Sistemas Operativos

Trabajo Práctico Nro. 2 (obligatorio): Construcción del Núcleo de un Sistema Operativo y estructuras de administración de recursos.

Introducción

Durante el transcurso de la materia aprendieron a usar la API de sistemas operativos de tipo UNIX, ahora en cambio crearán su propio kernel simple en base al trabajo práctico final de la materia anterior, Arquitectura de Computadoras. Para ello deberán implementar Memory Management, procesos, scheduling, mecanismos de IPC y sincronización.

Grupos

Se realizará con los grupos ya establecidos.

Requisitos Previos

DEBEN contar con una versión funcional del 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 en modo texto o gráfico y binarios de kernel space y user space separados. Si el sistema tenía problemas de memoria, como por ejemplo strings corruptos, **DEBERÁN** arreglarlos antes de comenzar el trabajo. El proyecto **DEBERÁ** compilar en la imagen de Docker provista por la cátedra.

Requerimientos

El kernel **DEBERÁ** implementar las siguientes features. Se les recomienda **FUERTEMENTE** implementarlas en el orden en el que son enumeradas.

System calls

Las system calls son la interface entre user y kernel space. El sistema **DEBERÁ** proveer system calls para que los procesos (explicados más adelante) interactúen con el kernel. Utilice exactamente el mismo mecanismo desarrollado en Arquitectura de Computadoras (interrupciones de software).

Physical Memory Management

El kernel **DEBERÁ** implementar los siguientes administradores de memoria física:

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

Tanto el kernel como los procesos pueden utilizar estos administradores de memoria física, además, **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.

Ambas implementaciones **DEBERÁN** ser intercambiables de forma transparente, para esto **DEBERÁN** compartir una interfaz. Esto implica que **NO DEBEN** existir construcciones como la siguiente dentro del código:

```
if (buddy){
  …
} else {
  …
}
```

Sin embargo, se admitirán construcciones como la siguiente si así lo prefieren:

```
#ifdef BUDDY
  …
#else
  …
#endif
```

El sistema también **DEBERÁ** permitir consultar el estado de la memoria: memoria total, ocupada, libre y cualquier otra variable que consideren necesaria.

Syscalls involucradas

- Reservar y liberar memoria.
- Consultar el estado de la memoria.

Aplicaciones de test provistas

- test_mm

Procesos, Context Switching y Scheduling

El sistema **DEBERÁ** contar con multitasking pre-emptivo 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:

- Priority-based round Robin.

El sistema también **DEBERÁ** proveer los siguientes servicios:

- Crear 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

Syscalls involucradas

- Crear y finalizar procesos.
- Obtener el ID del proceso que llama.
- Listar procesos.
- Matar un proceso.
- Modificar la prioridad de un proceso.
- Cambiar el estado de un proceso.
- Renunciar al CPU

Aplicaciones de test provistas

- test_processes
- test_priority

Sincronización

El sistema **DEBERÁ** implementar semáforos y un mecanismo para que procesos no relacionados puedan compartirlos acordando un identificador a priori. El sistema también **DEBERÁ** permitir listar todos los semáforos activos del sistema, su estado, los procesos bloqueados en cada uno y cualquier otra variable que consideren necesaria. 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 un semáforo.
- Modificar el valor de un semáforo.
- Listar el estado de todos los semáforos del sistema.

Aplicaciones de test provistas

- test_synchro
- test_no_synchro

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.

El sistema también **DEBERÁ** permitir listar todos los pipes activos del sistema, su estado y los procesos bloqueados en cada uno.

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)
- Listar el estado de todos los pipes del sistema.

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. Las aplicaciones resaltadas en **verde** pueden ser built-ins de la shell y no es necesario que se puedan conectar con pipes.

- **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. La shell también **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 ...
- **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.

Sincronización

- **sem:** Imprime la lista de todos los semáforos con sus propiedades: estado, los procesos bloqueados en cada uno y cualquier otra variable que consideren necesaria.

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.
- **pipe:** Imprime la lista de todos los pipes con sus propiedades: estado, los procesos bloqueados en cada uno y cualquier otra variable que consideren necesaria.

- **phylo:** Implementa el problema de los filósofos comensales. **DEBERÁ** permitir cambiar la cantidad de filósofos en runtime con las teclas “a” (add 1) y “r” (remove 1). También **DEBERÁ** mostrar el estado de la mesa de forma sencilla pero legible, por ejemplo, cada línea a continuación corresponde a una actualización del estado de la mesa con 5 filósofos. La letra “E” y el punto “.” representan un filósofo comiendo y esperando respectivamente. Como se puede observar, en ningún momento hay 2 filósofos contiguos comiendo y todos los filósofos comen, eventualmente.

```
E . . E .
. . . E .
. E . . E
. . E . E
```

Es muy importante que agreguen las aplicaciones que consideren necesarias para demostrar el funcionamiento de cada una de las capacidades del sistema, de otra forma la existencia de las mismas no podrá ser evaluada.

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 tarea de compilación, **DEBERÁ** aprovechar el potencial de Makefile.
- 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 y cppcheck **NO DEBERÁ** reportar errores en el código desarrollado tanto en Arquitecturas como en Sistemas Operativos. En caso de falsos positivos, justificar en el informe.

Informe

DEBERÁN presentar un informe **EN FORMATO PDF**, en el cual se desarrollen **DE FORMA BREVE**, los siguientes puntos:

- Decisiones tomadas durante el desarrollo, por ejemplo, detalles del algoritmo de scheduling y administrador de memoria
- Instrucciones de compilación y ejecución.
- Pasos a seguir para demostrar el funcionamiento de cada uno de los requerimientos.
- Limitaciones.
- Problemas encontrados durante el desarrollo y cómo se solucionaron.
- Citas de fragmentos de código reutilizados de otras fuentes.
- Modificaciones realizadas a las aplicaciones de test provistas, en caso de que su sistema no soporte alguna funcionalidad.

Entorno de compilación

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

```
docker pull agodio/itba-so:1.0
```

Evaluación

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

- [1] Deadline.
- [5] Funcionalidad (Mandatorio).
- [3] Calidad de código.
- [1] Informe.
- Defensa.

Entrega

Fecha: TBD hasta las 23:59.
Se habilitará la actividad “TP2” en el campus donde se podrá subir el material requerido. En caso de tener algún problema con la entrega en Campus, enviarlo por mail a los docentes a cargo de la clase práctica.
Entregables: Link del repositorio **ESPECIFICANDO** el hash del commit **Y** la rama correspondiente a la entrega y el informe. Recuerden proveer los permisos necesarios en caso de que el repositorio sea privado.
Defensa del trabajo práctico: Individual y obligatoria.