



Instituto Tecnológico de Buenos Aires

## **TP2: Kernel**

72.11 - Sistemas Operativos

Grupo N° 7

### **Integrantes:**

Cortés Teyssier, Manuel - 64159

Fernández Flores, Valentina - 64142

Othatceguy, Manuel - 64671

Tiscornia, Gregorio - 64087

### **Docentes:**

Godio, Ariel

Aquili, Alejo Ezequiel

Gleiser Flores, Fernando

Beade, Gonzalo

Mogni, Guido Matías

Fecha de entrega: 9 de Junio 2025

## Índice

<b>Introducción.....</b>	<b>2</b>
<b>Instrucciones de compilación y ejecución.....</b>	<b>2</b>
<b>Decisiones Desarrollo y funcionamiento.....</b>	<b>2</b>
Physical Memory Management.....	2
Procesos, Context Switching y Scheduling.....	4
Sincronización.....	5
IPC.....	7
<b>Limitaciones.....</b>	<b>7</b>
<b>Conclusión.....</b>	<b>8</b>
<b>Citas de fragmentos de código reutilizados de otras fuentes.....</b>	<b>9</b>

## Introducción

El presente trabajo consistió en la construcción del núcleo de un sistema operativo, tomando como base el trabajo desarrollado en la materia Arquitectura de Computadoras. Éste consta principalmente de cuatro funcionalidades, siendo estas el manejo de memoria, el manejo de procesos, la sincronización y la comunicación entre ellos.

## Instrucciones de compilación y ejecución

El repositorio contiene tres archivos de tipo *bash script*. A continuación se detallan sus características:

- *init.sh*: crea el contenedor de *Docker* “*SO\_Docker*” que hace un *binding* del *root* con la carpeta del trabajo práctico, es importante correr este archivo parado dentro de la carpeta sino no va a funcionar como corresponde.
- *compile.sh*: va a utilizar el contenedor creado con el archivo anterior. Puede recibir un sólo parámetro que es “*-buddy*” o “*-bitmap*”, por *default* se utiliza “*-buddy*” si es que no hay tal argumento o si lo hay pero es inválido.
- *run.sh*: corre con *Qemu* la imagen creada por *compile.sh*, y puede recibir un argumento “*-d*” que habilita la depuración con *GDB*, en caso de que sea inválido o no esté presente el argumento, por *default* corre sin opciones de depuración. Si se quiere correr con las opciones de depuración, es conveniente correrlo como  
**\$> ./run.sh -d 2>&1 | grep “v=”** para poder ver en cada línea cada interrupción que sucede en tiempo real.

## Decisiones Desarrollo y funcionamiento

A lo largo del desarrollo se intentó mantener una escalabilidad a la estructura inicial a medida que se iban desarrollando los principales pilares de este proyecto detallados a continuación.

### Physical Memory Management

En este trabajo se implementaron dos manejadores de memoria, un memory manager elegido por el grupo, que soporta la liberación de memoria, en nuestro caso el *bitmap system*, y un *buddy system*. Ambos cumplen con una interfaz común que permite intercambiarlos, definiendo el mecanismo a utilizar en tiempo de compilación, como se mencionó anteriormente. Éstos mecanismos operan sobre un rango de

memoria, a partir de una dirección fija llamada *HEAP\_START\_ADDRESS* (arbitrariamente es 0x600000).

Para el *buddy system* se definieron dos estructuras principales, por un lado *BuddyBlock*, la cual representa cada bloque de memoria, incluyendo su nivel, estado (libre, asignado o dividido), y punteros a los bloques adyacentes en la lista. Por otro lado utilizamos la estructura *MemoryManager* la cual contiene la memoria gestionada, con el puntero al inicio del *heap*, el tamaño total, la memoria utilizada, un arreglo con listas de bloques por nivel y el estado general del sistema. El funcionamiento del *buddy system* se basa en las funciones *createMemoryManager*, *allocMemory* y *freeMemory*. La inicialización configura la memoria y los niveles. La asignación busca bloques libres y los divide si es necesario mediante *splitBlockToLevel*, mientras que la liberación marca el bloque como libre y fusiona memoria usando *mergeBlocks*.

