Trabajo Práctico Nº2: Construcción del Núcleo de un Sistema Operativo y mecanismos de administración de recursos

Sistemas Operativos (72.11)

2025 - Primer Cuatrimestre

Integrantes:

Alexis Herrera Vegas - 64045

Sebastian Caules - 64331

Federico Kloberdanz - 62890

Introducción

Se presenta en este trabajo la implementación de diversas funcionalidades de un sistema operativo, incluyendo manejo de memoria física, procesos, context switching y scheduling, sincronización e Inter-Process Communication. Se tomó como base el trabajo práctico realizado en Arquitectura de Computadoras, que ya incluía manejo de interrupciones básico, system calls, drivers y separación entre kernel y user space.

Decisiones tomadas durante el desarrollo

Physical Memory Management

Para el manejo de memoria, implementamos un memory manager basado en listas enlazadas. La memoria disponible se modela como una lista de bloques *block_header_t*, cada uno con información sobre su tamaño, si está libre u ocupado, y un puntero al siguiente bloque.
Las asignaciones de memoria se realizan con *my_malloc*, que recorre la lista buscando un bloque libre suficientemente grande. Si es posible, el bloque se divide para optimizar el uso del heap. Al liberar memoria con *my_free*, el bloque se marca como libre y se intenta unir con bloques adyacentes contiguos también libres, para evitar la fragmentación.
Se implementó también un buddy memory manager. A diferencia del memory manager propio, donde el tamaño de los bloques puede variar libremente y requiere recorrer toda la lista para encontrar un espacio adecuado, el buddy system utiliza bloques de tamaño fijo (potencias de dos) y estructuras ordenadas por nivel (free lists), lo que facilita encontrar y reutilizar memoria de manera más rápida y ordenada.

Procesos, Context Switching y Scheduling

El scheduler implementa planificación de procesos con Round Robin con prioridades. Mantiene una lista de procesos activos y decide cuál ejecutar en cada tick de CPU. Cada proceso tiene un quantum proporcional a su prioridad. Cuando este se agota, el scheduler

elige el siguiente proceso listo en orden circular. También maneja la creación, terminación, bloqueo y desbloqueo de procesos.
Para asignar un PID, el scheduler le asigna a un proceso el primer PID disponible.

Sincronización

Se implementó un sistema de semáforos nombrados, permitiendo que procesos no relacionados compartan semáforos mediante un identificador acordado. Cada semáforo mantiene su valor, una cola circular de procesos bloqueados y un contador de uso. El acceso concurrente se protege mediante *spinlocks*, garantizando atomicidad y evitando *race conditions*. El diseño previene *deadlocks* al no permitir bloqueos circulares internos y al asegurar liberación ordenada y exclusiva mediante acquire/release.

Inter-Process Communication

Se implementaron pipes con operaciones de lectura y escritura bloqueantes. Cada pipe cuenta con un buffer circular protegido por semáforos nombrados, lo que permite la sincronización sin busy waiting entre procesos no relacionados. Las funciones *readPipe* y *writePipe* bloquean al proceso si el buffer está vacío o lleno, respectivamente. Los file descriptors asociados permiten redirigir entrada y salida estándar sin modificar el código de los programas, facilitando su ejecución tanto en forma aislada como encadenada mediante el intérprete de comandos.

Aplicaciones de User Space

Ejecutando el comando *help* se pueden ver las funciones ofrecidas por el kernel. Se dejaron las funciones principales del trabajo en la sección principal. Funciones del TPE de Arquitectura de Computadoras se movieron a otras secciones, y se creó una sección destinada a los testeos, pudiéndose abrir ejecutando *help test*.
Se le añadió una columna *% CPU* al comando *ps*, en la que se puede visualizar la proporción de timer ticks que usa cada proceso. La suma no siempre llega a 100 por cuestiones de redondeo. Para calcular los ticks totales y efectuar un cálculo correcto se debe primero recorrer todos los procesos. No funcionaría tener una variable *total_ticks* en el scheduler ya que esta incluiría los ticks de los procesos muertos, resultando en porcentajes incorrectos.
Se escribió el comando *filter* de manera que filtra una vez que el EOF es enviado, no en tiempo real.
Nuestra implementación del programa phylo crea un proceso por filósofo y usa semáforos nombrados para sincronizarlos sin busy waiting. Se controla la visualización para mostrar solo cambios relevantes en el estado de la mesa, y se valida que nunca haya dos filósofos contiguos comiendo. El mutex global garantiza exclusión mutua al modificar estados, y la interfaz por consola permite agregar y remover filósofos en tiempo real. Se incorporaron verificaciones y controles visuales adicionales para asegurar la validez de la sincronización. Por ejemplo, se imprime un error si dos filósofos contiguos están comiendo.

