# Trabajo práctico Nº 2

Construcción del Núcleo de un Sistema Operativo y mecanismos de administración de recursos

# Introducción

Durante el transcurso de la materia aprendieron a usar la API de sistemas operativos de tipo UNIX, ahora crearán su propio kernel simple en base al trabajo práctico de Arquitectura de Computadoras: un kernel monolítico de 64 bits, con manejo de interrupciones básico, system calls, driver de teclado, driver de video (modo texto o gráfico) y binarios de kernel space y user space separados.

# Requerimientos

## Physical Memory Management

El kernel deberá implementar los siguientes administradores de memoria física, los cuales serán utilizados tanto por los procesos como el mismo kernel en caso de necesitarlo:

- Memory manager elegido por el grupo que soporte liberación de memoria.
- Buddy system.

Estos administradores no deberán funcionar simultáneamente, sino que se deberá decidir qué mecanismo utilizar en tiempo de compilación, por ejemplo el comando **`make`** puede compilar con el memory manager elegido por ustedes y el comando **`make buddy`** puede compilar con el buddy system. Esto se puede lograr mediante el uso de directivas de preprocesador **`#ifdef #endif`** aunque no es obligatorio.

Ambas implementaciones deberán ser intercambiables de forma transparente, para esto deberán compartir una interfaz.

### Syscalls involucradas

- Reservar y liberar memoria.
- Consultar el estado de la memoria (total, ocupada, libre y cualquier otra variable que consideren necesaria).

### Tests provistos

- **`test_mm`** - Ciclo infinito que pide y libera bloques de tamaño aleatorio, chequeando en cada iteración que los mismos no se solapan. Toma como parámetro la cantidad máxima de memoria a utilizar en bytes.

# Procesos, Context Switching y Scheduling

El sistema deberá contar con multitasking preemptivo con una cantidad variable de procesos. Para ello el sistema deberá implementar algún mecanismo que permita suspender la ejecución de un proceso y continuar la ejecución de otro (context switching) con algún algoritmo que permita seleccionar al siguiente proceso basándose en su prioridad (scheduler).

El sistema deberá implementar el siguiente algoritmo de scheduling:

- Round Robin con prioridades.

## Syscalls involucradas

- Crear y finalizar un proceso. Deberá soportar el pasaje de parámetros.
- Obtener el ID del proceso que llama.
- Listar todos los procesos: nombre, ID, prioridad, stack y base pointer, foreground y cualquier otra variable que consideren necesaria.
- Matar un proceso arbitrario.
- Modificar la prioridad de un proceso arbitrario.
- Bloquear y desbloquear un proceso arbitrario.
- Renunciar al CPU.
- Esperar a que los hijos terminen.

## Tests provistos

- **test_processes** - Ciclo infinito que crea, bloquea, desbloquea y mata procesos dummy aleatoriamente. Toma como parámetro la cantidad máxima de procesos a crear.
- **test_priority** - Crea 3 procesos que incrementan, cada uno, una variable inicializada en 0. En primera instancia los 3 procesos poseen la misma prioridad. Luego se vuelven a ejecutar con 3 prioridades diferentes a fin de visualizar diferencias en su ejecución. Toma como parámetro el valor al que deben llegar las variables para finalizar.

# Sincronización

El sistema deberá implementar semáforos y un mecanismo para que procesos no relacionados puedan compartirlos acordando un identificador a priori. La solución debe estar libre de busy waiting, deadlock, race conditions y además deberá utilizar alguna instrucción que garantice atomicidad.

## Syscalls involucradas

- Crear, abrir y cerrar semáforos.
- Modificar el valor de un semáforo.

## Tests provistos

- **test_synchro** y **test_no_synchro** - Crea una cierta cantidad de procesos que incrementan y decrementan una misma variable global a fin de visualizar que el uso de semáforos previene la ocurrencia de condiciones de carrera y el resultado final es 0. Al no usar semáforos el resultado varía de ejecución en ejecución. Toman como parámetro la cantidad de procesos a crear y la cantidad de incrementos / decrementos a realizar.

# Inter Process Communication

El sistema deberá implementar pipes unidireccionales y las operaciones de lectura y escritura sobre los mismos deberán ser bloqueantes. Todo proceso deberá ser capaz de escribir/leer tanto de un pipe como de la pantalla sin necesidad de que su código sea modificado. Notar que esto permitirá que el intérprete de comandos conecte 2 programas utilizando un pipe (ver más adelante) y a su vez, estos programas podrán ser ejecutados de forma aislada. El sistema también deberá permitir que procesos no relacionados puedan compartirlos acordando un identificador a priori.

## Syscalls involucradas

- Crear y abrir pipes.
- Leer y escribir de un pipe (notar que debe ser transparente para un proceso leer o escribir de un pipe o de la terminal)

# Drivers

Podrán usar los drivers de teclado y video implementados en Arquitectura de Computadoras. Si hiciera falta deberán implementar las system calls necesarias para aislar kernel de user space. Es indistinto si el driver de video es en modo texto o modo gráfico.

# Aplicaciones de User space

Para mostrar el cumplimiento de todos los requisitos anteriores, deberán desarrollar una serie de aplicaciones, su mayoría wrappers de system calls, que muestren el funcionamiento del sistema llamando a las distintas system calls.

Deberán implementar las siguientes aplicaciones y los nombres de las mismas deberán ser los mismos que en este listado.