El segundo manejador de memoria que implementamos es el *bitmap system* el cual elegimos ya que soporta la liberación dinámica de memoria. Para este sistema, se definió una estructura principal la cual almacena el puntero al inicio del *heap*, la cantidad total de bloques disponibles, el número de bloques usados, un puntero al *bitmap* que indica la posición del bloque y un índice que indica la posición actual. En el caso del *bitmap system* se inicializa la estructura con la función *createMemoryManager*, la función *allocMemory* asigna bloques contiguos marcándolos como usados en el *bitmap*, y *freeMemory* que permite liberar la memoria asignada actualizando el *bitmap*.

Para verificar el funcionamiento del manejo de memoria, se puede ejecutar el comando *test\_mm*, pasando como parámetro la cantidad máxima de memoria que el *test* intentará reservar. En la terminal se puede observar la ejecución del test, como se muestra en la siguiente figura.



```
u$> test_mm 100
Iniciando test de gestion de memoria...
max_memory: 100
test_mm: finished successfully
Proceso 2 terminado con estado 0
```

Figura 1: Ejecución del comando *test\_mm* en la shell

Además, en ambas implementaciones utilizamos la función *getMemoryInfo* para obtener la información del estado de la memoria. Se puede ejecutar el comando

*memInfo* en la *shell* para que se imprima por pantalla el tamaño total de la memoria, el utilizado, y el libre, como se muestra en la siguiente figura.

```
u$> memInfo
Estado de memoria:
Memoria total: 530579456 bytes
Memoria usada: 54016 bytes
Memoria libre: 530525440 bytes
Proceso 2 terminado con estado 0
```

Figura 2: Ejecución del comando *memInfo* en la *shell*

### Procesos, Context Switching y Scheduling

Para este apartado se inició por una estructura básica de un proceso, siendo sus campos iniciales, el nombre del proceso, su argumentos (*argc*, *argv*), su código primario (la función que ejecutan), y no menos importante su *stack* y *base pointer*. Estos con la intención de poder ir *testeando* a medida que progresamos. A medida que avanzó el desarrollo, y surgían una necesidad de más campos para funcionalidades, se incorporaron los siguientes campos como *pid*, *parentPid*, *waitingForPid*, prioridad, estado, *entryPoint*, valor de retorno, *foreground* y *file descriptors*.

Una vez que teníamos una estructura básica, se desarrolló un Scheduler de carácter “*dummy*”, el cual se trataba de una lista doblemente anidada y circular de procesos en la que iban rotando los procesos a partir de un determinado *Quantum*, el cual se restaba con cada iteración. Así, buscando conseguir un *Context Switch* exitoso. Para esto, cada vez que llegara la interrupción del *TimerTick*, utilizamos su handler de la interrupción para llamar a *schedule()*. En caso de que al proceso actual se le haya acabado el *Quantum* se guarda toda la información propia de un proceso en el *stack* utilizando *processStackFrame*. Vale mencionar que nos detuvo bastante los avances con el proyecto, ya que sin un *Scheduler* funcional no teníamos forma de probar el resto. Sin embargo, una vez que consolidamos un modelo viable, para no retrasarse, iniciamos con el resto de secciones de este proyecto pero con el objetivo de volver a desarrollar un *scheduler* más eficiente.

A medida que salieron más necesidades de interactuar con los procesos existentes, optamos por el desarrollo del *Scheduler* y *ProcessManagerADT* actual. Siendo la estructura *ProcessManager* el componente central encargado de gestionar los procesos del sistema operativo. Contiene punteros y *queues* que permiten organizar y controlar los distintos estados de los procesos:

- *foregroundProcess* y *currentProcess* identifican el proceso en primer plano y el que está en ejecución, respectivamente.
- *readyQueue* y *blockedQueue* manejan procesos listos para ejecutarse y procesos en espera de recursos.
- *blockedQueueBySem* gestiona los bloqueos por semáforos, permitiendo una mejor sincronización.
- *zombieQueue* almacena procesos finalizados que aún deben ser limpiados.
- *idleProcess* garantiza que la CPU nunca quede inactiva.

Asimismo el *Scheduler* utiliza a *ProcessManager* como la “base de datos” central de procesos, delegando en él todas las operaciones de alta, baja, cambio de estado y consulta de procesos, mientras se encarga de la lógica de planificación y cambio de contexto.

