**Instituto Tecnológico de Buenos Aires**

# Trabajo Práctico Número 2: Construcción del Núcleo de un Sistema Operativo y estructuras de administración de recursos

*72.11 - Sistemas Operativos - Grupo 5*


Autores:

Bernasconi, Ian - 62867

Feferovich, Jeremías - 62701

Gutiérrez, Agustín - 62595

# Índice

**1. Introducción** **3**

**2. Decisiones Tomadas durante el desarrollo** **3**

2.1. Memory Manager . . . 3
2.2. Buddy Allocation memory manager . . . 3
2.3. Creación de Procesos . . . 3
2.4. Scheduler . . . 4
2.5. Semáforos . . . 5
2.6. Pipes . . . 5
2.7. Listas . . . 5
2.8. Scope del struct de procesos . . . 6

**3. Instrucciones de compilación y ejecución** **6**

**4. Pasos a seguir para demostrar el funcionamiento** **6**

4.1. Physical Memory Management . . . 6
4.2. Procesos, Context Switching y Scheduling . . . 7
4.3. Sincronización . . . 7
4.4. Inter Process Communication . . . 8
4.5. Aplicaciones de User Space . . . 8
4.5.1. sh . . . 8
4.5.2. help . . . 8
4.5.3. mem . . . 8
4.5.4. ps . . . 9
4.5.5. loop . . . 9
4.5.6. kill . . . 9
4.5.7. nice . . . 9
4.5.8. block . . . 9
4.5.9. cat . . . 9
4.5.10. wc . . . 10
4.5.11. filter . . . 10
4.5.12. phylo . . . 10
4.5.13. set-auto-prio . . . 10

**5. Limitaciones** **10**

**6. Problemas encontrados durante el desarrollo** **11**

**7. Citas de fragmentos de código reutilizados** **11**

**8. Modificaciones realizadas a las aplicaciones de test provistas** **12**

8.1. test-mm . . . . . 12

8.2. test-processes . . . . . 12

8.3. test-priority . . . . . 12

8.4. test-sync y test-no-sync . . . . . 13

# 1. Introducción

En el presente informe se muestra el desarrollo del Trabajo Práctico número 2 de la materia Sistemas Operativos (72.11). El mismo consiste en la creación de un kernel simple basándonos en el trabajo práctico final de la materia anterior, Arquitectura de Computadoras. Para ello, se implementaron un Memory Manager, procesos, scheduling y mecanismos de IPC y sincronización.

# 2. Decisiones Tomadas durante el desarrollo

## 2.1. Memory Manager

El Memory Manager utiliza dos listas doblemente encadenadas, una para los bloques libres y otra para los ocupados. Cada bloque conoce su tamaño y su dirección de comienzo. Se decidió que el struct de cada bloque sea almacenado al comienzo del mismo, por lo que al asignar un bloque de memoria se debe considerar el tamaño de dicho struct. Además de las funciones *allocMemory* y *freeMemory*, se implementó la función *reallocMemory*, que facilitó la programación en el lado de usuario.

## 2.2. Buddy Allocation memory manager

Para el Buddy Allocation memory manager, se utiliza un array en el que cada bit representa un bloque de 32 bytes, el cual indica si dicho bloque se encuentra ocupado o libre. Este arreglo representa un árbol binario, el cual se puede navegar para encontrar el bloque que se desee.

## 2.3. Creación de Procesos

La estructura *ProcessCDT* representa un proceso en nuestro sistema operativo. Contiene varios campos para almacenar información importante, como el PID, el nombre del proceso, su estado, la prioridad, los punteros significativos al stack, descriptores de archivo, y otros campos necesarios para su manejo.

La función *createProcess()* es responsable de crear un nuevo proceso. Recibe argumentos adicionales, como el punto de entrada del proceso, los argumentos de línea de comandos, su prioridad, si se debe ejecutar en foreground o background y los pipes. Esta función llama a *initProcess()* para inicializar el proceso y luego configura los file descriptors del proceso (STDIN y STDOUT). Si se proporcionan pipes, los asigna a los file descriptors correspondientes. Luego, la función prepara el stack del proceso, pusheando registros y argumentos en la pila. Finalmente, establece el estado del proceso en READY si tiene un punto de entrada válido, o en BLOCKED si no tiene punto de entrada.

