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

**Integrantes:**

- Banfi Malena, 61008
- Catino Kevin, 61643
- Fleischer Lucas, 61153

**Grupo: 1**

**Fecha de entrega:** 30/5/2022

# Decisiones tomadas durante el desarrollo

## Manejo de git

Decidimos que un integrante del grupo solamente haga los commits al repositorio, y trabajar a través de la extensión de Visual Studio Code llamada *LiveShare* para poder trabajar de forma colaborativa y tener una buena división de tareas.

## Mensaje de bienvenida

El mensaje que se muestra en pantalla “Welcome to SOS” queriamos aclarar que las siglas “SOS” hacen referencia a Sistemas Operativos/Operative System, reutilizando la letra “O”. No es un llamado de auxilio, sino un simple juego de palabras.

## Scheduling

Se decidió utilizar un algoritmo de Round Robin que implementa prioridades a través de un valor de quantum dentro de cada bloque de proceso, cuyo valor depende del nivel de prioridad que posee. De esta forma, los procesos con mayor prioridad corren por un mayor tiempo. Las prioridades van del 1 al 5  (las prioridades más altas son las de menor valor).

## Memory Management

### MemoryManagerADT.c

Se decidió adaptar la implementación del heap2 de FreeRTOS.

### Buddy

Se implementó el sistema buddy basado en heap2.

## Semáforos

La llamada a al sem_open devuelve el puntero a un tipo abstracto de datos correspondiente al del semáforo. Para cada llamada a función que involucre la manipulación del mismo, es necesario pasar como parámetro dicho valor de tipo Semaphore.

## Procesos

Los procesos con prioridad 1 son considerados “foreground”, mientras que los de prioridades 2 a 5 serán “background”. Al llamar a un proceso desde la shell, puede pasarse

como último parámetro el carácter ‘&’, lo cual le dará al proceso prioridad 1 por defecto. Si no se pasa el ‘&’, el proceso tendrá prioridad 2. La shell no correrá cuando haya otro proceso con prioridad 1 corriendo, y es únicamente rehabilitada cuando todos los otros procesos del foreground terminene su ejecución.

Al hacer el `sys_exit()` de cada proceso, se cierran los file descriptors correspondientes a su STDIN y STDOUT en caso de que éstos formen parte de un pipe (para los file descriptors de entrada por teclado y salida por consola, consideramos éstos un caso especial y nunca son cerrados).

**Se puede forzar el exit de un proceso que haya tomado el foreground de la shell presionando la tecla ‘ESC’.**

## Comando ‘cat’

El comando `cat` puede ser invocado sin argumentos, en cuyo caso imprimirá lo ingresado por entrada estándar en tiempo real, pero sino, es posible llamarlo con un string de argumento que puede contener los caracteres ‘\n’ para indicar un salto de línea. Hay que tener en cuenta que cada palabra espaciada representa un argumento diferente, y el límite de argumentos definido es de 6.

## Pipes

Los pipes se implementaron a través de una estructura en C que alberga un buffer, un puntero de escritura y otro de lectura, y booleanos que indican si ambos lados del pipe están o no en uso. Al crear un pipe con el método `sys_create_pipe()` es necesario pasar por parámetro los dos descriptores que serán los extremos de lectura y escritura del mismo. En el caso de los file descriptors correspondientes a entrada por teclado y salida por consola, no es necesario hacer una llamada a `sys_create_Fd()`, sino que basta con pasar las constantes ya definidas llamadas `STDIN` y `STDOUT` .

# Limitaciones

- Los argumentos de un proceso deben ser menos de 7 y su longitud no debe ser mayor a 50 caracteres.
- El tamaño de la memoria de un proceso es de 4KB.
- El memory manager puede reservar hasta 200MB de memoria.
- Pueden haber hasta 20 semáforo, y cada uno puede tener hasta 20 procesos esperando.
- Pueden haber hasta 20 pipes, y cada uno de ellos puede tener un único proceso esperando en el otro extremo, ya sea para leer o escribir.

- Pueden haber hasta 20 filósofos simultáneos dentro de la interfaz del comando `phylo`.

