T.U.K.I

The Ultimate Kernel Implementation

No será la mejor implementación, pero el nombre quedaba fachero

Queríamos la 3ra, y la conseguimos

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

-1C2023 -
Versión 1.1

Índice

Historial de Cambios

v1.0 (03/04/2023) Release inicial de trabajo práctico
v1.1 (29/04/2023) Agregada implementación esperada de Yield en el módulo Kernel
Agregada aclaración acerca de los archivos de pseudocódigo de la consola
Agregada aclaración que el FS atiende peticiones de a una

Objetivos del Trabajo Práctico

Mediante la realización de este trabajo se espera que el alumno:

- Adquiera conceptos prácticos del uso de las distintas herramientas de programación e interfaces (APIs) que brindan los sistemas operativos.
- Entienda aspectos del diseño de un sistema operativo.
- Afirme diversos conceptos teóricos de la materia mediante la implementación práctica de algunos de ellos.
- Se familiarice con técnicas de programación de sistemas, como el empleo de makefiles, archivos de configuración y archivos de log.
- Conozca con grado de detalle la operatoria de Linux mediante la utilización de un lenguaje de programación de relativamente bajo nivel como C.

Características

- Modalidad: grupal (5 integrantes ± 0) y obligatorio
- Fecha de comienzo: 03/04/2023
- Fecha de primera entrega: 15/07/2023
- Fecha de segunda entrega: 29/07/2023
- Fecha de tercera entrega: 05/08/2023
- Lugar de corrección: Laboratorio de Sistemas - Medrano.

Evaluación del Trabajo Práctico

El trabajo práctico consta de una evaluación en 2 etapas.

La primera etapa consistirá en las pruebas de los programas desarrollados en el laboratorio. Las pruebas del trabajo práctico se subirán oportunamente y con suficiente tiempo para que los alumnos puedan evaluarlas con antelación. Queda aclarado que para que un trabajo práctico sea considerado evaluable, el mismo debe proporcionar registros de su funcionamiento de la forma más clara posible.

La segunda etapa se dará en caso de aprobada la primera y constará de un coloquio, con el objetivo de afianzar los conocimientos adquiridos durante el desarrollo del trabajo práctico y terminar de definir la nota de cada uno de los integrantes del grupo, por lo que se recomienda que la carga de trabajo se distribuya de la manera más equitativa posible.

Cabe aclarar que el trabajo equitativo no asegura la aprobación de la totalidad de los integrantes, sino que cada uno tendrá que defender y explicar tanto teórica como prácticamente lo desarrollado y aprendido a lo largo de la cursada.

La defensa del trabajo práctico (o coloquio) consta de la relación de lo visto durante la teoría con lo implementado. De esta manera, una implementación que contradiga lo visto en clase o lo escrito en el documento *es motivo de desaprobación del trabajo práctico*. Esta etapa al ser la conclusión del todo el trabajo realizado durante el cuatrimestre *no es recuperable*.

Deployment y Testing del Trabajo Práctico

Al tratarse de una plataforma distribuida, los procesos involucrados podrán ser ejecutados en diversas computadoras. La cantidad de computadoras involucradas y la distribución de los diversos procesos en estas será definida en cada uno de los tests de la evaluación y es posible cambiar la misma en el momento de la evaluación. Es responsabilidad del grupo automatizar el despliegue de los diversos procesos con sus correspondientes archivos de configuración para cada uno de los diversos tests a evaluar.

Todo esto estará detallado en el documento de pruebas que se publicará cercano a la fecha de Entrega Final. Archivos y programas de ejemplo se pueden encontrar en el repositorio de la cátedra.

Finalmente, es mandatoria la lectura y entendimiento de las Normas del Trabajo Práctico donde se especifican todos los lineamientos de cómo se desarrollará la materia durante el cuatrimestre.

Aclaraciones

Debido al fin académico del trabajo práctico, los conceptos reflejados son, en general, versiones simplificadas o alteradas de los componentes reales de hardware y de sistemas operativos vistos en las clases, a fin de resaltar aspectos de diseño o simplificar su implementación.

Invitamos a los alumnos a leer las notas y comentarios al respecto que haya en el enunciado, reflexionar y discutir con sus compañeros, ayudantes y docentes al respecto.

Definición del Trabajo Práctico

Esta sección se compone de una introducción y definición de carácter global sobre el trabajo práctico. Posteriormente se explicarán por separado cada uno de los distintos módulos que lo componen, pudiéndose encontrar los siguientes títulos:

- Lineamiento e Implementación: Todos los títulos que contengan este nombre representarán la definición de lo que deberá realizar el módulo y cómo deberá ser implementado. La no inclusión de alguno de los puntos especificados en este título puede conllevar a la desaprobación del trabajo práctico.
- Archivos de Configuración: En este punto se da un archivo modelo y que es lo mínimo que se pretende que se pueda parametrizar en el proceso de forma simple. En caso de que el grupo requiera de algún parámetro extra, podrá agregarlo.

Cabe destacar que en ciertos puntos de este enunciado se explicarán exactamente cómo deben ser las funcionalidades a desarrollar, mientras que en otros no se definirá específicamente, quedando su implementación a decisión y definición del equipo. Se recomienda en estos casos siempre consultar en el foro de github.

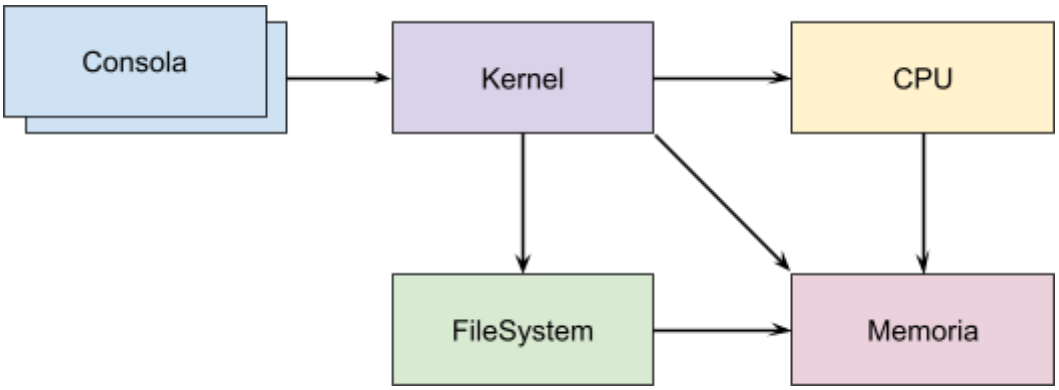
¿Qué es el trabajo práctico y cómo empezamos?

El objetivo del trabajo práctico consiste en desarrollar una solución que permita la simulación de un sistema distribuido, donde los grupos tendrán que planificar procesos, resolver peticiones al sistema y administrar de manera adecuada una memoria y un sistema de archivos bajo los esquemas explicados en sus correspondientes módulos.

Para el desarrollo del mismo se decidió la creación de un sistema bajo la metodología Iterativa Incremental donde se solicitarán en una primera instancia la implementación de ciertos módulos para luego poder realizar una integración total con los restantes.

Recomendamos seguir el lineamiento de los distintos puntos de control que se detallan al final de este documento para su desarrollo. Estos puntos están planificados y estructurados para que sean desarrollados a medida y en paralelo a los contenidos que se ven en la parte teórica de la materia. *Cabe aclarar que esto es un lineamiento propuesto por la cátedra y no implica impedimento alguno para el alumno de realizar el desarrollo en otro orden diferente al especificado.*

Arquitectura del sistema



El trabajo práctico consiste de 5 módulos: **Consola** (múltiples instancias), **Kernel**, **CPU, Memoria y File System** (1 instancia cada uno).