Para verificar el funcionamiento de los procesos, *context switch* y *scheduling*, se puede ejecutar desde la shell los siguientes comandos: *test\_prio* para verificar las prioridades de los procesos en tiempo de ejecución, *test\_processes* para verificar la correcta creación, bloqueo, desbloqueo y finalización de los procesos, y *ps* para listar los procesos, cada uno con su respectivo pid, estado, prioridad, parent pid, foreground, entre otros, como se ve en la siguiente figura.

PROCESS	PID	STATE	PRI	PPID	WAIT	FG	REGISTERS
idle	0	READY	6	none	none	no	RSP: 0x602F50 RBP: 0x603018 Entry: 0x1000BD
shell	1	BLOCKED	0	0	2	yes	RSP: 0x60AC80 RBP: 0x60B018 Entry: 0x400000
ps	2	RUNNING	1	1	none	yes	RSP: 0x60CF70 RBP: 0x60D018 Entry: 0x4021CD

Figura 3: Ejecución del comando ps en la shell

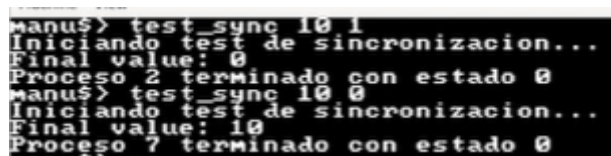
### Sincronización

La sincronización entre procesos involucró el desarrollo de un mecanismo de semáforos, permitiéndole a los procesos acceder a recursos con el objetivo de evitar condiciones de carrera y garantizando la correcta ejecución de los procesos. A continuación se detalla su implementación y funcionamiento.

Para implementar los semáforos se decidió utilizar un arreglo de una cantidad de semáforos fija. Cada uno cuenta con una variable que contiene el valor actual del

semáforo, una variable de exclusión mutua, una para indicar si el semáforo está actualmente en uso y una cola que almacena los procesos que están bloqueados esperando. Las funciones implementadas fueron *semManager* para inicializar el sistema de semáforos, *semInit* para inicializar un semáforo con un valor determinado, *semOpen* y *semClose* que permiten acceder o liberar un semáforo ya creado, *semWait* y *semPost* que realizan las operaciones básicas sobre los semáforos, internamente utilizan *wait* y *post* las cuales manejan las colas de procesos bloqueados y desbloqueados. Éstas últimas dos funciones, *wait* y *post*, utilizan *acquire* y *release* para asegurar exclusión mutua, al mismo tiempo *acquire* utiliza *xchg* para implementar un spinlock que garantiza atomicidad.

Para verificar el funcionamiento de los semáforos, se puede ejecutar *test\_sync* en la terminal, pasando por parámetros el número de iteraciones por proceso y si se quiere habilitar los semáforos (1 si se quieren utilizar y 0 en caso contrario). En la terminal se puede observar la ejecución como se muestra en la siguiente figura.



```
manu$> test_sync 10 1
Iniciando test de sincronizacion...
Final value: 0
Proceso 2 terminado con estado 0
manu$> test_sync 10 0
Iniciando test de sincronizacion...
Final value: 10
Proceso 7 terminado con estado 0
```

Figura 4: Ejecución del comando *test\_sync* en la shell

Si se ve “*Final value: 0*” al correrlo con el parámetro “*use sem*” en 1 es porque el test pasa correctamente, caso contrario el test falla.

Al correrlo sin el parámetro mencionado anteriormente es esperable que el valor varíe, ya que no hay sincronización.

Por otro lado, se implementó el problema de los filósofos comensales, en el cual cada filósofo es un proceso que alterna entre pensar y comer requiriendo acceso exclusivo a los tenedores, con el objetivo de evitar *race conditions* y *deadlocks*. Para esto utilizamos un *mutex* global que protege el acceso a la estructura del estado compartido, y un semáforo por filósofo, cuando un filósofo quiere comer verifica si los filósofos vecinos están comiendo, y si es así se bloquea en su semáforo. Una vez que los recursos estén disponibles, se desbloquean. Para ejecutar esta funcionalidad en la terminal se utiliza el comando *philo* y además se pasa por parámetros la cantidad de filósofos con la que se desea comenzar la ejecución. La cantidad mínima de filósofos es 1 y la máxima es 10.

