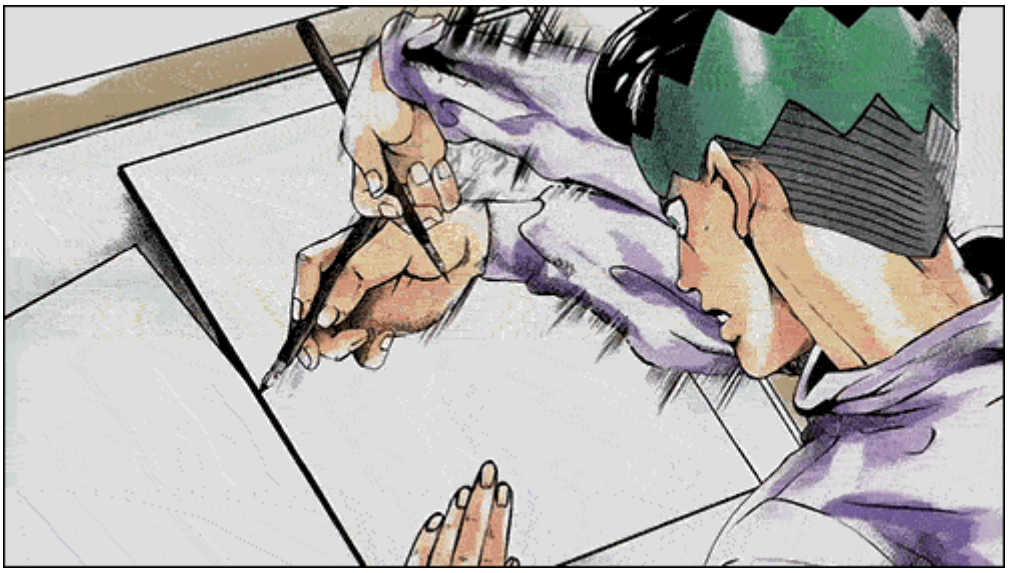
MaPPA

Manejo Planificado de Procesos y Archivos

Llegó la 4ta temporada



Sabemos que tardamos, pero siempre buscamos lo mejor

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

-2C2023 -
Versión 1.1

Índice

Historial de Cambios

v1.0 (25/08/2023) Release inicial de trabajo práctico

v1.1 (18/10/2023)

- *Agregadas aclaraciones sobre lectura/escritura de archivos en FS*
- *Ajustes en logs mínimos y obligatorios en Memoria*
- *Agregados logs mínimos y obligatorios en Kernel*
- *Aclaración en funcionamiento de “detener planificación” en Kernel*
- *Aclaración sobre los punteros en f_read y f_write en Kernel*

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: 02/09/2023
- Fecha de primera entrega: 02/12/2023
- Fecha de segunda entrega: 09/12/2023
- Fecha de tercera entrega: 16/12/2023
- 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.

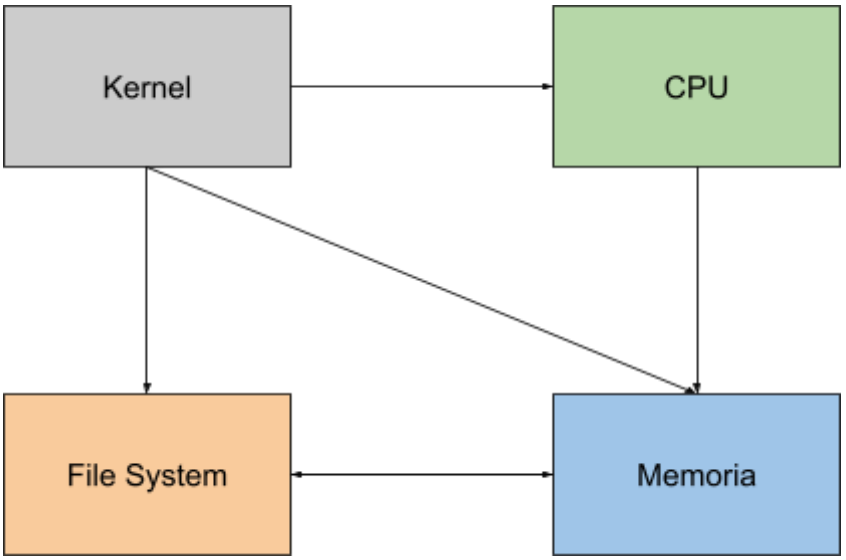
¿Qué es el trabajo práctico y cómo empezamos?