La función *createProcess()* es llamada desde *scheduler.c* por la función *execve()*, donde el nuevo proceso es agregado a la lista de procesos para que el scheduler lo pueda ejecutar.

## 2.4. Scheduler

La implementación del scheduler se basa en un algoritmo de priority based Round Robin. El scheduler está representado por una estructura llamada *SchedulerCDT*, que contiene varios campos para gestionar el proceso de scheduling, como listas para diferentes niveles de prioridad, un iterador para recorrerlas, una lista para procesos eliminados y otros campos para realizar un seguimiento del proceso actual, el quantum, los procesos en estado de suspensión (sleeping), etc.

Esta función se encarga de guardar el puntero del rsp del proceso actual, actualizar su estado, seleccionar el próximo proceso utilizando la función *getNextProcess()*, modificar si es necesario el proceso actual y devolver el puntero del rsp del nuevo proceso.

Inicialmente habíamos hecho el scheduler para que los procesos fueran subiendo o bajando de prioridad según su consumo del quantum. Pero utilizando este sistema no funcionaba el test de prioridades. Esto se debía a que aunque se les cambiara la prioridad a los procesos con la función *nice*, los mismos usaban todo su quantum haciendo que su prioridad sea la mínima, por lo que no se podía validar el funcionamiento del test. Por esto, implementamos un sistema de prioridades en el que un proceso tiene una prioridad fija (salvo que se la cambie con *nice*), y cada prioridad se corre $2^{prio}$ veces siendo *prio* la prioridad. Se puede pasar al sistema de scheduling dinámico con el comando *set-auto-prio*.

La función *schedule* es responsable de determinar el próximo proceso a ejecutar. Si se selecciona el scheduling dinámico, esta verifica si el proceso actual ha consumido su quantum o si el proceso decidió omitirlo. En el primer caso, se reduce la prioridad del proceso, mientras que en el segundo se aumenta. También se encarga del context switch llamando a la función *changeProcess()*.

La función *getNextProcess()* selecciona el próximo proceso a ejecutar de las listas según su prioridad. Las recorre desde la prioridad más alta hasta la más baja, buscando el primer proceso en estado READY. Si no se encuentra dicho proceso, devuelve el proceso especial llamado *empty* para indicar que no hay procesos para ejecutar.

Con respecto a evitar inanición de procesos con el scheduler dinámico, definimos un MAX_WAITING_TIME que refiere a una cantidad máxima de ticks que se necesita alcanzar para actualizar la prioridad del proceso que está esperando hace más tiempo el CPU. Se decidió implementarlo de forma tal que la prioridad del proceso se vuelve máxima. Esto se hizo para evitar que procesos como *phylos*, que genera múltiples procesos que se bloquean constantemente, se mantengan en la máxima prioridad y no permitan a otros procesos ejecutar.

## 2.5. Semáforos

La implementación de semáforos se basa en el uso de una lista que contiene cada uno de los semáforos utilizados. Cada semáforo está compuesto por varios elementos:

En primer lugar, cada semáforo tiene un nombre asociado. Si no se proporciona un nombre (es decir, se pasa null), el semáforo se considera anónimo.

Además, cada semáforo tiene un valor que indica el estado actual del semáforo. Este valor puede ser modificado mediante las funciones *semPost* y *semWait* para controlar el acceso a los recursos protegidos por el semáforo. Otro elemento es la lista de procesos que están esperando a que el semáforo se desbloquee. Asimismo, se mantiene una lista de todos los procesos que están conectados a dicho semáforo específico. Esta lista proporciona información sobre qué procesos están utilizando el semáforo en un momento dado. Por último, se utiliza un iterador para recorrer eficientemente la lista de procesos conectados al semáforo.

Cabe aclarar que para las funciones *semPost* y *semWait* utilizan instrucciones atómicas como *xchg* para evitar race conditions.