El proceso de ejecución del mismo consiste en crear *procesos*[1] a través del módulo **Consola**, los cuales enviarán la información necesaria al módulo **Kernel** para que el mismo pueda crear las estructuras necesarias a fin de administrar y planificar su ejecución mediante diversos algoritmos.

Estos procesos serán ejecutados en el módulo **CPU**, quien interpretará sus instrucciones y hará las peticiones necesarias a **Memoria** y/o al **Kernel**.

La **Memoria** administrará el espacio de memoria (valga la redundancia) de estos procesos implementando un esquema de segmentación y respondiendo a las peticiones de **CPU**, **Kernel** y **File System.**

El **File System** implementará un esquema indexado, tomando algunas características de un File System tipo Unix o ext2. El mismo estará encargado de administrar y persistir los archivos creados por los procesos que corren en el sistema, respondiendo a las peticiones de **Kernel** y haciendo las peticiones necesarias a **Memoria**

Una vez que un proceso finalice tras haber sido ejecutadas todas sus instrucciones, el Kernel devolverá un mensaje de finalización a su **Consola** correspondiente y cerrará la conexión.

Distribución Recomendada

Estimamos que a lo largo del cuatrimestre la carga de trabajo para cada módulo será la siguiente:

- Consola: **5%**
- Kernel: **45%**
- CPU: **10%**
- Memoria: **20%**
- FileSystem: **20%**

[1]Estos *procesos* son "simbólicos", en el sentido de que representan programas en ejecución para nuestros módulos, no procesos reales del Sistema Operativo en el que se ejecute el trabajo práctico.

Dado que se contempla que los conocimientos se adquieran a lo largo de la cursada, se recomienda que el trabajo práctico se realice siguiendo un esquema iterativo incremental, por lo que por ejemplo la memoria no necesariamente tendrá avances hasta pasado el primer parcial.

Aclaración Importante

*Será condición necesaria de aprobación demostrar conocimiento teórico y de trabajo en alguno de los módulos principales (**Kernel, Memoria o File System**).*

Desarrollar únicamente temas de conectividad, serialización, sincronización, o el módulo Consola, es insuficiente para poder entender y aprender los distintos conceptos de la materia. Dicho caso será un motivo de desaprobación directa.

En caso de haber trabajado exclusivamente en el módulo CPU se deberá demostrar participación en otro módulo, ya que el mismo no implementa suficientes aspectos teóricos.

Cada módulo contará con un listado de **logs mínimos y obligatorios**, pudiendo ser extendidos por necesidad del grupo en un archivo aparte.

De no cumplir con los logs mínimos, el trabajo práctico *no se considera apto para ser evaluado* y por consecuencia se considera *desaprobado*.

Módulo: Consola

El módulo consola, será el punto de partida de los diferentes procesos que se crearán dentro de esta simulación de un kernel. Cada instancia de dicho módulo ejecutará un solo proceso.

Para poder iniciar una consola, la misma deberá poder recibir por parámetro los siguientes datos:

- Archivo de configuración
- Archivo de pseudocódigo con las instrucciones a ejecutar.

Lineamiento e Implementación

El módulo **consola** deberá recibir la ruta (path) de los 2 archivos mencionados anteriormente (configuración y pseudocódigo)[2].

Al iniciar, el módulo leerá ambos archivos y parseará cada línea terminada en `"\n"` del archivo de pseudocódigo como una instrucción para generar el listado de instrucciones.

Luego se conectará al kernel y enviará la información correspondiente al listado de instrucciones. Una vez recibida la confirmación de recepción, la consola quedará a la espera del mensaje del Kernel que indique la finalización del mismo.

Archivo de pseudocódigo

Este archivo contendrá una serie de líneas terminadas en `"\n"` las cuales deben ser parseadas para ser transformadas en instrucciones que luego se enviarán al Kernel.

Cada línea tendrá el formato “INSTRUCCIÓN parámetros”. Los parámetros serán siempre valores separados por espacios. Según la instrucción, cada línea puede tener entre 0 y 3 parámetros.

Dicho esto, las posibles instrucciones a parsear serán las siguientes:

- **F_READ, F_WRITE**: 3 parámetros
- **SET, MOV_IN, MOV_OUT, F_TRUNCATE, F_SEEK, CREATE_SEGMENT**: 2 parámetros.
- **I/O, WAIT, SIGNAL, F_OPEN, F_CLOSE, DELETE_SEGMENT**: 1 parámetro.
- **EXIT, YIELD**: 0 parámetros.

El comportamiento de cada una de estas instrucciones está detallada en el módulo CPU que será el que finalmente deba ejecutarlas.

Ninguna línea contendrá errores sintácticos ni semánticos, aunque puede ocurrir que haya saltos de línea vacíos (es decir, dos o más caracteres `'\n'` consecutivos, o uno al final).

[2] Se recomienda utilizar los parámetros `argc` y `argv` de la función `main` ayudándose con esta guía.

Ejemplo de líneas a parsear

```
SET AX HOLA
MOV_OUT 120 AX
WAIT DISCO
I/O 10
SIGNAL DISCO
MOV_IN BX 120
F_OPEN ARCHIVO
YIELD
F_TRUNCATE ARCHIVO 64
F_SEEK ARCHIVO 10
CREATE_SEGMENT 1 128
F_WRITE ARCHIVO 4 4
F_READ ARCHIVO 16 4
DELETE_SEGMENT 1
F_CLOSE ARCHIVO
EXIT
```

Archivo de configuración

Campo	Tipo	Descripción
IP_KERNEL	String	IP del Kernel al cual debe conectarse
PUERTO_KERNEL	Numérico	Puerto del Kernel al cual debe conectarse

Ejemplo de Archivo de Configuración

```
IP_KERNEL=127.0.0.1
PUERTO_KERNEL=8000
```

Módulo: Kernel

El módulo **Kernel,** dentro de nuestro trabajo práctico, será el encargado de gestionar la ejecución de los diferentes procesos que ingresen al sistema mediante las instancias del módulo **Consola**, planificando su ejecución en la CPU del sistema.

Lineamiento e Implementación

Este módulo deberá mantener una conexión activa con la CPU, la Memoria, File System y las diferentes consolas que se conecten. Para ello, deberá implementarse mediante una estrategia de múltiples hilos de ejecución.

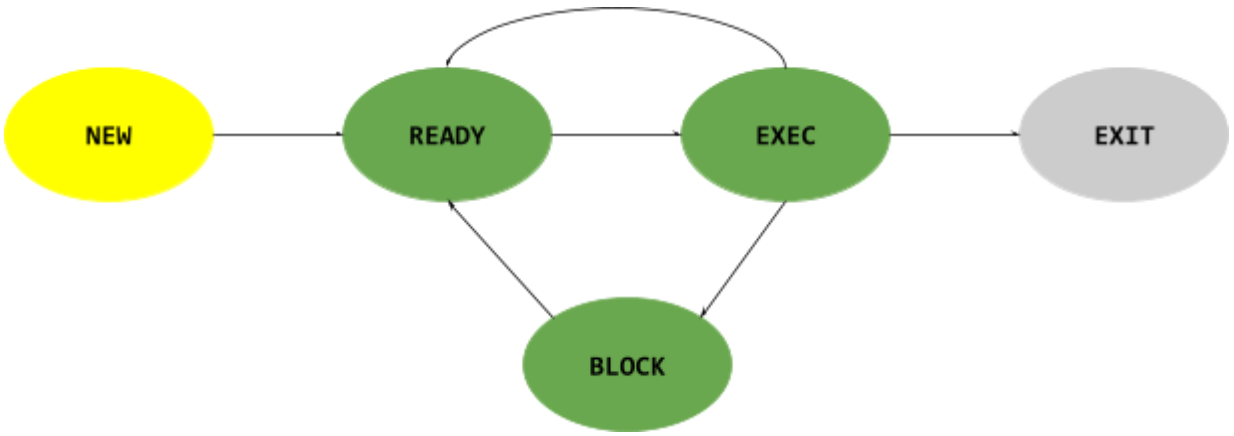
Cualquier fallo en las comunicaciones deberá ser informado mediante un mensaje apropiado en el archivo de log y el sistema deberá finalizar su ejecución.