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



Distribución Recomendada

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

- Kernel: **40%**
- CPU: **5%**
- Memoria: **25%**
- FileSystem: **30%**

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 y/o el módulo CPU es insuficiente para poder entender y aprender los distintos conceptos de la materia. Dicho caso será un motivo de desaprobación directa.

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: Kernel

El módulo **Kernel,** en el contexto de nuestro trabajo práctico, será el encargado de gestionar la ejecución de los diferentes procesos que se generen por medio de su consola interactiva.

Lineamiento e Implementación

El módulo Kernel será el encargado de iniciar los procesos del sistema, para ello contará con una consola interactiva la cual permitirá las siguientes operaciones:

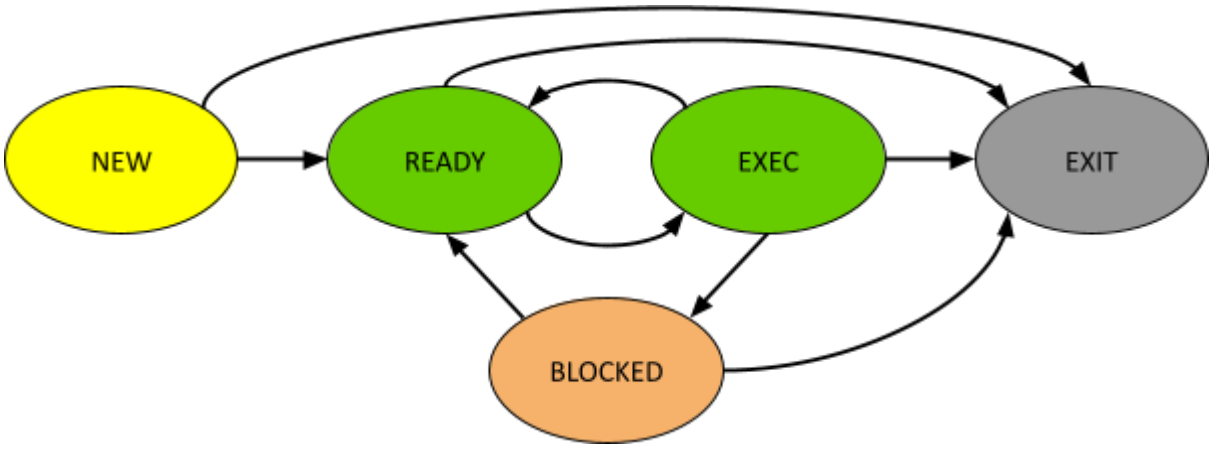
- Iniciar proceso
- Finalizar proceso
- Iniciar planificación
- Detener planificación
- Modificar grado multiprogramación
- Listar procesos por estado

Además de las operaciones de la consola interactiva, el Kernel será el encargado de gestionar las peticiones contra la Memoria y el File System, por lo que deberá implementarse siguiendo una estrategia multihilo que permita la concurrencia de varias solicitudes provenientes de diferentes orígenes.

Sumado a esto, el Kernel se encargará de planificar la ejecución de los procesos del sistema en el módulo CPU a través de dos conexiones con el mismo: una de *dispatch* y otra de *interrupt*.

Diagrama de estados

El kernel utilizará el diagrama de 5 estados para la planificación de los procesos. Dentro del estado BLOCK, se tendrán **múltiples colas**, las cuales pueden ser correspondientes a operaciones de File System, teniendo una por cada archivo abierto.



PCB

El PCB será la estructura base que utilizaremos dentro del Kernel para administrar los procesos. 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).
- **Program_counter**: Número de la próxima instrucción a ejecutar.
- **Prioridad**: Contendrá la prioridad con la que se solicitó la creación del proceso, representada por un número entero.
- **Registros de la CPU**: Estructura que contendrá los valores de los *registros de uso general* de la CPU.
- **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 solicitar la creación de un nuevo proceso al Kernel, deberá generarse la estructura PCB detallada anteriormente y asignar este PCB 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. 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.

