# Trabajo Práctico 2

# RoccOS

72.11 - Sistemas Operativos

Construcción de un Sistema Operativo
y sus mecanismos de administración de recursos

*Instituto Tecnológico de Buenos Aires*

**Grupo 6**
Bautista Pessagno - 65101
Rocco Perrone - 65628
Pascal Ordano - 65485

10/11/2025

# 1. Introducción

En el siguiente trabajo práctico se realizó un kernel monolítico de 64 bits, partiendo de un trabajo práctico de la materia Arquitectura de Computadoras como base. El TP incluye un manejo de memoria, utilización y manejo de procesos por medio de un scheduler y el uso de IPCs para la comunicación entre procesos. A su vez se realizaron las syscalls solicitadas y algunas más para poder ofrecer una abstracción de las funciones del kernel al usuario y poder usar las mismas en programas como sh.

# 2. Manejo de Memoria

El kernel implementa un **administrador de memoria dinámica** basado en un *heap contiguo y fijo*, comprendido entre las direcciones 0x0000000000100000 y 0x00000000003FFFFF

Sobre este espacio se construyó un asignador que mantiene una **lista libre circular ordenada por dirección** y administra los bloques mediante estructuras `Header`, garantizando la alineación y la correcta delimitación de cada región de memoria.

```
typedef union Header {
   struct {
       union Header *next;   // solo válido cuando está en la free-list
       size_t units;         // tamaño EN UNIDADES de Header (incluye header)
   } s;
   Align _;                  // fuerza alineación del Header/payload
} Header;
```

Durante la inicialización del kernel, el **manager se ubica dinámicamente** justo después del código del kernel y su pila (aproximadamente 32 KiB), alineándose a los límites de página de 4 KiB para asegurar seguridad y coherencia en los accesos. A partir de allí, se define el *pool* de memoria que se extiende hasta el límite superior permitido, excluyendo el área reservada para los módulos de userland, como el *shell*. En el arranque, se crea un **bloque libre único** que abarca todo el espacio administrado, desde el cual se derivan las asignaciones posteriores.

```
uintptr_t kernel_end = (uintptr_t)&endOfKernel;
uintptr_t manager_begin = align_up_uintptr(kernel_end + stack_size + page_size, page_size);

// Construir el objeto en la dirección calculada (después del kernel)
mm = (MemoryManagerADT)manager_begin;

uintptr_t manager_end   = manager_begin + (uintptr_t)sizeof(*mm);
```

```
uintptr_t aligned_pool_start = align_up_uintptr(manager_end, sizeof(Header));
uintptr_t pool_end_uint = (uintptr_t)MEMORY_MANAGER_LAST_ADDRESS;
```

Utilizamos dos tipos de memory manager, uno con política de asignacion “buddy” y otro con política de asignación "first fit”.

Para "first fit” se recorre la lista libre hasta hallar el primer bloque de tamaño suficiente. Si el bloque excede el tamaño solicitado, se lo divide (*split*) dejando el remanente en la lista libre; si coincide exactamente, se elimina de la misma. Al liberar memoria, el sistema identifica el bloque correspondiente y aplica una política de coalescing para fusionar áreas contiguas, minimizando la fragmentación externa.

Para “buddy” el objetivo es administrar un heap del kernel dividiéndolo en bloques de tamaño potencia de 2. En cada asignación se redondea hacia arriba al siguiente tamaño potencia de 2 y se parte (“split”) un bloque grande hasta alcanzar ese tamaño. En la liberación, se intenta fusionar (“coalesce”) con el buddy contiguo de igual tamaño, repitiendo mientras sea posible para recomponer bloques grandes.

```
Header *findBlock(Header *header, size_t memory_required) {
    if (header == NULL) {
        return NULL;
    }

    Header *current = header;
    Header *prev = NULL;

    if (current->s.units == memory_required) {
        header = header->s.next;
        return current;
    }

    prev = current;
    current = current->s.next;

    while (current != NULL){
        if (current->s.units == memory_required) {
            prev->s.next = current->s.next;
            return current;
        }
        prev = current;
        current = current->s.next;
    }

    return NULL;
```

```
}
```

```
void splitHeader(Header *header) {
    if (header == NULL || header->s.units < 2) {
        return; // Cannot split a block smaller than 2 units
    }

    size_t half_size = header->s.units / 2;

    // Create the second header at the midpoint
    Header *second_header = (Header *)((uint8_t *)header + half_size);

    // Set up the second header
    second_header->s.units = half_size;
    second_header->s.next = header->s.next;

    // Update the first header
    header->s.units = half_size;
    header->s.next = second_header;
}
```

Nuestro memory manager solo expone las funciones mm_free y mm_malloc para poder alternar entre ambas politicas de asignacion de memoria simplemente modificando argumentos en la compilacion.