Diagrama de estados

El kernel utilizará el diagrama de 5 estados para ordenar la planificación de los procesos. La implementación de cada una de las transiciones será detallada en la sección del planificador correspondiente.

Dentro del estado BLOCK, se tendrán **múltiples colas.**

- Todas las operaciones de I/O se realizarán de forma paralela (como si existiese un dispositivo de I/0 dedicado por cada proceso en ejecución), por lo que no deben encolarse en ningún momento por este motivo. La cantidad de tiempo que un proceso pase bloqueado por I/O lo informará la CPU al momento de devolver el *Contexto de Ejecución* por este motivo.
- Tendremos distintos recursos definidos por archivo de configuración, donde cada uno tiene su propia cola de procesos bloqueados, para más detalles de implementación ver la sección “Manejo de recursos”.



PCB

El PCB será la estructura base que utilizaremos dentro del Kernel para administrar los procesos lanzados por medio de las consolas. El mismo deberá contener como mínimo los datos definidos a continuación que representan la información administrativa necesaria y el *Contexto de Ejecución* del proceso que se deberá enviar a la CPU a través de la conexión de *dispatch* al momento de poner a ejecutar un proceso, *pudiéndose extender esta estructura con más datos que requiera el grupo*.

- **PID**: Identificador del proceso (deberá ser un número entero, único en todo el sistema).
- **Instrucciones**: Lista de instrucciones a ejecutar.[3]
- **Program_counter**: Número de la próxima instrucción a ejecutar.
- **Registros de la CPU**: Estructura que contendrá los valores de los *registros de uso general* de la CPU.
- **Tabla de Segmentos**: Contendrá ids, direcciones base y tamaños de los segmentos de datos del proceso.
- **Estimado de próxima ráfaga**: Estimación utilizada para planificar los procesos en el algoritmo HRRN, la misma tendrá un valor inicial definido por archivo de configuración y será recalculada bajo la fórmula de promedio ponderado vista en clases.
- **Tiempo de llegada a ready**: Timestamp en que el proceso llegó a ready por última vez (utilizado para el cálculo de tiempo de espera del algoritmo HRRN).
- **Tabla de archivos abiertos**: Contendrá la lista de archivos abiertos del proceso con la posición del puntero de cada uno de ellos.

Planificador de Largo Plazo

Al conectarse una Consola al Kernel, deberá generarse la estructura PCB detallada anteriormente y asignarse este proceso al estado **NEW**.

En caso de que el grado máximo de multiprogramación lo permita, los procesos pasarán al estado **READY**, enviando un mensaje al módulo Memoria para que inicialice sus estructuras necesarias y obtenga la tabla de segmentos inicial que deberá ser almacenada en el **PCB**. La salida de **NEW** será mediante el algoritmo **FIFO**.

Cuando se reciba un mensaje de CPU con motivo de finalizar el proceso, se deberá pasar al mismo al estado **EXIT**, liberar todos los recursos que tenga asignados y dar aviso al módulo Memoria para que éste libere sus estructuras. Una vez hecho esto, se dará aviso a la Consola de la finalización del proceso.

Planificador de Corto Plazo

Los procesos que estén en estado **READY** serán planificados mediante uno de los siguientes algoritmos:

- **FIFO**: First in First out.
- **HRRN**: Highest Response Ratio Next (sin desalojo)

[3] A diferencia de la realidad donde el código de un proceso estaría en Memoria, tendremos las instrucciones dentro del PCB para facilitar la implementación del trabajo práctico en los tiempos del cuatrimestre y de forma incremental con respecto a los temas que se ven en clase.

Al recibir el *Contexto de Ejecución* del proceso en ejecución, en caso de que se deba replanificar se seleccionará el siguiente proceso a ejecutar según indique el algoritmo. Durante este período la CPU se quedará esperando el nuevo contexto.

Al recibir de CPU una instrucción YIELD, el proceso será enviado al final de la cola **READY**.
En caso de utilizar el algoritmo HRRN, se deberá además calcular un nuevo estimado para su próxima ráfaga utilizando la fórmula de promedio ponderado vista en clases.

Manejo de Recursos

Los recursos del sistema vendrán indicados por medio del archivo de configuración, donde se encontrarán 2 variables con la información inicial de los mismos:

- La primera llamada RECURSOS, la cual listará los nombres de los recursos disponibles en el sistema.
- La segunda llamada INSTANCIAS_RECURSOS será la cantidad de instancias de cada recurso del sistema, y estarán ordenadas de acuerdo a la anterior lista (ver ejemplo).

A la hora de recibir de la CPU un *Contexto de Ejecución* desplazado por WAIT, el Kernel deberá verificar primero que exista el recurso solicitado y en caso de que exista restarle 1 a la cantidad de instancias del mismo. En caso de que el número sea estrictamente menor a 0, el proceso que realizó WAIT se bloqueará en la cola de bloqueados correspondiente al recurso.

A la hora de recibir de la CPU un *Contexto de Ejecución* desplazado por SIGNAL, el Kernel deberá verificar primero que exista el recurso solicitado y luego sumarle 1 a la cantidad de instancias del mismo. En caso de que corresponda, desbloquea al primer proceso de la cola de bloqueados de ese recurso. Una vez hecho esto, se devuelve la ejecución al proceso que peticionó el SIGNAL.

Para las operaciones de WAIT y SIGNAL donde el recurso no exista, se deberá enviar el proceso a EXIT.

Manejo de Memoria

La memoria de los procesos se solicita por medio de operaciones del Kernel, ya que el mismo conoce todos los segmentos de todos los procesos y es quien posee los privilegios[4] para poder administrar todo el espacio de memoria.

En esta implementación, el módulo Kernel recibirá el *Contexto de Ejecución* de la CPU con motivo de ejecutar un CREATE_SEGMENT o un DELETE_SEGMENT y realizará la rutina correspondiente manteniendo el proceso en estado EXEC[5].

Para realizar un CREATE_SEGMENT, el Kernel deberá enviarle a la Memoria el mensaje para crear un segmento con el tamaño definido, en este punto pueden ocurrir 3 cosas:

[4] No se tiene una implementación de privilegios como tal, pero se respeta lo visto en la teoría.
[5] En un sistema real representarían llamadas al sistema no bloqueantes, donde se lanzaría un *cambio de contexto* y la CPU ejecutaría la rutina en modo privilegiado (modo kernel).

1. Que el segmento se cree exitosamente y que la memoria nos devuelva la base del nuevo segmento.
2. Que no se tenga más espacio disponible en la memoria y por lo tanto el proceso tenga que finalizar con error **Out of Memory**.
3. Que se tenga el espacio disponible, pero que el mismo no se encuentre contiguo, por lo que se deba compactar, este caso lo vamos a analizar más en detalle, ya que involucra controlar las operaciones de File System que se estén ejecutando.

Aclaraciones:

- No se solicitará nunca la creación de un segmento con tamaño mayor al máximo configurado en el sistema.
- No se solicitará la creación de un segmento de un Id inválido (0 o mayor o igual al Id máximo).
- No se solicitará la creación de un segmento que se encuentre con un tamaño distinto a 0 (es decir, ya creado)

Para realizar un DELETE_SEGMENT, el Kernel deberá enviarle a la Memoria el Id del segmento a eliminar y recibirá como respuesta de la Memoria la tabla de segmentos actualizada.