# Pasos para demostrar el funcionamiento

## Pipes

- `cat hola\n como\n estas | wc` y se debería observar que el resultado es 3.
- `help | filter` y se debería obtener en pantalla la salida del comando `help` sin todas sus vocales.
- `cat hola\n como \n estas | cat` y se deberían 3 líneas, con el contenido “hola”, “como” y “estas” respectivamente.
- `loop 5 | filter &` y luego correr `pipe` para observar el estado del pipe que utilizan ambos procesos para comunicarse.
- `cat | filter` y observar como el comportamiento es el mismo que el de hacer `filter` con la diferencia de que existe una comunicación entre dos procesos mediante un pipe.

## Semáforos

- `syncTest` y observar que dos de los procesos finalizan con el contador en 0 (son los últimos procesos en modificar el valor de la variable).
- `syncTest &` y luego correr `sem` para observar el estado del semáforo junto a los procesos esperando por el mismo.
- `syncTest no-sem` usa el mismo programa de prueba con la diferencia de no incluir semáforos. Observar que los valores de todos los procesos dan distinto de 0.
- `phylo` y seguir las instrucciones en pantalla para agregar y quitar filósofos de la mesa.

## Memory Manager

- Correr `memStatus` inicialmente para ver la cantidad del heap usado. Luego correr `mmTest 40 &` y `memStatus` para observar que el testeo de memoria efectivamente haga uso del heap, reduciendo el espacio disponible. Verificar que luego de correr `mmTest`, la memoria se haya liberado correctamente viendo que los valores obtenidos en `memStatus` son los mismos que al inicio.
- Correr `mmTest 90` y observar que se obtiene un puntero NULL que indica que no ha sido posible reservar la memoria necesaria.

## Procesos

- Correr `loop 5 &` y luego `ps` para observar los procesos corriendo.
- Correr `loop 5 &` y luego `kill <pid>`, observando que el loop finaliza.
- Correr `loop 5`, luego presionar la tecla ‘ESC’, y finalmente correr `ps`, observando que el proceso es efectivamente terminado.
- Correr `loop 5 &`, luego `ps` (observar el nivel de prioridad seteado en 2 del proceso `loop`). Mientras corre el loop, correr `nice <pid> 3` para modificar el nivel de prioridad. Finalmente, correr `ps` para observar que el nivel de prioridad efectivamente cambió.
- Correr `loop 5 &`, esperar unos segundos y luego correr `block <pid>`. Observar que el proceso se bloquea. Verificar que el proceso está bloqueado corriendo `ps`.

# Compilación y ejecución

Para ejecutar el programa, estando en la “TP2-SO” hay dos alternativas:

- La primera es ejecutando el script de compilación llamado `compile.sh`, el cual compila y linkedita el proyecto con docker y toma como default al BUDDY.

  NOTA: Si se compila ejecutando `./compile.sh BUDDY` también funcionará y tomará el BUDDY como predeterminado.

- La segunda es ejecutando el script de compilación llamando `./compile.sh HEAP2`, el cual compila y linkedita el proyecto con docker y toma como predeterminado al HEAP2.

# Inconvenientes durante el desarrollo

- Al hacer uso de los pipes a través de un comando de shell, ocurría que uno de los dos procesos directamente no aparecía en la lista de procesos del scheduler, y no entendíamos por qué. El problema resultó ser que, dado que la shell se bloqueaba inmediatamente cuando otro proceso venía a ocupar el foreground, la línea que creaba el segundo proceso del pipe nunca llegaba a ejecutarse. Solucionamos el problema eliminando el forzado del scheduler desde la función del kernel, y haciendo que la shell haga un `sys_yield()` al terminar de crear los procesos necesarios para luego ceder el foreground.
- Al querer implementar una señal que elimina todos los procesos del foreground dejando únicamente a la shell, tuvimos el inconveniente de que la función sólo eliminaba al primer proceso del foreground en lugar de a todos, y debido a que la shell se desbloqueaba cuando ningún otro proceso ocupaba el foreground, el sistema quedaba en un estado de bloqueo total.