## 2.6. Pipes

Para la implementación de pipes, se utiliza una lista que contiene todos los pipes utilizados en el sistema. Cada pipe en esta lista está compuesto por varios elementos: Cada pipe tiene asociado un nombre que puede ser proporcionado al crearlo. Si se pasa null como nombre, el pipe se considera anónimo. También tiene un buffer para ir almacenando los datos que se transmiten a través del pipe.

Para facilitar las operaciones de lectura y escritura en el pipe, se mantienen índices que indican la posición actual para la lectura y la escritura en el buffer.

Además, se mantiene un registro con la cantidad de procesos que están conectados al pipe, así como también la cantidad de procesos que están bloqueados en el pipe.

Finalmente, se utilizan semáforos para permitir que los pipes sean bloqueantes.

## 2.7. Listas

Se ha optado por implementar una linked list para diversas funcionalidades. Específicamente, se utiliza en el scheduler para organizar los procesos en cada prioridad, así como para administrar los procesos en estado de sleep y los procesos eliminados. Además, se emplea una linked list para la sincronización de procesos que están conectados a través de semáforos y los que están esperando ser liberados. Resultó útil tener un iterador en la estructura del scheduler y en la de semáforos para evitar liberar memoria múltiples veces a lo largo del código.

### 2.8. Scope del struct de procesos

Decidimos expandir el scope del struct de los procesos para que se pueda acceder desde el archivo *scheduler.c*, ya que si no debíamos implementar getters y setters que se llamarían muchas veces, ralentizando los content switches.

## 3. Instrucciones de compilación y ejecución

Todos los comandos que se muestran a continuación deben ser realizados dentro de la carpeta *RowDaBoat-x64barebones-d4e1c147f975*. Para la compilación, se debe realizar uno de los siguientes comandos:

```
$ make all
```

```
$ make buddy
```

Según se quiera utilizar el Memory Manager o el Buddy Allocator respectivamente.

Para la ejecución se debe correr el siguiente comando:

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

## 4. Pasos a seguir para demostrar el funcionamiento

### 4.1. Physical Memory Management

Tanto para el Memory Manager como para el Buddy Allocator, se puede realizar el test de memoria escribiendo el siguiente comando.

```
$>test -mm memory
```

Siendo *memory* la cantidad de memoria a testear. El test se repetirá, imprimiendo en pantalla el número de test completado si la ejecución fue exitosa, hasta que se detenga el comando utilizando Ctrl+C.

Asimismo, se puede reservar memoria utilizando el comando:

```
$>malloc size
```

Siendo *size* la cantidad de memoria a reservar en hexadecimal.

Se puede liberar memoria utilizando el comando:

```
$>free dir
```

Siendo *dir* una dirección en la que antes se había reservado memoria.

Se puede reasignar memoria utilizando el comando:

```
$>realloc dir size
```

Siendo *dir* una dirección en la que antes se había reservado memoria y *size* el nuevo tamaño que se desea asignar.

## 4.2. Procesos, Context Switching y Scheduling

Se puede realizar el test de procesos escribiendo el siguiente comando.

```
$>test-processes cant
```

Siendo *cant* la cantidad de procesos a testear. El test se repetirá, imprimiendo en pantalla el número de test completado si la ejecución fue exitosa, hasta que se detenga el comando utilizando Ctrl+C.

Se puede realizar el test de prioridades escribiendo el siguiente comando.

```
$>test-prio
```

## 4.3. Sincronización

Se puede realizar el test de sincronización escribiendo el siguiente comando.

```
$>test-sync cant
```

Siendo *cant* la cantidad de procesos a testear.

Se puede realizar el mismo test pero sin utilizar mecanismos de sincronización utilizando el siguiente comando:

```
$>test-no-sync cant
```

Siendo *cant* la cantidad de procesos a testear.