Nota: No se solicitará nunca la eliminación del segmento 0 o de un segmento inexistente.

En ambos casos, al finalizar la rutina, se deberá devolver el *Contexto de ejecución* a la CPU para que esta continúe con la ejecución del proceso.

Compactación de Memoria

Al momento de recibir de la Memoria el mensaje que indica que se dispondría del espacio solicitado post compactación, el Kernel deberá primero validar que no se estén ejecutando operaciones entre el File System y la Memoria (F_READ y F_WRITE).

En caso de que se tengan operaciones del File System en curso, el Kernel deberá esperar la finalización de las mismas para luego proceder a solicitar la compactación a la Memoria.

Como resultado de la compactación, el Kernel recibirá las *tablas de segmentos* actualizadas de todos los procesos y deberá actualizar las mismas en los PCB de cada proceso.

Una vez terminada esta rutina, el Kernel repetirá la solicitud de creación del segmento.

Manejo de File System

El Kernel, al igual que en un kernel monolítico, es el encargado de orquestar las llamadas al File System, es por esto que las acciones que se pueden realizar en cuanto a los archivos van a venir por parte de la CPU como llamadas por medio de las funciones correspondientes a archivos.

Para la gestión de los archivos, el Kernel deberá implementar una **tabla global de archivos abiertos**, a fin de poder gestionar los mismos como si fueran un *recurso* de una única instancia. Cada archivo podrá estar abierto por solo un proceso a la vez, por lo que se deberá implementar **locks exclusivos** y **obligatorios** sobre cada archivo.

Para ejecutar estas funciones el Kernel va a recibir el *Contexto de Ejecución* de la CPU, el accionar de cada función va a depender de la misma:

- **F_OPEN**: Esta función será la encargada de abrir o crear el archivo pasado por parámetro. Dado que el Kernel maneja una tabla global de archivos abiertos y una tabla por cada proceso, es posible que se den los siguientes escenarios:
    - La tabla global de archivos abiertos tenga este archivo, se agrega la entrada en la tabla de archivos abiertos del proceso con el puntero en la posición 0 y se bloqueará al proceso que ejecutó F_OPEN en la cola correspondiente a este archivo.
    - El archivo no esté presente en la tabla global de archivos abiertos, se deberá consultar al módulo File System si existe o no el archivo.
        - En caso de que el archivo no exista, primero se le deberá solicitar al File System que cree dicho archivo con tamaño 0.
        - En ambos casos se agrega la entrada a la tabla global de archivos abiertos y se agrega a la tabla de archivos abiertos del proceso con el puntero en la posición 0. Se deberá devolver el contexto de ejecución a la CPU para que continúe el mismo proceso.
- **F_CLOSE**: Esta función se encargará de quitar la entrada correspondiente al archivo recibido por parámetro de la tabla de archivos abiertos del proceso, en este punto pueden ocurrir 2 cosas:
    - Hay procesos bloqueados esperando por este archivo. En este caso, se deberá desbloquear al primer proceso bloqueado en la cola del archivo.
    - No queda ningún proceso que tenga abierto el archivo, por lo que se deberá eliminar la entrada también de la tabla global de archivos abiertos.
- **F_SEEK**: Actualiza el puntero del archivo en la tabla de archivos abiertos hacia la ubicación pasada por parámetro. Se deberá devolver el contexto de ejecución a la CPU para que continúe el mismo proceso.
- **F_TRUNCATE**: Esta función solicitará al módulo File System que actualice el tamaño del archivo al nuevo tamaño pasado por parámetro y bloqueará al proceso hasta que el File System informe de la finalización de la operación.
- **F_READ**: Para esta función se solicita al módulo File System que lea desde el puntero del archivo pasado por parámetro la cantidad de bytes indicada y lo grabe en la **dirección física** de memoria recibida por parámetro. El proceso que llamó a F_READ, deberá permanecer en estado bloqueado hasta que el módulo File System informe de la finalización de la operación.
- **F_WRITE**: Esta función solicitará al módulo File System que escriba en el archivo desde la **dirección física** de memoria recibida por parámetro la cantidad de bytes indicada. El proceso que llamó a F_WRITE, deberá permanecer en estado bloqueado hasta que el módulo File System informe de la finalización de la operación.

Nota: Es importante aclarar que para las operaciones de F_READ y F_WRITE siempre se van a pasar tamaños y punteros válidos.

Logs mínimos y obligatorios

Creación de Proceso: “Se crea el proceso <PID> en NEW”

Fin de Proceso: “Finaliza el proceso <PID> - Motivo: <SUCCESS / SEG_FAULT / INVALID_RESOURCE / OUT_OF_MEMORY>”

Cambio de Estado: “PID: <PID> - Estado Anterior: <ESTADO_ANTERIOR> - Estado Actual: <ESTADO_ACTUAL>”

Motivo de Bloqueo: “PID: <PID> - Bloqueado por: <IO / NOMBRE_RECURSO / NOMBRE_ARCHIVO>”

I/O: “PID: <PID> - Ejecuta IO: <TIEMPO>”

Ingreso a Ready: “Cola Ready <ALGORITMO>: [<LISTA DE PIDS>]”

Wait: “PID: <PID> - Wait: <NOMBRE RECURSO> - Instancias: <INSTANCIAS RECURSO>”
Nota: El valor de las instancias es después de ejecutar el Wait

Signal: “PID: <PID> - Signal: <NOMBRE RECURSO> - Instancias: <INSTANCIAS RECURSO>”
Nota: El valor de las instancias es después de ejecutar el Signal

Crear Segmento: “PID: <PID> - Crear Segmento - Id: <ID SEGMENTO> - Tamaño: <TAMAÑO>”

Eliminar Segmento: “PID: <PID> - Eliminar Segmento - Id Segmento: <ID SEGMENTO>”

Inicio Compactación: “Compactación: <Se solicitó compactación / Esperando Fin de Operaciones de FS>”

Fin Compactación: “Se finalizó el proceso de compactación”

Abrir Archivo: “PID: <PID> - Abrir Archivo: <NOMBRE ARCHIVO>”

Cerrar Archivo: “PID: <PID> - Cerrar Archivo: <NOMBRE ARCHIVO>”

Actualizar Puntero Archivo: “PID: <PID> - Actualizar puntero Archivo: <NOMBRE ARCHIVO> - Puntero <PUNTERO>” **Nota**: El valor del puntero debe ser luego de ejecutar F_SEEK.

Truncar Archivo: “PID: <PID> - Archivo: <NOMBRE ARCHIVO> - Tamaño: <TAMAÑO>”

Leer Archivo: “PID: <PID> - Leer Archivo: <NOMBRE ARCHIVO> - Puntero <PUNTERO> - Dirección Memoria <DIRECCIÓN MEMORIA> - Tamaño <TAMAÑO>”

Escribir Archivo: “PID: <PID> - Escribir Archivo: <NOMBRE ARCHIVO> - Puntero <PUNTERO> - Dirección Memoria <DIRECCIÓN MEMORIA> - Tamaño <TAMAÑO>”

Archivo de configuración

Campo	Tipo	Descripción
IP_MEMORIA	String	IP a la cual se deberá conectar con la Memoria
PUERTO_MEMORIA	Numérico	Puerto al cual se deberá conectar con la Memoria
IP_FILESYSTEM	String	IP a la cual se deberá conectar con el FileSystem
PUERTO_FILESYSTEM	Numérico	Puerto al cual se deberá conectar con la FileSystem
IP_CPU	String	IP a la cual se deberá conectar con la CPU
PUERTO_CPU	Numérico	Puerto al cual se deberá conectar con la CPU
PUERTO_ESCUCHA	Numérico	Puerto en el cual se escucharán las conexiones de los módulos Consola
ALGORITMO_PLANIFICACION	String	Define el algoritmo de planificación de corto plazo. (FIFO / HRRN)
ESTIMACION_INICIAL	Numérico	Estimación inicial para el cálculo de la primer ráfaga de CPU en milisegundos
HRRN_ALFA	Numérico	Alfa para el cálculo de las rafagas de CPU
GRADO_MAX_MULTIPROGRAMACION	Numérico	Grado máximo de multiprogramación del módulo
RECURSOS	Lista	Lista ordenada de los nombres de los recursos compartidos del sistema
INSTANCIAS_RECURSOS	Lista	Lista ordenada de la cantidad de unidades por recurso