- Al querer implementar ciertas syscalls de funciones con muchos parámetros (como es el caso del `sys_create_process()`), nos pasaba que ciertos argumentos de la función no tomaban el valor que se pasaba por parámetro, y no lográbamos comprender por qué. Resultó ser que el manejo de syscalls que habíamos desarrollado para el SO en Arquitectura de las Computadoras, únicamente soportaba 3 argumentos, e incluso pisaba uno de los registros donde recibía argumentos para pasar el número de la syscall a la función `sysCallHandler()` desarrollada en C. Tuvimos que desarrollar una función de Assembler aparte para el pasaje del valor de la syscall, y así permitir que hasta 6 parámetros sean admitidos en cada función de syscall.
- Tuvimos problemas al implementar el `sys_exit()` dado que no forzábamos el cambio de contexto luego de liberar la memoria del proceso, y entonces los procesos continuaban su ejecución en lugar de borrarse del scheduler inmediatamente, lo que llevaba a una excepción general.
- En una parte de todo el código hacíamos incorrectamente `allocMemory(sizeof(* struct ...))`, lo cual terminaba reservando un espacio menor al necesario para la estructura. Esto generaba que en reservas de memoria siguiente se pisen valores de memoria que nosotros estábamos utilizando.
- En un comienzo implementábamos incorrectamente el foreground de la shell directamente haciendo una llamada explícita a la función correspondiente en lugar de crear un proceso nuevo. Esto no permitía que sea posible terminar la ejecución de un proceso de foreground para recuperar el control por parte de la shell ya que es la misma shell misma la que se encuentra ejecutando.

# Modificaciones de los tests

- Para el test de memoria, se modificó el mismo de forma tal que el argumento recibido sea un valor entre 0 y 100 representando el porcentaje de la memoria total a testear en lugar del tamaño explícito en bytes. Fuera de eso, el test es el provisto por la cátedra.
- Para el test de sincronización, se unificaron los programas de test con y sin sincronización para que usen o no los semáforos en base a los argumentos recibidos dentro del proceso.
- En todos los tests se eliminaron los ciclos infinitos de forma tal que los programas puedan ser llamados en forma de comandos desde la shell, además se agregaron prints explícitos para que el usuario sepa lo que el test ejecuta en cada etapa.
- Se intercambiaron los nombres de las funciones que traía la interfaz de testeo para que usen directamente las funciones definidas en el sistema operativo.

# Justificaciones de análisis estático y dinámico

## PVS Studio

Se obtiene lo siguiente:

```
/home/khcatino/SO/TP2-SO/Bootloader/BMFS/bmfs.c 238 note  V576 Incorrect format. Consider check
/home/khcatino/SO/TP2-SO/Bootloader/BMFS/bmfs.c 254 note  V576 Incorrect format. Consider check
/home/khcatino/SO/TP2-SO/Bootloader/BMFS/bmfs.c 254 note  V576 Incorrect format. Consider check
/home/khcatino/SO/TP2-SO/Bootloader/BMFS/bmfs.c 369 note  V584 The 'diskSize' value is present
/home/khcatino/SO/TP2-SO/Bootloader/BMFS/bmfs.c 408 note  V576 Incorrect format. Consider check
/home/khcatino/SO/TP2-SO/Bootloader/BMFS/bmfs.c 452 note  V575 The potential null pointer is pa
/home/khcatino/SO/TP2-SO/Bootloader/BMFS/bmfs.c 482 note  V575 The potential null pointer is pa
/home/khcatino/SO/TP2-SO/Bootloader/BMFS/bmfs.c 552 err V595 The 'disk' pointer was utilized be
/home/khcatino/SO/TP2-SO/Bootloader/BMFS/bmfs.c 689 note  V576 Incorrect format. Consider check
```

Todos los errores son respecto a archivos del Bootloader que no fueron desarrollados por el grupo, por lo que decidimos mantenerlo como vino provisto por la cátedra.

