Informe - Trabajo Práctico N°2

72.11 - Sistemas Operativos

Grupo 14 - ColidiOS
Cortese, Andrés - 64612 - acortese@itba.edu.ar
Esquivel, Tomás Jerónimo - 64756 - tesquivel@itba.edu.ar

Fecha de entrega: 09/06/2025

2025

Índice

1. Introducción

El objetivo del presente trabajo consistió en el desarrollo de múltiples funcionalidades propias de un sistema operativo, tomando como base el kernel monolítico de 64 bits implementado en la materia Arquitectura de Computadoras, que cuenta con características fundamentales como manejo básico de interrupciones, system calls, driver de teclado, driver de video (modo texto o gráfico) y separación entre binarios de kernel space y user space.

El desarrollo se enfocó en cuatro funcionalidades principales: manejo de memoria, manejo de procesos, sincronización y comunicación entre procesos. Este trabajo permitió profundizar los conocimientos adquiridos sobre sistemas operativos de tipo UNIX mediante la construcción y análisis de un sistema propio.

2. Manejo de Memoria

2.1 Implementación

El sistema operativo desarrollado cuenta con dos administradores de memoria intercambiables: un administrador propio implementado desde cero y un sistema buddy. Ambos administradores comparten una interfaz común para permitir su intercambio mediante directivas de preprocesador al momento de compilación.

El memory manager propio está basado en una estrategia de lista enlazada no ordenada, donde cada bloque libre se administra en una estructura que contiene información sobre el tamaño y su disponibilidad. Se implementan funciones para reserva (`malloc`) y liberación (`free`) de bloques, y se maneja la fusión de bloques adyacentes libres para reducir fragmentación.

Por otro lado, el buddy system está basado en divisiones de memoria en potencias de dos. El área total de memoria se inicializa como un bloque único de nivel 0 y puede subdividirse hasta un máximo de niveles predefinido. La asignación busca el bloque más pequeño que satisfaga la solicitud, y la liberación permite fusionar "buddies" adyacentes para mejorar la reutilización.

Ambos administradores están preparados para ser utilizados tanto por el kernel como por procesos en user space. Se ofrecen system calls para:

- Reservar memoria.
- Liberar memoria.
- Consultar el estado de la memoria (total, libre, ocupada).

Los tests `test_mm` fueron superados exitosamente para ambas implementaciones. Para utilizar el test_mm se recomienda correrlo en background y luego jugar con el resto de comandos para ver si ocurren fallas en la memoria. Si no ocurren fallas en el manejo de errores, no salta ningún mensaje por la terminal.

Un ejemplo de línea de comando puede ser:

```
$> testmm 20 &
```

2.2 Limitaciones

El administrador propio presenta ciertas limitaciones con respecto a la fragmentación externa, dado que al no mantener una lista ordenada por dirección o tamaño, puede haber dificultades para reutilizar bloques pequeños.

El buddy system, si bien mejora el manejo de fragmentación externa y facilita la fusión de bloques, presenta como limitación principal la fragmentación interna: todas las asignaciones se redondean a potencias de dos, pudiendo desaprovechar memoria en solicitudes pequeñas.

También cabe destacar que la cantidad de niveles y el tamaño mínimo de bloque están fijos en tiempo de compilación, lo cual puede limitar la flexibilidad del sistema en escenarios con necesidades muy variables de tamaños de asignación.

En ambos casos, no se implementó soporte para `realloc`, y la estrategia de asignación no contempla prioridades ni categorías diferenciadas entre procesos.

3. Manejo de Procesos

3.1 Implementación

El sistema operativo implementa multitasking con cambio de contexto preemptivo mediante interrupciones del timer. Cada proceso se representa con una estructura que contiene su identificador (PID), prioridad, estado (READY, BLOCKED, etc.), punteros de pila (stack pointer y base pointer), y demás información relevante.

Los procesos se almacenan en una lista enlazada, y se asignan pilas mediante un manejador de stacks propio. Al crear un nuevo proceso, se inicializa su contexto con los registros necesarios, incluyendo los punteros de pila, y se coloca en la lista de READY.

El cambio de contexto se realiza guardando los registros del proceso actual y cargando los del siguiente proceso seleccionado por el scheduler. Este mecanismo está soportado por rutinas en scheduler.c y process.c.

El algoritmo de planificación implementado es Round Robin con prioridades. Cada proceso tiene una prioridad asignada (LOW, MEDIUM o HIGH), y el scheduler selecciona el siguiente proceso READY con la mayor prioridad disponible. Si hay varios con la misma prioridad, se rota en orden de llegada.

Se ofrecen system calls para:

- Crear procesos, permitiendo el pasaje de parámetros.
- Finalizar procesos.
- Obtener el PID del proceso actual.
- Consultar la lista de procesos.
- Cambiar prioridad.
- Matar procesos.
- Bloquear/desbloquear procesos.
- Renunciar al CPU (yield).
- Esperar a que los hijos terminen.

Los tests test_processes y test_priority fueron ejecutados como procesos independientes y superados correctamente.