Ejemplo de Archivo de Configuración

```
IP_MEMORIA=127.0.0.1
PUERTO_MEMORIA=8002
IP_FILESYSTEM=127.0.0.1
PUERTO_FILESYSTEM=8003
IP_CPU=127.0.0.1
PUERTO_CPU=8001
PUERTO_ESCUCHA=8000
ALGORITMO_PLANIFICACION=HRRN
ESTIMACION_INICIAL=10000
HRRN_ALFA=0.5
GRADO_MAX_MULTIPROGRAMACION=4
RECURSOS=[DISCO, RECURSO_1]
INSTANCIAS_RECURSOS=[1, 2]
```

Módulo: CPU

El módulo **CPU** es el encargado de interpretar y ejecutar las instrucciones de los Contextos de Ejecución recibidos por parte del **Kernel**. Para ello, simulará un ciclo de instrucción simplificado (Fetch[6], Decode y Execute).

A la hora de ejecutar instrucciones que lo requieran, sea para interactuar directamente con la Memoria o relacionadas al FileSystem que interactúen con Memoria, tendrá que traducir las *direcciones lógicas* (propias del proceso) a *direcciones físicas* (propias de la memoria).

Durante el transcurso de la ejecución de un proceso, se irá actualizando su **Contexto de Ejecución,** que luego será devuelto al **Kernel** bajo los siguientes escenarios: finalización del mismo (instrucción **EXIT o ante un error**), necesitar ser bloqueado (instrucción **I/O** o **WAIT**), peticiones al Kernel (F_* y *_SEGMENT), o pedir ser desalojado (**YIELD**).

Lineamiento e Implementación

Al iniciarse el módulo, se conectará con la **Memoria** y realizará un *handshake* inicial para verificar que se está conectando correctamente.

Quedará a la espera de las conexiones por parte del **Kernel**. Una vez conectado, el **Kernel** le enviará el **Contexto de Ejecución** para ejecutar. Habiéndose recibido, se procederá a realizar el ciclo de instrucción tomando como punto de partida la instrucción que indique el *Program Counter* recibido.

La CPU contará con una serie de **registros de propósito general**, los cuales podrán almacenar caracteres alfanuméricos[7] (cadenas de caracteres) las cuales no terminarán en `'\0'` y serán los siguientes:

- Registros de 4 bytes: AX, BX, CX, DX.
- Registros de 8 bytes: EAX, EBX, ECX, EDX
- Registros de 16 bytes: RAX, RBX, RCX, RDX

Estos valores deberán devolverse al Kernel como parte del **Contexto de Ejecución** del proceso.

Los valores a almacenar en los registros siempre tendrán la misma longitud que el registro, es decir que si el registro es de 16 bytes siempre se escribirán 16 caracteres. Por ejemplo:

```
SET RAX ESTO_ES_UN_EJ000
```

Lo mismo ocurre para los registros de 4 y 8 bytes:

```
SET EAX TEXTO111
SET AX UNO6
```

[6] A diferencia de la realidad donde el Fetch busca la instrucción en memoria, en esta simplificación, buscaremos las instrucciones dentro del PCB.

[7] En los registros reales de una CPU los strings se manejan con registros de tipo puntero, pero a fin de simplificar la implementación, vamos a guardar los strings en los registros.

Ciclo de Instrucción

Fetch

La primera etapa del ciclo consiste en buscar la próxima instrucción a ejecutar. En este trabajo práctico, las instrucciones estarán contenidas dentro del Contexto de Ejecución[8] a modo de lista. Teniendo esto en cuenta, utilizaremos el *Program Counter* (también llamado *Instruction Pointer*), que representa el número de instrucción a buscar, para buscar la próxima instrucción. Al finalizar el ciclo, este último deberá ser actualizado (sumarle 1).

Decode

Esta etapa consiste en interpretar qué instrucción es la que se va a ejecutar y si la misma requiere de una traducción de dirección lógica a dirección física.

En el caso de la instrucción SET deberá esperar un tiempo definido por archivo de configuración (RETARDO_INSTRUCCION), a modo de simular el tiempo que transcurre en la CPU.

El resto de las instrucciones, no tendrán retardo de instrucción, ya que el mismo se encuentra en los módulos con los que se comunican.

Execute

En este paso se deberá ejecutar lo correspondiente a cada instrucción:

- **SET** (Registro, Valor)**:** Asigna al registro el valor pasado como parámetro.
- **MOV_IN** (Registro, Dirección Lógica): Lee el valor de memoria correspondiente a la Dirección Lógica y lo almacena en el Registro.
- **MOV_OUT** (Dirección Lógica, Registro)**:** Lee el valor del Registro y lo escribe en la dirección física de memoria obtenida a partir de la Dirección Lógica.
- **I/O** (Tiempo): Esta instrucción representa una syscall de I/O bloqueante. Se deberá devolver el **Contexto de Ejecución** actualizado al **Kernel** junto a la cantidad de unidades de tiempo que va a bloquearse el proceso.
- **F_OPEN** (Nombre Archivo): Esta instrucción solicita al kernel que abra o cree el archivo pasado por parámetro.
- **F_CLOSE** (Nombre Archivo): Esta instrucción solicita al kernel que cierre el archivo pasado por parámetro.
- **F_SEEK** (Nombre Archivo, Posición): Esta instrucción solicita al kernel actualizar el puntero del archivo a la posición pasada por parámetro.
- **F_READ** (Nombre Archivo, Dirección Lógica, Cantidad de Bytes): Esta instrucción solicita al Kernel que se lea del archivo indicado, la cantidad de bytes pasada por parámetro y se escriba en la dirección física de Memoria la información leída.

[8] Esto no es lo que ocurriría en Sistemas Operativos actuales ni lo que se ve en clase, pero lo tomamos como una simplificación para la realización del trabajo en los tiempos del cuatrimestre y facilitar un desarrollo de tipo iterativo incremental.

- **F_WRITE** (Nombre Archivo, Dirección Lógica, Cantidad de bytes): Esta instrucción solicita al Kernel que se escriba en el archivo indicado, la cantidad de bytes pasada por parámetro cuya información es obtenida a partir de la dirección física de Memoria.
- **F_TRUNCATE** (Nombre Archivo, Tamaño): Esta instrucción solicita al Kernel que se modifique el tamaño del archivo al indicado por parámetro.
- **WAIT** (Recurso): Esta instrucción solicita al Kernel que se asigne una instancia del recurso indicado por parámetro.
- **SIGNAL** (Recurso): Esta instrucción solicita al Kernel que se libere una instancia del recurso indicado por parámetro.
- **CREATE_SEGMENT** (Id del Segmento, Tamaño): Esta instrucción solicita al kernel la creación del segmento con el Id y tamaño indicado por parámetro.
- **DELETE_SEGMENT** (Id del Segmento): Esta instrucción solicita al kernel que se elimine el segmento cuyo Id se pasa por parámetro.
- **YIELD**: Esta instrucción desaloja voluntariamente el proceso de la CPU. Se deberá devolver el **Contexto de Ejecución** actualizado al Kernel.
- **EXIT**: Esta instrucción representa la syscall de finalización del proceso. Se deberá devolver el **Contexto de Ejecución** actualizado al Kernel para su finalización.