## 4.4. Inter Process Communication

En este Sistema Operativo, los mecanismos de IPC implementados son pipes y semáforos. Para utilizar pipes entre procesos mediante la consola, se debe utilizar el símbolo "|".
En cuanto a sus aplicaciones, los semáforos se utilizan para bloquear la consola y otros procesos que necesiten leer del STDIN, esperando la entrada de caracteres. Además, la funcionalidad de "waitpid.[en] el scheduler permite que el proceso que llama a la función se bloquee mediante un "semWait"hasta que el proceso al que espera termine de ejecutarse. De esta forma, se libera mediante un "semPost". También se utilizan en la implementación del proceso 'phylo' como vimos en las clases. En nuestra aplicación, para evitar el 'starving' de algún filósofo, agregamos las variables 'timesEaten' y 'maxTimesEaten' que permiten a un filósofo comer si ha comido menos de dos veces que los filósofos que están a su lado.

## 4.5. Aplicaciones de User Space

### 4.5.1. sh

Cuando se inicial el kernel, se llama a la consola. Para correr un proceso en background se debe insertar el símbolo '&' al final de un comando.
Para conectar dos procesos mediante un pipe, se debe agregar el símbolo 'l' entre ambos. No es posible conectar más de dos procesos con pipes.
Cuando un proceso está corriendo, la shell provee soporte para Ctrl+C para matar al proceso en foreground y Ctrl+D para enviar end of file.

### 4.5.2. help

El comando help se ejecuta de la siguiente forma:

```
1        $>help [command]
```

Siendo *command* el comando que se desea conocer. Si no se pasa ningún parámetro, se mostrará una lista de todos los comandos disponibles. Si se pasa *all* como parámetro, se explicarán todos los comandos. Si se pasa *tests* como parámetro, se explicarán solamente los tests.

### 4.5.3. mem

El comando mem se ejecuta de la siguiente forma:

```
1        $>mem
```

#### 4.5.4. ps

El comando ps se ejecuta de la siguiente forma:

```
$>ps [-k]
```

Si se pasa el parámetro *-k*, se incluirán los procesos zombies.

#### 4.5.5. loop

El comando loop se ejecuta de la siguiente forma:

```
$>loop sec
```

Siendo *sec* la cantidad de segundos de espera entre mensajes.

#### 4.5.6. kill

El comando kill se ejecuta de la siguiente forma:

```
$>kill pid
```

Siendo *pid* el PID del proceso que se desea matar.

#### 4.5.7. nice

El comando nice se ejecuta de la siguiente forma:

```
$>nice pid prio
```

Siendo *pid* el PID de un proceso y *prio* la prioridad que se le desea asigna.

#### 4.5.8. block

El comando block se ejecuta de la siguiente forma:

```
$>block pid
```

Siendo *pid* el PID del proceso que se desea bloquear/desbloquear.

#### 4.5.9. cat

El comando cat se ejecuta de la siguiente forma:

```
$>cat
```

#### 4.5.10. wc

El comando wc se ejecuta de la siguiente forma:

```
$>wc
```

#### 4.5.11. filter

El comando filter se ejecuta de la siguiente forma:

```
$>filter
```

#### 4.5.12. phylo

El comando phylo se ejecuta de la siguiente forma:

```
$>phylo
```

#### 4.5.13. set-auto-prio

El comando set-auto-prio se ejecuta de la siguiente forma:

```
$>set-auto-prio status
```

Siendo *status* el estado (1 o 0) en el que se quiere setear las prioridades automáticas.

## 5. Limitaciones

Cuando se utiliza el Buddy Allocation memory manager, si el tamaño de la memoria proporcionada como entrada no es una potencia de dos, el algoritmo realizará una operación de truncamiento para ajustarla a la potencia de dos más cercana, ya que necesita que el tamaño sea potencia de dos para poder dividirlo en los bloques.

Otras limitaciones están relacionadas con el tamaño limitado de algunos elementos, por ejemplo el buffer de los pipes tiene un tamaño máximo de 1024 bytes, mientras que el tamaño de los stacks está limitado a 4096 bytes.

Por otro lado, es importante mencionar que no se cuenta con protección de memoria, lo que significa que el usuario tiene la posibilidad de provocar un fallo en el Memory Manager si escribe de más.

# 6. Problemas encontrados durante el desarrollo

Inicialmente, para la creación de nuevos procesos teníamos pensado implementar un funcionamiento similar al de UNIX con *fork* y *exec*, pero al no tener memoria virtual y poder implementar Copy on Write, debíamos copiar el contenido del stack del padre, pero deshacer el stackframe, lo que dificultaba bastante su implementación. Finalmente, nos decantamos por el uso de una única función *execve* que combina ambos comportamientos.