Ejemplos de línea de comando para ejecutar tests:

Test de prioridades:

```
$> testprio
```

Test de procesos (en background):

```
$>testproc 10 &
```

`$>ps` (para ver cómo varía la cantidad de procesos y su estado)

3.2 Limitaciones

El sistema actual no soporta herencia de prioridades ni manejo jerárquico de procesos padres e hijos más allá del wait. Además, no se implementa suspensión o paginación, por lo que todos los procesos deben residir completamente en memoria.

También, la cantidad de procesos simultáneos está limitada por el espacio disponible para stacks y estructuras de control, y no se contempla una cola de espera para procesos que no pueden iniciarse por falta de recursos.

4. Sincronización

4.1 Implementación

La sincronización entre procesos en el sistema se logra mediante semáforos, implementados en el archivo semaphores.c. Se proporciona una interfaz de system calls para operar con semáforos nombrados e identificados por enteros, lo cual permite sincronización tanto en kernel space como entre procesos en user space.

Cada semáforo contiene:

- Un valor entero que representa su estado (disponible u ocupado).
- Una cola de procesos bloqueados (cuando el valor es 0 o menor).
- Un identificador único dentro del sistema.

Las funciones disponibles a nivel de sistema incluyen:

- libcSemOpen(id, value): Inicializa el semáforo con el identificador id y el valor value. Si ya existe, lo reutiliza.
- libcSemWait(id): Decrementa el semáforo. Si su valor es negativo, el proceso actual se bloquea.
- libcSemPost(id): Incrementa el semáforo. Si hay procesos bloqueados, despierta al primero en la cola.
- libcSemClose(id): Libera los recursos del semáforo.
- libcSemOpenGetId(value): Devuelve un nuevo ID automáticamente asignado para inicializar un nuevo semáforo.

Internamente, la cola de bloqueados es manejada con una lista ordenada (ordered_list_adt.c), y la relación entre semáforos y procesos está correctamente desacoplada del scheduler para evitar problemas de dependencia cíclica.

Los semáforos son utilizados ampliamente en los tests test_sync y en otros procesos de prueba para sincronizar el acceso a recursos compartidos como salida estándar, memoria compartida o impresiones secuenciales.

Ejemplos de línea de comandos para ejecutar tests:

Testeo sin sincronización:

`$> testsync 1000 0` (el resultado debe ser distinto de 0)

Testeo con sincronización:

`$>testsync 1000 1` (el resultado correcto debe ser 0)

4.2 Limitaciones

La implementación actual no incluye mecanismos para semáforos binarios explícitos ni políticas avanzadas como prioridad en el desbloqueo o herencia de prioridad. Además, no se implementa limpieza automática de semáforos al finalizar todos los procesos que lo utilizan.

5. Comunicación Entre Procesos

5.1 Implementación

La comunicación entre procesos (IPC) se implementa mediante pipes unidireccionales. La funcionalidad está centralizada en el archivo pipe.c.

Las operaciones disponibles mediante system calls son:

- libcPipeOpen(id, mode): Abre un pipe con identificador id en modo de lectura o escritura.
- libcPipeOpenFree(mode): Crea un pipe nuevo con ID único automáticamente, útil para procesos que se comunican sin acordar un ID fijo.
- libcPipeRead(id, buffer, size): Intenta leer hasta size bytes del pipe; si no hay datos disponibles, el proceso se bloquea.
- libcPipeWrite(id, buffer, size): Escribe en el pipe; si no hay espacio suficiente, el proceso también se bloquea.
- libcPipeClose(id): Cierra la conexión al pipe, liberando recursos si no hay más referencias.
- libcPipeReserve(): Devuelve un nuevo ID válido para evitar colisiones al crear pipes nuevos.

Internamente, los pipes utilizan semáforos para coordinar el acceso concurrente entre procesos lectores y escritores.

Ejemplo de línea de comando para usar un pipe:

```
$>ps | filter
```

5.2 Limitaciones

- El tamaño del buffer del pipe es fijo, lo que puede limitar el throughput en escenarios con muchos datos.
- No se implementó una cola de prioridad en el acceso a pipes; todos los procesos esperan en orden FIFO.
- No se contempla multiplexado de entrada/salida (como select o poll), por lo que un proceso debe bloquearse completamente al leer si no hay datos.

6. Terminal

La shell de Arquitectura de Computadoras fue alterada para que corra como un proceso. Esta soporta las siguientes características:

- Ejecución de comandos en foreground y background: si el comando termina con &, el proceso se lanza en segundo plano. De lo contrario, la shell espera a que el proceso finalice antes de continuar.

- Pipes entre dos procesos: utilizando el carácter |, la shell divide el input en dos programas, crea un pipe, y conecta el stdout del primero con el stdin del segundo.
- Soporte de atajos de teclado: se manejan señales como:
    - Ctrl + C: termina el proceso en foreground usando una syscall de kill.
    - Ctrl + D: interpreta EOF y cierra la entrada estándar, lo que puede finalizar algunos procesos o retornar al prompt.

