctamente

Comandos a ejecutar (en orden):

3.1 Crear contenedor

./create-container

3.2 Compilar kernel y userland

./compile.sh

3.3 Ejecutar en QEMU

./run.sh

4. Instrucciones para demostrar el funcionamiento de los requerimientos

Están programados test para probar el funcionamiento de la creación de procesos, el funcionamiento de semáforos, y la creación y uso de pipes. Estos son los comandos que están prefijados por “test” que se pueden ver al hacer help.

5. Manual

Tuvimos que borrar el manual de nuestro codigo por problemas de memoria asi que lo ponemos aca:
"help: Displays a basic help with all commands. Use to get a list of them but use man to get a more detailed description\n\
of each command\n",
"registers: Displays the latest backup of registers. Press the esc key at ant time to back them up\n\
then write this command to see them\n",
"time: Displays time and date on GTM -3\n",
"eliminator: starts the eliminator game. Play alone or with a friend to get the highest\n\
score possible\n",
"echo [string]: prints the [string] argument in the display\n\
on later iterations of the OS\n",
"clear: clears the display\n",
"change_font: Changes the current format from small to big or from big to small\n",
"nano_song: autism goes BRRRRRR\n",
"test_zero_divition: Tests the Zero division exception\n",
"test_invalid_opcode: Test the Invalid Opcode exception\n",
"test_malloc: starts the malloc test\n\
return value: returns 0 if the test was passed, any other number is an error.\n\
look at the code to see where it broke\n",
"man [command]: displays the manual for [command]. Has useful info about how and why to use it\n\
Also, some things may not be implemented yet and may cause unexpected behavior. If that happens, the entry\n\
inside man will have \"TODO: Not implemented\"",
"todo: don't know what to do? run this command and you'll be given a random task in the TODO list\n",
"functions: Displays every page inside the manual. Useful for testing and to play around\n",

"mini_process: Creates a new process according to simpleProcess.c\n",
"test_priority: There will be 2 processes created, process 1 has priority 1 and process 2 has priority 7 (max).\n\
they will be created in that order. The 2 are in a file in Userland called priorityTest.c\n\
check what happens to adress if the priority system is working.\n",
"sh\n", "mem\n", "ps\n", "loop\n", "kill\n", "nice\n", "block\n", "cat\n", "wc\n", "filther\n",
"phylo\n",

// Useful

"malloc:\n\
use: void* malloc(uint64_t size)\n\
description: the malloc function allocates [size] bytes and returns a pointer to the allocated memory.\n\
return value: returns a pointer to the allocated memory or NULL in case of error or no space available\n",
"realloc:\n\
use: void* realloc(void* pnt, uint64_t size)\n\
description: the realloc function allocates [size] bytes and returns a pointer to the allocated memory. The memory is the same from the start until the end of pnt\n\
return value: returns a pointer to the allocated memory or NULL in case of error or no space available\n",
"calloc:\n\
use: void* calloc(uint64_t size)\n\
description: the malloc function allocates [size] bytes and returns a pointer to the allocated memory. It sets every value inside of it to 0\n\
return value: returns a pointer to the allocated memory or NULL in case of error or no space available\n",
"free:\n\
use: void free(void* pnt)\n\
description: free frees the memory that was previously allocated by pnt via malloc() or calloc(). Nothing happens if pnt is NULL\n\
return value: free doesnt return anything\”",
"createProcess:\n\
use:\n\
description: \n\
return value: \n",

"getPriority:\n\
use: getPriority(pid_t pid)\n\
description: gets the priority of the sent pid\n\
return value: returns the priority of the sent pid. Returns -1 if pid is out of bounds. Runtime error if no process exists with that pid\n",
"setPriority:\n\
use: setPriority(pid_t pid, int newPriority)\n\
description: sets a new priority for the sent pid. Higher numbers mean higher priorities.\n\