Además permite agregar o quitar filósofos en tiempo de ejecución mediante los comandos 'a' y 'e', y también finalizar la ejecución con 'f'. En la terminal se puede observar la ejecución como se muestra en la siguiente figura.

```

s$> phylo 5
Filosofos comensales
Comandos: 'a' agregar, 'e' eliminar, 'f' finalizar
Comenzamos...
E . . . .
E . E . .
. . E . E
E . . . E
Agregando filosofo
. E . E . .
. E . E . E
E . . E . .
E . E . . .
E . E . E
Eliminando filosofo...
. E . . E
. E . E .
E . . . .
E . E . .
. . E . E
. E . . E

```

Figura 5: Ejecución del comando phylo en la shell

## IPC

Para permitir la comunicación entre procesos, se implementó un sistema de *pipes*. Cada *pipe* consiste de una estructura que resuelve el problema de lectores y escritores, haciendo uso de los semáforos previamente descritos. Contiene un *buffer* en el que se cargan los datos que el escritor provee, y mediante semáforos se controla la disponibilidad de datos para garantizar el acceso exclusivo y sincronizado.

Un aspecto importante a destacar es que la entrada estándar se implementa como un *pipe*, lo cual permite resolver correctamente el problema de sincronización entre el teclado (escritor) y los procesos que leen desde él. Sin embargo, la salida estándar y la salida de error no se trataron de la misma manera: en lugar de usar *buffering*, se optó por imprimir directamente, realizando un *flushing* inmediato tras cada operación de escritura. En otras palabras, no se agrupan varias salidas antes de imprimir, sino que se imprimen en el momento en que se generan.

## **Limitaciones**

Sobre las limitaciones de nuestro trabajo práctico por suerte son casi todas modificables, es decir, pueden variar a gusto del usuario (pre compilación). Hay 300 semáforos, 16 *pipes* y 512MB de memoria *RAM*.



Hay un archivo de encabezado en la carpeta *SharedLibraries* que se llama “*limitations*”, dentro de él se encuentran las siguientes definiciones:

```
#define NUM_SEMS 300
#define MAX_PIPES 16
#define PIPE_BUFFER_SIZE 100
```

Cambiando los *defines* con valores coherentes y compatibles con las definiciones del sistema (es decir, se esperan valores coherentes para el correcto funcionamiento) se puede ver cómo se modifica en cada compilación del programa.

Si se desea cambiar la memoria *RAM* de *Qemu* se deben hacer dos sencillas modificaciones: la primera es en el archivo *run.sh*, en la línea 11 se debe cambiar el valor de la *flag* “-m” por el valor en megabytes de memoria que se desee tener en esa ejecución. Es importante que se cambie en el archivo *defs.h* de la carpeta *include* de *Kernel* el valor de *HEAP\_END\_ADDRESS* por el valor deseado de memoria menos uno, esa será la última dirección posible de memoria. Se llama así porque el *heap* está planteado que ocupe desde la dirección *HEAP\_START\_ADDRESS* (también modificable, únicamente con valores que superen 0x600000 para evitar pisar a *Userland*) hasta el fin de la memoria disponible. No se recomienda cambiar el tamaño de memoria, a pesar de que funciona sin inconvenientes.

## Conclusión

En conclusión consideramos que este fue un proyecto bastante abarcativo y completo que nos puso a prueba, tanto en nuestro conocimiento y entendimiento de las tareas de un Sistema Operativo, como programadores. Asimismo, aun habiendo llegado a cumplir con las funcionalidades pedidas por la cátedra de manera justa de tiempo, nos hubiera gustado llegar a pulir alguna de estas o inclusive algunos aspectos visuales. Sin embargo, no consideramos que hayamos tenido un mal manejo de tiempo ya que nos topamos con varios problemas que no nos permitían continuar. Tomaron arduas horas o días de *debugging* tanto con *prints* como con *gdb* y búsquedas en el foro de la cátedra (mención especial a los *flag -no-pie* y *-fno-pie* ).

Para sintetizar todo, fue una experiencia intensa, que nos exigió bastante, y creemos que estuvimos a la altura, con lugar a volver algún día a este proyecto y perfeccionarlo.

## **Citas de fragmentos de código reutilizados de otras fuentes**

Se tomó la idea de la consola del sistema *Xv6* de tener una estructura que represente un comando, eso facilitó el pasaje a los *handlers*.

Sobre el problema de los filósofos comensales se tomó como guía la solución propuesta por A. Tanenbaum en el libro “*Sistemas Operativos Modernos*”.