Comandos soportados

La terminal reconoce tanto comandos built-in como comandos ejecutados como procesos. Algunos ejemplos son:

- help: muestra los comandos disponibles y los tests incluidos.
- ps, kill, nice, block: relacionados a manejo de procesos.
- mem: imprime el estado actual de la memoria.
- loop, cat, wc, filter, phylo: ejemplos de programas ejecutables.

7. Warnings PVS

Durante el desarrollo del sistema operativo, se utilizó la herramienta **PVS-Studio** para realizar análisis estático de código en busca de posibles errores, código no seguro o prácticas de programación subóptimas. El análisis abarcó tanto el kernel como los módulos del espacio de usuario.

Advertencias que persistieron hasta la entrega final:

bmfs.c (437)	V595 The 'buffer' pointer was utilized before it was verified against nullptr. Check lines: 437, 467.
bmfs.c (442)	V595 The 'disk' pointer was utilized before it was verified against nullptr. Check lines: 442, 461.
bmfs.c (519)	V1032 The pointer 'dir_copy' is cast to a more strictly aligned pointer type.
bmfs.c (569)	V1032 The pointer 'Directory' is cast to a more strictly aligned pointer type

Todas las advertencias finales se encuentran en el archivo bmfs.c dentro de la carpeta Bootloader. Como este código fue otorgado por la cátedra y refiere a una lógica que no forma parte de los principales aprendizajes del trabajo práctico, se decidió no modificarlo como medida de precaución.

8. Conclusión

El desarrollo de este sistema operativo permitió consolidar conocimientos clave sobre arquitectura de sistemas, manejo de memoria, procesos, sincronización y comunicación entre procesos. A lo largo del proyecto, se implementaron desde cero módulos fundamentales del kernel, así como una terminal funcional para interactuar con el sistema desde userland.

Entre los logros más destacados se encuentran:

- Dos estrategias completas de manejo de memoria (from_scratch y buddy system), con alocación y liberación dinámica.
- Un modelo de procesos con scheduler round-robin y soporte para foreground/background, bloqueos, y prioridades.
- Sincronización robusta mediante semáforos y comunicación efectiva a través de pipes.
- Una shell interactiva con comandos propios, pipes y parsing de argumentos.

Asimismo, se aplicaron buenas prácticas de desarrollo como modularización, diseño de estructuras abstractas reutilizables (listas, colas, etc.) y análisis estático del código mediante PVS-Studio, lo que mejoró la confiabilidad del sistema.

El proyecto no solo sirvió para aplicar contenidos teóricos vistos en clase, sino también para adquirir experiencia en diseño de software de bajo nivel, debugging complejo, y trabajo estructurado sobre un sistema completo.

9. Instrucciones de Compilación y Ejecución

Requisitos:

- Docker instalado y en funcionamiento.
- Sistema operativo anfitrión: macOS, Ubuntu o Windows Subsystem for Linux (WSL).
- Los scripts create.sh, compile.sh y run.sh deben tener permisos de ejecución (por ejemplo, usando chmod +x).
- Estar en la carpeta x64BareBones-master

Scripts disponibles:

El proyecto incluye tres scripts que automatizan la compilación y ejecución del sistema operativo:

1. create.sh: inicializa el entorno de desarrollo dentro de un contenedor Docker. Este paso es necesario realizarlo una única vez (o cuando se desee recrear el entorno).

2. compile.sh [memory_manager] [opcional: –pvs]: compila el sistema operativo. Puede recibir un argumento que define qué manejador de memoria utilizar:
    - buddy: selecciona la implementación basada en el algoritmo buddy system.
    - from_scratch o sin argumento: utiliza la implementación alternativa, desarrollada desde cero.

Puede también recibir el argumento “--pvs” si es que se quiere generar un reporte utilizando PVS Studio.

3. Ejemplos de uso:
    - ./compile.sh buddy
    - ./compile.sh from_scratch
    - ./compile.sh
    - ./compile.sh buddy –pvs
    - ./compile.sh from_scratch –pvs
4. run.sh: ejecuta el sistema operativo en QEMU. Al iniciarse, el script solicita al usuario que ingrese su sistema operativo anfitrión, eligiendo entre MAC, Ubuntu o WSL, ya que algunas configuraciones dependen de esta elección.

Flujo típico de uso:

cd x64BareBones-master

./create.sh

./compile.sh [memory manager] (“buddy”, “from_scratch” o “ ”) [--pvs](opcional)

./runs.sh [so] (“MAC”, “Ubuntu” o “WSL”)

Primero se corre create.sh para preparar el entorno. Luego, se compila el sistema con compile.sh especificando (o no) el tipo de manejo de memoria y si se quiere compilar con PVS Studio. Finalmente, se ejecuta el sistema con run.sh indicando el sistema operativo del host.

Extras:

Dentro del contenedor también se puede utilizar make clean para limpiar los archivos temporales o make rebuild para forzar una recompilación total del proyecto.