Para las siguientes instrucciones se deberá devolver al módulo **Kernel** el **Contexto de Ejecución** actualizado junto al motivo del desalojo y los parámetros que necesite cada instrucción: I/O, F_OPEN, F_CLOSE, F_SEEK, F_READ, F_WRITE, F_TRUNCATE, WAIT, SIGNAL, CREATE_SEGMENT, DELETE_SEGMENT, YIELD y EXIT.

MMU

A la hora de **traducir direcciones lógicas a físicas**, la CPU debe tomar en cuenta que el esquema de memoria del sistema es de **Segmentación**. Por lo tanto, las direcciones lógicas se compondrán de la siguiente manera:

[N° Segmento | Desplazamiento]

Estas traducciones, en los ejercicios prácticos que se ven en clases y se toman en los parciales, normalmente se hacen a partir de su representación en binario. Como en la realización del trabajo práctico es más cómodo utilizar operaciones aritméticas, la forma de realizar los cálculos sería más parecida a la siguiente:

num_segmento = *floor*(dir_logica / **tam_max_segmento**)

desplazamiento_segmento = dir_logica *%* **tam_max_segmento**

En caso de que el desplazamiento dentro del segmento (**desplazamiento_segmento**) sumado al tamaño a leer / escribir, sea mayor al tamaño del segmento, deberá devolverse el Contexto de Ejecución al **Kernel** para que este lo finalice con motivo de **Error: Segmentation Fault (SEG_FAULT)**.

Logs mínimos y obligatorios

Instrucción Ejecutada: "`PID: <PID> - Ejecutando: <INSTRUCCION> - <PARAMETROS>`"

Acceso Memoria: "`PID: <PID> - Acción: <LEER / ESCRIBIR> - Segmento: <NUMERO SEGMENTO> - Dirección Física: <DIRECCION FISICA> - Valor: <VALOR LEIDO / ESCRITO>`"

Error Segmentation Fault: "`PID: <PID> - Error SEG_FAULT- Segmento: <NUMERO SEGMENTO> - Offset: <OFFSET> - Tamaño: <TAMAÑO>`"

Archivo de configuración

Campo	Tipo	Descripción
RETARDO_INSTRUCCION	Numérico	Tiempo en milisegundos que se deberá esperar al ejecutar las instrucciones SET.
IP_MEMORIA	String	IP a la cual se deberá conectar con la Memoria
PUERTO_MEMORIA	Numérico	Puerto al cual se deberá conectar con la Memoria
PUERTO_ESCUCHA	Numérico	Puerto en el cual se escuchará la conexión del Kernel
TAM_MAX_SEGMENTO	Numérico	Tamaño máximo del segmento en bytes

Ejemplo de Archivo de Configuración

```
RETARDO_INSTRUCCION=1000
IP_MEMORIA=127.0.0.1
PUERTO_MEMORIA=8002
PUERTO_ESCUCHA=8001
TAM_MAX_SEGMENTO=128
```

Módulo: Memoria

Este módulo será el encargado de responder los pedidos realizados por la CPU para leer y/o escribir en los segmentos de Datos del proceso en ejecución, referenciados por diversas tablas de segmentos.

Lineamiento e Implementación

Al iniciar el módulo esperará las conexiones de CPU, File System y Kernel y una vez establecidas todas las conexiones se procederá a crear las estructuras administrativas necesarias. Además, se creará inicialmente un segmento de tamaño definido por archivo de configuración, que será compartido entre todos los procesos, por lo cual, el mismo será asignado como **segmento 0** de todos los procesos que se creen durante la ejecución. Este segmento compartido o segmento 0, siempre será de un tamaño menor o igual al tamaño máximo de segmento definido en la CPU.

Estructuras

La memoria estará compuesta principalmente por 2 estructuras las cuales son:

- Un espacio contiguo de memoria (representado por un **void***). Este representará el espacio de usuario de la misma, donde los procesos podrán leer y/o escribir.
- Estructuras auxiliares para la implementación de segmentación, estas pueden extenderse según lo necesite el equipo, por ejemplo:
    - Tablas de segmentos (una por proceso)
    - Lista de huecos libres

Es importante aclarar que **cualquier implementación que no tenga todo el espacio de memoria dedicado a representar el espacio de usuario de manera contigua será motivo de desaprobación directa**, para esto se puede llegar a controlar la implementación a la hora de iniciar la evaluación.

Esquema de memoria

La asignación de memoria de este trabajo práctico utilizará un esquema de **Segmentación**, donde los segmentos se podrán ubicar en los espacios libres utilizando alguno de los siguientes algoritmos:

- First Fit
- Best Fit
- Worst Fit

Comunicación con Kernel, CPU y File System

Inicialización del proceso

El módulo deberá crear las estructuras administrativas necesarias y enviar como *respuesta* la tabla de segmentos inicial del proceso. Esta tabla tendrá un tamaño fijo definido por archivo de configuración.

Finalización de proceso

Al ser finalizado un proceso, se debe liberar su espacio de memoria.

Acceso a espacio de usuario

Tanto CPU como File System pueden, dada una dirección física, solicitar accesos al espacio de usuario de Memoria. El módulo deberá realizar lo siguiente:

- Ante un pedido de lectura, devolver el valor que se encuentra en la posición pedida.
- Ante un pedido de escritura, escribir lo indicado en la posición pedida y responder un mensaje de 'OK'.

Para simular la realidad y la velocidad de los accesos a Memoria, cada acceso al espacio de usuario tendrá un tiempo de espera en milisegundos definido por archivo de configuración.

Creación de Segmento

El módulo deberá verificar en sus estructuras administrativas la disponibilidad del espacio para crear el segmento. En caso de contar con el espacio necesario de manera contigua creará el segmento y devolverá su dirección base al Kernel.

En caso de contar con el espacio necesario pero el mismo se encuentre en diferentes huecos, se deberá informar al Kernel que debe solicitar una compactación previa a crear el segmento.

En caso de no contar con el espacio necesario, se deberá informar al Kernel de la falta de espacio libre.

Eliminación de Segmento

El módulo marcará el segmento como libre y en caso de que tenga huecos libres aledaños, los deberá consolidar actualizando sus estructuras administrativas.

Compactación de Segmentos

La memoría deberá mover los segmentos a fin de eliminar los huecos libres entre los mismos dejando un único hueco libre de todo el espacio disponible.

Como tarea principal de esta operación, el módulo Memoria deberá informar al kernel las tablas de segmentos actualizadas.

Logs mínimos y obligatorios

Creación de Proceso: “Creación de Proceso PID: <PID>”

Eliminación de Proceso: “Eliminación de Proceso PID: <PID>”

Creación de Segmento: “PID: <PID> - Crear Segmento: <ID SEGMENTO> - Base: <DIRECCIÓN BASE> - TAMAÑO: <TAMAÑO>”

Eliminación de Segmento: “PID: <PID> - Eliminar Segmento: <ID SEGMENTO> - Base: <DIRECCIÓN BASE> - TAMAÑO: <TAMAÑO>”

Inicio Compactación: “Solicitud de Compactación”

Resultado Compactación: Por cada segmento de cada proceso se deberá imprimir una línea con el siguiente formato:

“PID: <PID> - Segmento: <ID SEGMENTO> - Base: <BASE> - Tamaño <TAMAÑO>”

Acceso a espacio de usuario: “PID: <PID> - Acción: <LEER / ESCRIBIR> - Dirección física: <DIRECCIÓN_FÍSICA> - Tamaño: <TAMAÑO> - Origen: <CPU / FS>”

Archivo de configuración

Campo	Tipo	Descripción
PUERTO_ESCUCHA	Numérico	Puerto en el cual se escuchará la conexión de módulo.
TAM_MEMORIA	Numérico	Tamaño expresado en bytes del espacio de usuario de la memoria.
TAM_SEGMENTO_0	Numérico	Tamaño expresado en bytes del segmento compartido entre todos los procesos.
CANT_SEGMENTOS	Numérico	Es la cantidad de entradas de la tabla segmentos de cada proceso.
RETARDO_MEMORIA	Numérico	Tiempo en milisegundos que se deberá esperar para dar una respuesta al CPU / File System ante un pedido de acceso al espacio de usuario.
RETARDO_COMPACTACION	Numérico	Tiempo en milisegundos que se deberá esperar para dar una respuesta al Kernel ante una petición de compactación
ALGORITMO_ASIGNACION	String	Algoritmo de asignación de segmentos (FIRST/BEST/WORST).