- **sh**: Shell de usuario que permita ejecutar las aplicaciones.
    - Deberá contar con algún mecanismo para determinar si va a ceder o no el foreground al proceso que se ejecuta, por ejemplo, bash ejecuta un programa en background cuando se agrega el símbolo “&” al final de un comando. Este requisito es muy importante para poder demostrar el funcionamiento del sistema en general ya que en la mayoría de los casos es necesario ejecutar más de 1 proceso.
    - Deberá permitir conectar 2 procesos mediante un pipe, por ejemplo, bash hace esto al agregar el símbolo “|” entre los 2 programas a ejecutar. No es necesario que permita conectar más de 2 procesos con pipes, es decir, p1 | p2 | p3.
    - Deberá proveer soporte para Ctrl + D (envío de end of file) y Ctrl + C (matar al proceso el foreground).
- **help**: muestra una lista con todos los comandos disponibles. Deberá tener un apartado con el listado de tests provistos por la cátedra.

Physical memory management

- **mem**: Imprime el estado de la memoria.

Procesos, context switching y scheduling

- **ps**: Imprime la lista de todos los procesos con sus propiedades: nombre, ID, prioridad, stack y base pointer, foreground y cualquier otra variable que consideren necesaria.
- **loop**: Imprime su ID con un saludo cada una determinada cantidad de segundos.
- **kill**: Mata un proceso dado su ID.
- **nice**: Cambia la prioridad de un proceso dado su ID y la nueva prioridad.
- **block**: Cambia el estado de un proceso entre *bloqueado* y *listo* dado su ID.

Inter process communication

- **cat**: Imprime el stdin tal como lo recibe.
- **wc**: Cuenta la cantidad de líneas del input.
- **filter**: Filtra las vocales del input.
- **mvar:** Implementa el problema de múltiples lectores y escritores sobre una variable global, similar a una MVar de Haskell. Toma como parámetros la cantidad de escritores y lectores. Cada escritor realiza una espera activa aleatoria, luego espera a que la variable esté vacía y finalmente escribe un valor único (por ejemplo, 'A', 'B', 'C', etc.). Cada lector realiza una espera activa aleatoria, luego espera a que la

variable tenga un valor para leer y finalmente lo consume e imprime junto con un identificador único (por ejemplo, un color). De esta forma, se simula el comportamiento de una MVar, garantizando que solo un proceso accede a la variable a la vez y que los accesos están correctamente sincronizados. El proceso principal debe terminar inmediatamente después de crear los lectores y escritores.

| Comando | Acción tomada | Salida estándar |
|---|---|---|
| mvar 2 2 | Ninguna, se lo deja correr. | ABABABABA |
| mvar 2 3 | Ninguna, se lo deja correr. | ABABABABA |
| mvar 3 2 | Ninguna, se lo deja correr. | ABCABCABC |
| mvar 2 2 | Se mata al escritor B tras un tiempo | ABABAAAAA |
| mvar 2 2 | Se mata al lector rojo tras un tiempo | ABABABABAB |
| mvar 2 1 | Se le sube la prioridad al escritor B tras un tiempo | ABABABBBABBBABBBABB |

# Entrega

Cada entrega es grupal y se realiza a través del campus donde se podrá adjuntar el link del repositorio especificando el hash del commit y la rama correspondiente a la entrega. Recuerden proveer los permisos necesarios en caso de que el repositorio sea privado.

# Evaluación

La evaluación incluye y no se limita a los siguientes puntos:

- [1] Deadline.
- [5] Funcionalidad.
- [3] Calidad de código.
- [1] README
- Defensa (grupal con nota individual)

El proyecto deberá cumplir los siguientes requerimientos de forma excluyente, es decir, deberá recuperarse en caso de no cumplirse alguno de ellos:

- Los tests **test_mm, test_processes, test_synchro y test_no_synchro** deberán ejecutarse correctamente como procesos de usuario (no built-ins) en foreground y background. En el caso de **test_mm**, al menos uno de los memory managers solicitados deberá pasar el test **test_mm**.

# Entorno de compilación

Es un requisito obligatorio para la compilación, utilizar la imagen provista por la cátedra:

```
docker pull agodio/itba-so-multi-platform:3.0
```

# README

El repositorio deberá poseer un README en formato markdown y desarrollar de forma breve los ítems listados a continuación:

- Instrucciones de compilación y ejecución.
- Instrucciones de replicación:
    - Nombre preciso y breve descripción de cada comando / test y parámetros que admiten.
    - Caracteres especiales para pipes y comandos en background.
    - Atajos de teclado para interrumpir la ejecución y enviar EOF.
    - Ejemplos, por fuera de los tests, para demostrar el funcionamiento de cada requerimiento.
    - Requerimientos faltantes o parcialmente implementados.

- Limitaciones.
- Citas de fragmentos de código / uso de IA

## Consideraciones generales

- El único mecanismo mediante el cual los procesos se comunican con el kernel deberá ser a través de las system calls.
- El sistema deberá estar libre de deadlocks, race conditions y busy waiting.
- El código deberá tener un Makefile para las tareas de compilación.
- Para el desarrollo deberán utilizar algún servicio de control de versiones desde el principio del desarrollo. No se admitirán repositorios sin un historial desde el comienzo del TP.
- El repositorio utilizado no deberá tener binarios ni archivos de prueba.
- La compilación con todos los warnings activados (-Wall) no deberá arrojar warnings en el código desarrollado tanto en Arquitecturas como en Sistemas Operativos.
- El análisis con PVS-studio no deberá reportar errores en el código desarrollado tanto en Arquitecturas como en Sistemas Operativos. En caso de falsos positivos, justificar en el informe.