# 3. Manejo de Procesos

El manejo de procesos del kernel se apoya en un PCB que concentra PID, padre, prioridad, estado, punteros al stack y descriptores de archivo y datos para espera y retorno.  Los estados posibles son READY, RUNNING, BLOCKED, ZOMBIE, DEAD.

```
typedef struct Process {

uint16_t pid;

uint16_t parentPid;

uint8_t priority;

ProcessState state;

void *stackBase;

void *stackPos;

char *name;

char **argv;

void *zombieChildren;  // LinkedListADT

uint16_t waitingForPid;

int32_t retValue;

uint8_t unkillable;

int16_t fileDescriptors[3];

} Process;

typedef enum {
  READY = 0,
  RUNNING = 1,
  BLOCKED = 2,
  ZOMBIE = 3,
  DEAD = 4
} ProcessState;
```

En createScheduler() se crea e inicializa el scheduler dejando listo el sistema de colas READY multinivel y una cola separada para BLOCKED. La selección del próximo proceso recorre de mayor a menor prioridad, y el tick de timer aplica preemptive scheduling por quantum: guarda el contexto del actual, ajusta prioridad por aging y devuelve el stack del siguiente para efectuar el cambio en assembler.

La creación de procesos reserva el PCB y la pila, arma un argv contiguo en memoria de kernel, configura FDs (estándar o pipes) y construye el stack frame inicial para entrar por un wrapper que invoca main(argc, argv) y garantiza la terminación ordenada.

El bloqueo/desbloqueo mueve nodos entre colas y, al desbloquear, otorga un *boost* para reactividad. La terminación marca ZOMBIE, cierra FDs, notifica/ despierta al padre si está esperando y destruye o encola el zombi; waitpid retira y libera el zombi devolviendo su código. El sistema reutiliza PIDs, expone snapshots para monitoreo y se integra con syscalls que crean/esperan procesos, todo sobre una API sched_* estable y acotada.

# 4. Sincronización

El kernel implementa dos mecanismos de sincronización fundamentales: semáforos y pipes, ambos integrados con el planificador para coordinar la ejecución de procesos concurrentes.

Los semáforos funcionan como contadores enteros protegidos por un mecanismo de exclusión mutua basado en la instrucción atómica _xchg, que garantiza acceso exclusivo al modificar su estado.

```
static void acquireMutex(Semaphore *sem) {
   while (_xchg((volatile uint64_t *)&(sem->mutex), 1)) {
       uint16_t pid = getpid();
       appendElement(sem->mutexQueue, (void *) ((uint64_t) pid));
       setStatus(pid, BLOCKED);
       yield();
   }
}
```

Cada semáforo mantiene una cola de espera para los procesos bloqueados tanto por la sección crítica como por el propio valor del contador. Cuando un proceso realiza una operación down, el kernel adquiere el mutex y, si el valor del semáforo es cero, el proceso se bloquea y cede la CPU mediante yield(), siendo reactivado cuando otro proceso ejecuta up. Este último incrementa el contador y reanuda al primer proceso en espera, restableciendo así el flujo sincronizado.

```
static int down(Semaphore *sem) {
    acquireMutex(sem);
    while (sem->value == 0) {
        uint16_t pid = getpid();
        appendElement(sem->semaphoreQueue, (void *) ((uint64_t) pid));
        setStatus(pid, BLOCKED);
        releaseMutex(sem);
        yield();

        acquireMutex(sem);
    }
    sem->value--;
    releaseMutex(sem);

    return 0;
}
```

```
static int up(Semaphore *sem) {
    acquireMutex(sem);
    sem->value++;
    resumeFirstAvailableProcess(sem->semaphoreQueue);
    releaseMutex(sem);
    return 0;
}
```