Ejemplo de Archivo de Configuración

```
PUERTO_ESCUCHA=8002
TAM_MEMORIA=4096
TAM_SEGMENTO_0=128
CANT_SEGMENTOS=16
RETARDO_MEMORIA=1000
RETARDO_COMPACTACION=60000
ALGORITMO_ASIGNACION=BEST
```

Módulo: File System

Este módulo **File System** será el encargado de persistir la información por medio de una serie de archivos. Estos archivos, se encontrarán configurados dentro del archivo de configuración.

El **File System** será un cliente más del módulo **Memoria** y será servidor del módulo **Kernel**.

Lineamiento e Implementación

Al iniciarse el módulo, se conectará con la **Memoria** y realizará un *handshake* inicial para verificar que se está conectando correctamente.

Luego de establecer la comunicación con la **Memoria**, el módulo **File System** deberá levantar el bitmap de bloques y recorrer el directorio de FCBs para crear las estructuras administrativas que le permita administrar los archivos.

Quedará a la espera de la conexión por parte del **Kernel**. Una vez conectado, atenderá una por una las solicitudes que el **Kernel** le envíe para operar con sus archivos. Estas operaciones serán:

- Abrir Archivo
- Crear Archivo
- Truncar Archivo
- Leer Archivo
- Escribir Archivo

Nota: Para operar con el archivo de bloques recomendamos optar por utilizar alguna de las siguientes funciones:

- Combinar `fseek()`, `fread()` y `fwrite()`
- Usar `mmap()` con álgebra de punteros.

Ambas opciones de implementación serán válidas para la construcción del TP.

Comunicación con Kernel y Memoria

Abrir Archivo

Esta operación consistirá en verificar que exista el FCB correspondiente al archivo y en caso de que exista deberá devolver un OK, caso contrario, deberá informar que el archivo no existe.

Crear Archivo

Para esta operación se deberá crear un archivo FCB correspondiente al nuevo archivo, con tamaño 0 y sin bloques asociados.

Siempre será posible crear un archivo y por lo tanto esta operación deberá devolver OK.

Truncar Archivo

Al momento de truncar un archivo, pueden ocurrir 2 situaciones:

- Ampliar el tamaño del archivo: Al momento de ampliar el tamaño del archivo deberá actualizar el tamaño del archivo en el FCB y se le deberán asignar tantos bloques como sea necesario para poder direccionar el nuevo tamaño.
- Reducir el tamaño del archivo: Se deberá asignar el nuevo tamaño del archivo en el FCB y se deberán marcar como libres todos los bloques que ya no sean necesarios para direccionar el tamaño del archivo (descartando desde el final del archivo hacia el principio).

Leer Archivo

Esta operación deberá leer la información correspondiente de los bloques a partir del puntero y el tamaño recibidos. Esta información se deberá enviar a la **Memoria** para ser escrita a partir de la dirección física recibida por parámetro y esperar su finalización para poder confirmar el éxito de la operación al Kernel.

Escribir Archivo

Se deberá solicitar a la **Memoria** la información que se encuentra a partir de la dirección física y escribirlo en los bloques correspondientes del archivo a partir del puntero recibido.

El tamaño de la información a leer de la memoria y a escribir en los bloques también deberá recibirse por parámetro desde el **Kernel**.

Persistencia

Todas las operaciones que se realicen sobre los FCBs, Bitmap y Bloques deberán mantenerse actualizadas en disco a medida que ocurren.

En caso de utilizar la función `mmap()`, se recomienda investigar el uso de la función `msync()` para tal fin.

Logs mínimos y obligatorios

Crear Archivo: “`Crear Archivo: <NOMBRE_ARCHIVO>`”

Apertura de Archivo: “`Abrir Archivo: <NOMBRE_ARCHIVO>`”

Truncate de Archivo: “`Truncar Archivo: <NOMBRE_ARCHIVO> - Tamaño: <TAMAÑO>`”

Acceso a Bitmap: “`Acceso a Bitmap - Bloque: <NUMERO BLOQUE> - Estado: <ESTADO>`”
Nota: El estado es 0 o 1 donde 0 es libre y 1 es ocupado.

Lectura de Archivo: “`Leer Archivo: <NOMBRE_ARCHIVO> - Puntero: <PUNTERO ARCHIVO> - Memoria: <DIRECCION MEMORIA> - Tamaño: <TAMAÑO>`”

Escritura de Archivo: “`Escribir Archivo: <NOMBRE_ARCHIVO> - Puntero: <PUNTERO ARCHIVO> - Memoria: <DIRECCION MEMORIA> - Tamaño: <TAMAÑO>`”

Acceso a Bloque: “`Acceso Bloque - Archivo: <NOMBRE_ARCHIVO> - Bloque Archivo: <NUMERO BLOQUE ARCHIVO> - Bloque File System <NUMERO BLOQUE FS>`”

Archivo de configuración

Campo	Tipo	Descripción
IP_MEMORIA	String	IP a la cual se deberá conectar con la Memoria
PUERTO_MEMORIA	Numérico	Puerto al cual se deberá conectar con la Memoria
PUERTO_ESCUCHA	Numérico	Puerto en el cual se escuchará la conexión del Kernel
PATH_SUPERBLOQUE	String	Path al archivo de superbloque
PATH_BITMAP	String	Path al archivo de bitmap
PATH_BLOQUES	String	Path al archivo de bloques
PATH_FCB	String	Path al directorio de los FCB
RETARDO_ACCESO_BLOQUE	Numérico	Tiempo en milisegundos que se deberá esperar por cada acceso a bloques (de datos o punteros)

Ejemplo de Archivo de Configuración

```
IP_MEMORIA=127.0.0.1
PUERTO_MEMORIA=8002
PUERTO_ESCUCHA=8003
PATH_SUPERBLOQUE=/home/utnso/fs/superbloque.dat
PATH_BITMAP=/home/utnso/fs/bitmap.dat
PATH_BLOQUES=/home/utnso/fs/bloques.dat
PATH_FCB=/home/utnso/fs/fcb
RETARDO_ACCESO_BLOQUE=15000
```

Descripción de las entregas

Debido al orden en que se enseñan los temas de la materia en clase, los checkpoints están diseñados para que se pueda realizar el trabajo práctico de manera iterativa incremental tomando en cuenta los conceptos aprendidos hasta el momento de cada checkpoint.

Checkpoint 1: Conexión Inicial

Fecha: 15/04/2023

Objetivos:

- Familiarizarse con Linux y su consola, el entorno de desarrollo y el repositorio.
- Aprender a utilizar las Commons, principalmente las funciones para listas, archivos de configuración y logs.
- Definir el Protocolo de Comunicación.
- Todos los módulos están creados y son capaces de establecer conexiones entre sí.