En las últimas etapas del proyecto, nos enfocamos en arreglar memory leaks que obtuvimos a lo largo del trabajo. Para esto fue de gran ayuda el uso de *gdb* que permitió encontrar las líneas que generaban la pérdida de memoria.

Otro desafío que surgió durante el desarrollo del proyecto fue el manejo de la tecla *left ctrl*. En un principio, no encontramos dificultades para asignarle la funcionalidad deseada. Sin embargo, comenzamos a tener problemas con la combinación *CTRL+C*, ya que el evento de presionar la tecla era capturado con valores diferentes, los cuales fuimos modificando para lograr su correcto funcionamiento. Finalmente, nos dimos cuenta de que estábamos utilizando *CTRL* como tecla para guardar registros desde nuestro proyecto en Arquitectura de Computadoras, lo que interfería con el valor de *RAX*.

Una complicación fue que cuando se producía una excepción generada, como las generadas por comando, y se intentaba recuperar, se reinicializaba el scheduler y el buffer del videoDriver, pero no se habían borrado los anteriores, lo que producía errores. Esto se solucionó llamando a *closeSchecduler* y a *closeKeyboardBuffer* en *kernel.c*, si no era la primera vez que se corría.

Para terminar, otro problema encontrado durante el desarrollo fue que observamos que no era posible bloquear un proceso en estado de sleep desde la terminal, ya que el proceso se quedaba bloqueado en ese estado y el comando *block* simplemente cambiaba entre estos dos estados. Para abordar esta situación, decidimos implementar el estado SLEEPING en un proceso, lo que permitió bloquear un proceso en estado de dormido y también detener el tiempo en el que debía despertarse.

# 7. Citas de fragmentos de código reutilizados

Si bien no copiamos fragmentos de código de otras fuentes, nos basamos en las sugerencias de *GitHub Copilot* para varias partes del código. Utilizamos estas sugerencias como guía y punto de partida, teniendo

que adaptarlas a nuestras necesidades.

# 8. Modificaciones realizadas a las aplicaciones de test provistas

Para todos los tests, se modificó el prototipo de las funciones para que puedan ser llamadas directamente desde la consola, quedando con el siguiente formato:

```
1    char test_xxx(char argc, char *args[]);
```

## 8.1. test-mm

Se diferenció entre el Memory Manager y el Buddy Allocator usando condiciones de la siguiente forma:

```
1    #ifdef BUDDY
2
3    #else
4
5    #endif
```

Para el MM, la variable *uniform* se modificó para que considere el tamaño del struct:

```
1    uniform = max_memory - total - 1 - BLOCK_STRUCT_SIZE;
```

Y se agregó un chequeo para ver si *uniform* es menor o igual a 0. Mientras que para el Buddy se dejó como estaba en el test provisto. Además, para el MM se le agrega a *total* el tamaño del struct, mientras que para Buddy el tamaño de cada *mm_rqs[rq]* se agranda hasta la potencia de dos más cercana, para que coincida con el tamaño del bloque.

## 8.2. test-processes

No se realizaron cambios más allá de utilizar los nombres de las funciones que habíamos implementado y cambiar los argumentos *argvAux*, quedando como se muestra a continuación.

```
1    char *argvAux[] = {"endless_loop", "0", NULL};
```

## 8.3. test-priority

No se realizaron cambios más allá de utilizar los nombres de las funciones que habíamos implementado y cambiar los argumentos *argv*, quedando como se muestra a continuación.

```
1     char *argv[] = {"endless_loop_print", "0", NULL};
```

## 8.4. test-sync y test-no-sync

No se realizaron cambios más allá de utilizar los nombres de las funciones que habíamos implementado y cambiar los argumentos *argvDec* y *argvInc*, quedando como se muestra a continuación.

```
1 char *argvDec[] ={"my_process_inc", "1", argv[0], "-1", argv[1], NULL};
2 char *argvInc[] ={"my_process_inc", "0", argv[0], "1", argv[1], NULL};
```