El modelo de pipes implementa una sincronización del tipo productor–consumidor, en la cual dos procesos se comunican a través de un búfer circular compartido. Si el proceso escritor intenta enviar datos cuando el búfer está lleno, se bloquea hasta que el lector libere espacio; de forma análoga, el lector se bloquea si intenta leer cuando el búfer está vacío. En ambos casos, el mecanismo de desbloqueo se activa cuando el proceso complementario modifica el estado del búfer, restableciendo la condición de disponibilidad y reprogramando la ejecución mediante setStatus(..., READY).

```
while (writtenBytes < len && (int) pipe->buffer[bufferPosition(pipe)] != EOF) {
    if (pipe->currentSize >= PIPE_SIZE) {
        pipe->isBlocking = 1;
        setStatus((uint16_t) pipe->inputPid, BLOCKED);
        yield();
    }
    …
    if (pipe->isBlocking) {
        setStatus((uint16_t) pipe->outputPid, READY);
        pipe->isBlocking = 0;
    }
```

Tanto semáforos como pipes se integran directamente con el scheduler, que actualiza el estado de los procesos y los reubica en las colas correspondientes según las operaciones de bloqueo y desbloqueo. Esto garantiza una cooperación ordenada entre procesos, evitando condiciones de carrera. En conjunto, estos mecanismos constituyen la base de la sincronización y exclusión mutua del kernel, permitiendo una ejecución concurrente consistente y eficiente.

# 5. Comunicación Entre Procesos

El kernel implementa un sistema de comunicación entre procesos (IPC) basado en *pipes*, que permite la transferencia unidireccional de datos entre un proceso productor y otro consumidor.

```
typedef struct Pipe {

   char buffer[PIPE_SIZE];

   uint16_t startPosition;

   uint16_t currentSize;

   int16_t inputPid, outputPid;

   uint8_t isBlocking;

} Pipe;
```

Cada *pipe* posee un búfer circular en memoria del kernel, administrado por una estructura que almacena el estado del búfer, las posiciones de lectura y escritura, los identificadores de los procesos en cada extremo y un indicador de bloqueo.

El mecanismo se expone a los procesos mediante descriptores de archivo (FDs): durante la creación de un proceso, el kernel asocia los FDs estándar (`STDIN`, `STDOUT`, `STDERR`) o, en caso de existir pipes personalizados, abre los extremos correspondientes para el proceso en modo lectura o escritura según su rol.

El ciclo de vida de un pipe comienza con la creación del objeto en el `pipe manager`, la asignación de un identificador libre y la vinculación de los procesos que lo utilizarán. Si uno de los extremos cierra su conexión, el sistema aplica una política segura de finalización: al cerrarse el extremo de escritura, el kernel inyecta un byte `EOF` en el búfer para despertar al lector y permitirle terminar sin bloqueo; si el cierre ocurre del lado del lector, se libera completamente el recurso.

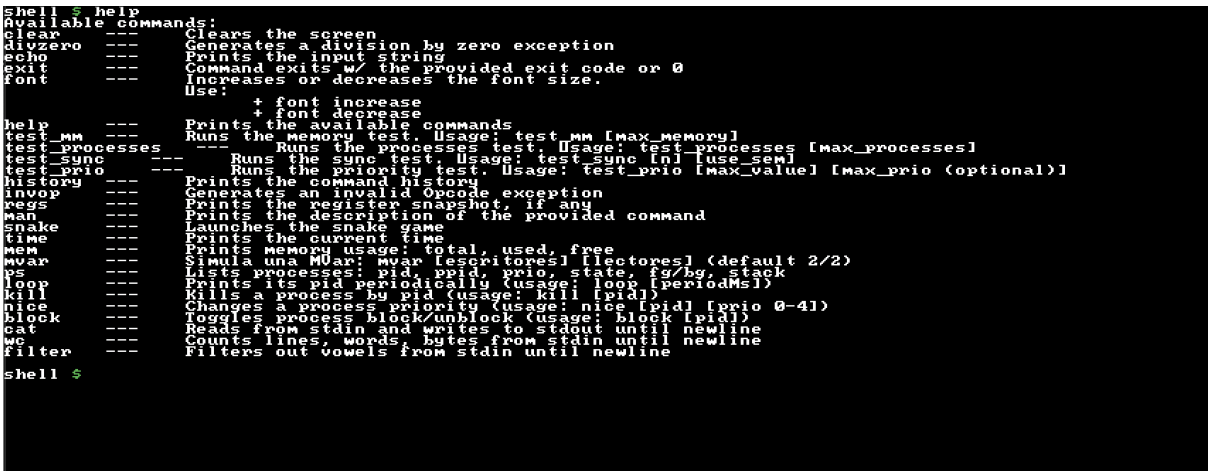
El comportamiento de bloqueo y sincronización está integrado al planificador del kernel. Si un proceso intenta escribir cuando el búfer está lleno, su estado cambia a *BLOCKED* y cede la CPU mediante `yield()`, reanudandose cuando el lector libera espacio. De manera análoga, un lector que intenta leer un búfer vacío se bloquea hasta que el escritor produzca nuevos datos. Este esquema de *wake-up selectivo* garantiza que cada extremo se active sólo cuando el otro haya modificado el estado del búfer, evitando condiciones de carrera y eliminando la necesidad de semáforos adicionales.