## Cppcheck

Se obtiene lo siguiente:

```
[Bootloader/BMFS/bmfs.c:236]: (warning) %d in format string (no. 2) requires 'int' but the argu
[Bootloader/BMFS/bmfs.c:252]: (warning) %lld in format string (no. 2) requires 'long long' but
[Bootloader/BMFS/bmfs.c:687]: (warning) %lld in format string (no. 1) requires 'long long' but
[Bootloader/BMFS/bmfs.c:260] -> [Bootloader/BMFS/bmfs.c:262]: (performance) Buffer 'DiskInfo' i
[Bootloader/BMFS/bmfs.c:274]: (style) The scope of the variable 'writeSize' can be reduced.
[Bootloader/BMFS/bmfs.c:449]: (style) The scope of the variable 'percent' can be reduced.
[Bootloader/BMFS/bmfs.c:723]: (style) The scope of the variable 'tfile' can be reduced.
[Bootloader/BMFS/bmfs.c:724]: (style) The scope of the variable 'tint' can be reduced.
[Bootloader/BMFS/bmfs.c:755]: (style) The scope of the variable 'tfile' can be reduced.
[Bootloader/BMFS/bmfs.c:756]: (style) The scope of the variable 'tint' can be reduced.
[Kernel/LibC/lib.c:86]: (style) The scope of the variable 'temp' can be reduced.
[Toolchain/ModulePacker/main.c:111]: (style) The scope of the variable 'read' can be reduced.
[Userland/SampleCodeModule/_loader.c:11]: (style) The function '_start' is never used.
[Userland/SampleCodeModule/LibC/libc.c:182]: (style) The function 'hexaToInt' is never used.
[Kernel/kernel.c:44]: (style) The function 'initializeKernelBinary' is never used.
[Kernel/LibC/lib.c:55]: (style) The function 'intToHexa' is never used.
[Kernel/IDT/irqDispatcher.c:12]: (style) The function 'irqDispatcher' is never used.
[Userland/SampleCodeModule/LibC/libc.c:95]: (style) The function 'isAlpha' is never used.
[Userland/SampleCodeModule/LibC/libc.c:69]: (style) The function 'isDigit' is never used.
[Userland/SampleCodeModule/LibC/libc.c:90]: (style) The function 'isWhiteSpace' is never used.
[Kernel/IDT/sysCallDispatcher.c:20]: (style) The function 'loadSyscallNum' is never used.
[Kernel/LibC/naiveConsole.c:94]: (style) The function 'ncPrintBin' is never used.
[Kernel/IDT/time.c:13]: (style) The function 'seconds_elapsed' is never used.
[Kernel/utilities/scheduler.c:261]: (style) The function 'switchProcess' is never used.
[Kernel/IDT/sysCallDispatcher.c:27]: (style) The function 'sysCallDispatcher' is never used.
[Userland/SampleCodeModule/LibC/libc.c:74]: (style) The function 'toLower' is never used.
[Userland/SampleCodeModule/LibC/libc.c:82]: (style) The function 'toUpper' is never used.
(information) Cppcheck cannot find all the include files (use --check-config for details)
```

En las primeras líneas, las advertencias que se obtienen son respecto al scope de ciertas variables definidas en los archivos del Bootloader (que vino como parte de la implementación el sistema operativo “Barebones”) y a los tipos de datos de variables. Decidimos no modificarlo dado que esa parte del código fue brindada ya funcional por la cátedra.

El resto de las advertencias son respecto a la no utilización de ciertas funciones. En la mayoría de los casos se trata de funciones de la librería estándar de C, que se implementó durante el desarrollo del TPE de Arquitectura de las Computadoras y consideramos que podrían incluirse como parte del Sistema Operativo para utilidad del usuario que vaya a usarlo.

Por otro lado, otras funciones como `sysCallDispatcher()` y `switchProcess()` son utilizadas debidamente en el código Assembler correspondiente, por lo que la advertencia es errónea.

## Valgrind

Se obtiene la siguiente advertencia:

```
HEAP SUMMARY:
==28708==     in use at exit: 128,801 bytes in 523 blocks
==28708==   total heap usage: 1,035 allocs, 512 frees, 165,656 bytes allocated
==28708==
==28708== 8 bytes in 1 blocks are definitely lost in loss record 61 of 312
==28708==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd
==28708==    by 0x198523: xmalloc (in /usr/bin/bash)
==28708==    by 0x19199E: set_default_locale (in /usr/bin/bash)
==28708==    by 0x136C8A: main (in /usr/bin/bash)
==28708==
==28708== LEAK SUMMARY:
==28708==    definitely lost: 8 bytes in 1 blocks
==28708==    indirectly lost: 0 bytes in 0 blocks
==28708==      possibly lost: 0 bytes in 0 blocks
==28708==    still reachable: 128,793 bytes in 522 blocks
==28708==         suppressed: 0 bytes in 0 blocks
```

Parece ser que el leak de memoria ocurre en un proceso del Valgrind en sí, y no sabemos cómo solucionarlo.