Lectura recomendada:

- Tutoriales de "Cómo arrancar" de la materia y TP0: https://docs.utnso.com.ar/primeros-pasos
- Git para el Trabajo Práctico - https://docs.utnso.com.ar/guias/consola/git
- Guía de Punteros en C - https://docs.utnso.com.ar/guias/programacion/punteros
- Guía de Sockets - https://docs.utnso.com.ar/guias/linux/sockets
- SO Commons Library - https://github.com/sisoputnfrba/so-commons-library

Checkpoint 2: Avance del Grupo

Fecha: 13/05/2023

Objetivos:

- **Módulo Consola:**
  - Levanta el archivo de configuración
  - Levanta e interpreta el archivo de pseudocódigo
  - Se conecta al Kernel y envía las instrucciones
- **Módulo Kernel:**
  - Levanta el archivo de configuración
  - Se conecta a CPU, Memoria y File System
  - Espera conexiones de las consolas
  - Recibe de las consolas las instrucciones y arma el PCB
  - Planificación de procesos con FIFO
- **Módulo CPU:**
  - Levanta el archivo de configuración
  - Genera las estructuras de conexión con el proceso Kernel y Memoria.
  - Ejecuta las instrucciones SET, ADD, SUB y EXIT.
- **Módulo Memoria:**
  - Levanta el archivo de configuración
  - Espera las conexiones de CPU, Kernel y File System
- **Módulo File System:**
  - Levanta el archivo de configuración
  - Se conecta a Memoria y espera la conexión de Kernel

Lectura recomendada:

- Guía de Buenas Prácticas de C - https://docs.utnso.com.ar/guias/programacion/buenas-practicas
- Guía de Serialización - https://docs.utnso.com.ar/guias/linux/serializacion
- Charla de Threads y Sincronización - https://docs.utnso.com.ar/guias/linux/threads

Checkpoint 3: Obligatorio - Presencial

Fecha: 03/06/2023

Objetivos:

- **Realizar pruebas mínimas en un entorno distribuido.**
- **Módulo Consola (Completo):**
  - Finalizar la implementación del módulo
- **Módulo Kernel:**
  - Planificación de procesos con FIFO y HRRN
  - Maneja recursos compartidos
- **Módulo CPU:**
  - Interpreta todas las operaciones
  - Ejecuta correctamente I/O, WAIT, SIGNAL
- **Módulo Memoria:**
  - Espera las peticiones de los demás módulos y responde con mensajes genéricos.
- **Módulo File System:**
  - Espera las peticiones del Kernel y responde las mismas con mensajes genéricos.
  - Levanta los archivos de Superbloque, bitmap y archivo de bloques.

Lectura recomendada:

- Sistemas Operativos, Stallings, William 5ta Ed. - Parte IV: Planificación
- Sistemas Operativos, Silberschatz, Galvin 7ma Ed. - Capítulo 5: Planificación
- Sistemas Operativos, Stallings, William 5ta Ed. - Parte VII: Gestión de la memoria (Cap. 7)
- Guía de Debugging - https://docs.utnso.com.ar/guias/herramientas/debugger
- Guía de Despliegue de TP - https://docs.utnso.com.ar/guias/herramientas/deploy
- Guía de uso de Bash - https://docs.utnso.com.ar/guias/consola/bash

Checkpoint 4: Avance del Grupo

Fechas: 24/06/2023

Objetivos:

- **Realizar pruebas mínimas en un entorno distribuido.**
- **Módulo Consola (Completo)**
- **Módulo Kernel**
  - Recibe las funciones de la CPU
  - Solicita a la memoria la creación y/o finalización de procesos.
  - Solicita a memoria la creación y/o eliminación de segmentos.
  - Solicita al File System la ejecución de las operaciones de creación y truncado de archivos.
- **Módulo CPU (Completo)**
  - Ejecuta todas las instrucciones haciendo los llamados correspondientes a Kernel y Memoria
- **Módulo Memoria:**
  - Implementa las estructuras administrativas
  - Implementa el espacio de usuario
  - Responde correctamente a las peticiones de CPU y Kernel (Sin compactación)
- **Módulo File System:**
  - Implementa las estructuras administrativas para manejar los FCB
  - Permite la creación de archivos
  - Permite el truncado de archivos.

Lectura recomendada:

- Sistemas Operativos, Silberschatz, Galvin 7ma Ed. - Capítulo 8: Memoria principal
- Sistemas Operativos, Stallings, William 5ta Ed. - Gestión de Ficheros (Cap.128)
- Sistemas Operativos, Silberschatz, Galvin 7ma Ed. - Capítulo 10: Interfaz del sistema de archivos
- Sistemas Operativos, Silberschatz, Galvin 7ma Ed. - Capítulo 11: Implementación de sistemas de archivos
- Tutorial de Valgrind - https://docs.utnso.com.ar/guias/herramientas/valgrind

Checkpoint 5: Entregas Finales

Fechas: 15/07/2023 - 29/07/2023 - 05/08/2023

Objetivos:

- Finalizar el desarrollo de todos los procesos.
- Probar de manera intensiva el TP en un entorno distribuido.
- Todos los componentes del TP ejecutan los requerimientos de forma integral.

Lectura recomendada:

- Guía de Despliegue de TP - https://docs.utnso.com.ar/guias/herramientas/deploy
- Guía de uso de Bash - https://docs.utnso.com.ar/guias/consola/bash

Anexo: Implementación de File System

Estructuras

El módulo File System de este trabajo práctico utilizará un sistema de archivos ideado a fines de simplificar la creación del TP.

Contará con una serie de estructuras las cuales nos ayudarán a persistir los datos:

- Superbloque
- Bitmap de bloques
- Archivo de Bloques
- FCB

Superbloque

Este archivo contendrá la información administrativa del File System, esta información consta de 2 valores que son:

- Block size: Indica el tamaño en bytes de cada bloque.
- Block count: Indica la cantidad de bloques que posee el File System.

El superbloque será compatible con el formato de los archivos de config de las commons, por lo que se recomienda la utilización de las funciones ya creadas para facilitar su implementación.

Ejemplo de Superbloque

```
BLOCK_SIZE=64
BLOCK_COUNT=65536
```

Bitmap de Bloques

Este archivo es un archivo que contendrá **un bit por cada bloque disponible en el filesystem**, cualquier otra implementación será motivo de desaprobación.

Se recomienda investigar el uso de `mmap()`, y las funciones de bitarray de las commons para facilitar la implementación del mismo.

Archivo de Bloques

Este archivo es un archivo que contendrá los bloques de datos de nuestro File System, el mismo se puede considerar como un array de bloques.

FCB

La estructura de *File Control Block* que utilizará el FS del Sistema tomará como referencia una versión simplificada de un EXT2, por lo tanto el mismo tendrá los siguientes datos:

- Nombre del archivo[9]: Este será el identificador del archivo, ya que no tendremos 2 archivos con el mismo nombre.
- Tamaño del archivo: Indica el tamaño del archivo expresado en *bytes*.
- Puntero directo: Apunta al primer bloque de datos del archivo.
- Puntero indirecto: Apunta a un bloque que contendrá los punteros a los siguientes bloques del archivo.

Nota: todos los punteros del FS se almacenarán como números enteros no signados de 4 bytes (uint32_t)

Ejemplo de FCB

```
NOMBRE_ARCHIVO=Notas1erParcialK9999
TAMANIO_ARCHIVO=256
PUNTERO_DIRECTO=12
PUNTERO_INDIRECTO=45
```

Los archivos de FCB serán compatibles con el formato de los archivos de config de las commons, por lo que se recomienda la utilización de las funciones ya creadas para facilitar su implementación.

[9] En un esquema real de FS tipo UNIX (EXT), el nombre del archivo no es parte del FCB ya que debería estar en las entradas de directorio, los FCBs reales suelen tener un id a modo de identificación.