A nivel de diseño, los *pipes* ofrecen un modelo de sincronización embebido: el control de concurrencia se logra exclusivamente a través del scheduler (`setStatus`/`yield`) y los flags internos del pipe, sin depender de primitivas externas. La interfaz desde userland abstrae esta complejidad, permitiendo que los programas creen pipelines entre procesos simplemente manipulando FDs y usando las llamadas a sistema de lectura y escritura habituales. En conjunto, este sistema proporciona una comunicación segura, bloqueante y ordenada, cumpliendo el rol fundamental de sincronizar la ejecución de procesos concurrentes y facilitar la cooperación entre ellos.

# 6. Terminal

La manera de interactuar con el sistema operativo es a través de la terminal. La terminal actúa como interfaz principal entre el usuario y el sistema operativo. Esta terminal permite ejecutar comandos, iniciar procesos, observar el estado interno del sistema y probar funcionalidades del kernel. Su diseño se basa en un modelo de E/S estándar, donde el shell lee instrucciones desde `STDIN`, las interpreta y las despacha como procesos independientes, redirigiendo su salida a `STDOUT` o a través de *pipes* cuando se encadenan comandos.

Al iniciar el entorno, el shell muestra un prompt (`shell $`) y acepta comandos definidos dentro del sistema. El listado completo se obtiene mediante `help`, que imprime las operaciones disponibles junto con su descripción y uso. Entre ellas se incluyen herramientas de diagnóstico como `test_mm`, `test_processes` y `test_sync`, utilizadas para evaluar el correcto funcionamiento del manejador de




memoria, del scheduler y de los mecanismos de sincronización respectivamente. También se proveen comandos de prueba de excepciones (`divzero`, `invop`), monitoreo (`ps`, `stack`, `mem`), y utilitarios clásicos como `cat`, `wc`, `filter`, o `loop`, que permiten leer desde la entrada estándar, contar líneas y caracteres, o ejecutar procesos periódicos.

El shell soporta además manipulación de procesos en ejecución: mediante `kill [pid]` es posible finalizar un proceso, `nice [pid] [prio]` modifica su prioridad, y `block` permiten cambiar su estado de ejecución entre bloqueado y desbloqueado. Estas órdenes interactúan directamente con el planificador, ofreciendo una forma práctica de validar la gestión de procesos en tiempo real.

El entorno también incorpora comandos de sistema y mvar y tests, pensado para probar la concurrencia y la sincronización entre procesos. Gracias a esta interfaz, el usuario puede interactuar con todos los subsistemas del kernel sin intervención externa, verificando el comportamiento del heap, los semáforos, los pipes y la planificación, todo dentro de un entorno controlado y visualmente representativo del funcionamiento del sistema operativo.

# 7. Conclusión

El desarrollo del trabajo práctico ayudó a la comprensión de la profundidad del mecanismo detrás de un sistema operativo. A pesar de no haber implementado algunos aspectos de manera totalmente alineadas a la realidad, se logró comprender y aprender cómo opera y cómo implementar aspectos fundamentales como el scheduler para manejar el uso del cpu , el memory manager para la asignación de memoria, y los IPCs para la comunicación entre procesos

# 8. Escenarios problemáticos

Al buscar errores con PVS-Studio, se encontró errores los cuales no se trataron, ya que muchos correspondían a cosas ajenas a nuestro trabajo y código realizado, ya sea código en Bootloader. A su vez asumimos un *edge case* el cual sucede al hacer CTRL + C sobre el test_processes [PROCESSES]  al estar en el foreground

# 9. Instrucciones de Compilación y Ejecución

Para compilar y ejecutar tenga Docker instalado, luego ingrese en en el proyecto y simplemente ejecute `./compile.sh` para compilar y `./run.sh` Si quiere ejecutar todo junto corra `./compile.sh && ./run.sh` Recuerde que probablemente deba ajustar los permisos de ambos archivos para un correcto funcionamiento, simplemente utilice chmod +x. Para compilar con la política de asignación de memoria “buddy” compilar con ./compile.sh buddy