Limitaciones

En el memory manager, aunque los bloques libres contiguos se fusionan en *my_free*, el sistema no compacta ni mueve bloques. Esto puede generar fragmentación externa si quedan muchos bloques pequeños dispersos que no alcanzan para satisfacer futuras asignaciones grandes. Aparte de eso, *my_malloc* recorre la lista desde el principio cada vez que se solicita memoria. Esto puede volverse ineficiente a medida que crece la cantidad de bloques, especialmente en sistemas con muchas asignaciones y liberaciones. El tamaño del heap es estático y definido al inicio. No hay soporte para crecimiento dinámico del heap, lo cual limita la flexibilidad del sistema.

En cuanto a procesos, existe un mecanismo para modificar la prioridad de un proceso de forma explícita mediante *setPriority*. Sin embargo, el sistema de ajuste de prioridades no es dinámico y no depende del comportamiento de los procesos. Otra limitación es que hay una cantidad limitada de 64 procesos.

Instrucciones de Compilación y Ejecución

Compilación y Ejecución del Kernel

1. Clonar el repositorio en una carpeta abriendo una terminal en la carpeta y corriendo el siguiente comando:
   git clone https://github.com/SebasCaules/G21_TP2_SO_1C25.git
2. Acceder a la carpeta del repositorio:
   cd G21_TP2_SO_1C25
3. Correr el siguiente comando si no tiene la imagen ya descargada en la computadora:
   docker pull agodio/itba-so-multi-platform:3.0
4. Correr el siguiente comando para acceder al contenedor:
   docker run -v ${PWD}:/root --security-opt seccomp:unconfined -ti agodio/itba-so-multi-platform:3.0
5. Dentro del docker correr estos comandos:
   a. Compilación con MM normal:
   cd root/2_Toolchain/
   make all
   cd ..
   make all
   exit
   b. Compilación con Buddy MM:
   cd root/2_Toolchain/
   make MM=”USE_BUDDY”
   cd ..
   make MM=”USE_BUDDY”
   exit

6. Luego fuera del docker correr el siguiente comando para lanzar el SO:
   ./run.sh
   (Si no tiene permisos de ejecución correr el comando *chmod +x run.sh*)

Testeo del Memory Manager afuera del kernel

1. Correr el container de docker:
   docker start <NOMBRE>
2. Lanzar la terminal del container:
   docker exec -it <NOMBRE> bash
3. Ir al directorio del Test memory:
   cd root/7_Memory_Tests
4. Compilar los archivos:
   make all
5. Correr el test:
   ./mmTest

Instrucciones para demostrar el funcionamiento de los requerimientos

- Ejecutar ***testmm <mem> &*** para testear el memory manager en el background.
- Ejecutar ***ts <proc> &*** para testear el scheduler.
- Ejecutar ***tp &*** para testear las prioridades de los procesos.
- Ejecutar ***tsem <n> <use_sem>*** para testear la implementación de sincronización, donde *n* es la cantidad de iteraciones y *use_sem* es el flag para testear con semáforos *(use_sem > 0)*, o sin semáforos *(use_sem = 0)*.

Análisis con PVS-Studio

Para testear el código con PVS-Studio, basta con correr el comando ./test_pvs-studio.sh, el cual realiza lo necesario para compilar y testear todos los archivos .c del programa, y dar el output legible en un nuevo archivo report.tasks. En el output se pueden ver los errores, warnings y notes que da el análisis realizado por la herramienta.

A continuación se presentan las warnings que resalta el análisis del código con PVS-Studio explicados:

1. /root/3_Kernel/2_libraries/scheduler.c 123 warn V557 Array overrun is possible. The value of 'pid' index could reach 65535.

Esto es imposible ya que un par de líneas antes se loopea hasta que i < MAX_PROCESSES.

2. /root/3_Kernel/4_interruptions/exceptions.c 49 warn V512 A call of the 'sys_read' function will lead to overflow of the buffer '& c'.

No puede pasar ya que al primer caracter que se lee se sale del while.

3. /root/3_Kernel/4_interruptions/exceptions.c 49 warn V1032 The pointer '& c' is cast to a more strictly aligned pointer type.
/root/3_Kernel/4_interruptions/exceptions.c 63 warn V1032 The pointer '": 0x"' is cast to a more strictly aligned pointer type.
/root/3_Kernel/4_interruptions/exceptions.c 65 warn V1032 The pointer '"0"' is cast to a more strictly aligned pointer type.
/root/3_Kernel/4_interruptions/exceptions.c 68 warn V1032 The pointer '"\\n"' is cast to a more strictly aligned pointer type.

Estas warnings son inevitables por nuestro sys_write.

El resto del output de PVS-Studio son Warnings iguales a las anteriores pero en otros archivos (aplica la misma explicación), o Notes que hace el chequeo para que ver que no esté pasando nada que no sea lo esperado, que no es el caso.

Conclusión

El desarrollo de este sistema nos permitió integrar y aplicar conceptos fundamentales de bajo nivel vistos en la materia, como manejo de memoria, procesos, scheduling, sincronización y comunicación entre procesos. A través del diseño e implementación de componentes como el scheduler, memory manager, semáforos y pipes, adquirimos una comprensión práctica de los mecanismos internos de un sistema operativo y de los desafíos asociados a la gestión de procesos, memoria y sincronización.