Planificador de Corto Plazo

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

- **FIFO**
- **Round Robin**
- **Prioridades (con desalojo)**

Una vez seleccionado el siguiente proceso a ejecutar, se lo transicionará al estado **EXEC** y se enviará su *Contexto de Ejecución* al CPU a través del puerto de *dispatch,* quedando a la espera de recibir dicho contexto actualizado después de la ejecución, junto con un *motivo de desalojo* por el cual fue desplazado a manejar.

En caso que el algoritmo requiera desalojar al proceso en ejecución, se enviará una interrupción a través de la conexión de *interrupt* para forzar el desalojo del mismo.

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

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 lista anterior (ver ejemplo)

A la hora de recibir de la CPU un *Contexto de Ejecución* desalojado 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* desalojado por SIGNAL, el Kernel deberá verificar primero que exista el recurso solicitado, luego que el proceso cuente con una instancia del recurso (solicitada por WAIT) y por último 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 peticiona el SIGNAL.

Para las operaciones de WAIT y SIGNAL donde no se cumpla con las verificaciones (ya sea que el recurso no exista o no haya sido solicitado por ese proceso), se deberá enviar el proceso a EXIT.

Manejo de Memoria

El Kernel será el encargado de gestionar las peticiones de memoria para la creación y eliminación de procesos y sustituciones de páginas dentro del sistema. A continuación se detalla el comportamiento ante cada solicitud.

Creación de Procesos

Ante la solicitud de la consola de crear un nuevo proceso el Kernel deberá informarle a la memoria que debe crear un proceso con el nombre del archivo de pseudocódigo junto con el tamaño en bytes que ocupará el mismo.

Eliminación de Procesos

Ante la llegada de un proceso al estado de EXIT (ya sea por solicitud de la CPU o por ejecución desde la consola del Kernel) el Kernel deberá solicitar a la memoria que libere todas las estructuras asociadas al proceso y marque como libre todo el espacio que este ocupaba.

En caso de que el proceso se encuentre ejecutando en CPU, se deberá enviar una señal de interrupción a través de la conexión de *interrupt* con el mismo y aguardar a que éste retorne el *Contexto de Ejecución* antes de iniciar la liberación de recursos.

Page Fault

En caso de que el módulo CPU devuelva un PCB desalojado por Page Fault, se deberá crear un hilo específico para atender esta petición.

La resolución del Page Fault consta de los siguientes pasos:

1.- Mover al proceso al estado Bloqueado. Este estado bloqueado será independiente de todos los demás ya que solo afecta al proceso y no compromete recursos compartidos.

2.- Solicitar al módulo memoria que se cargue en memoria principal la página correspondiente, la misma será obtenida desde el mensaje recibido de la CPU.

3.- Esperar la respuesta del módulo memoria.

4.- Al recibir la respuesta del módulo memoria, desbloquear el proceso y colocarlo en la cola de ready.

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.

Las operaciones de Filesystem se dividirán en dos tipos de locks **obligatorios**: ***lock de lectura (R)*** y ***lock de escritura (W).*** Estos locks estarán identificados en la función **F_OPEN** y finalizarán con la función **F_CLOSE**. El lock de lectura se resolverá sin ser bloqueado siempre y cuando no exista activamente un lock de escritura. Al momento en el que se active el lock de escritura se deben empezar a encolar todos los locks de lectura o escritura que se requieran para dicho archivo. Dicho lock se bloqueará hasta que se resuelvan todos los pedidos que participen en el lock de dicho archivo que ya se encontraban activos.

De esta manera, se tendrá un único lock de lectura (compartido por todos los **F_OPEN** en modo lectura) y un lock exclusivo por cada pedido de apertura que llegue en modo escritura.

Además, cada proceso va a contar con su propio puntero, que le permitirá desplazarse por el contenido del archivo utilizando **F_SEEK** para luego efectuar una lectura (**F_READ**) o escritura (**F_WRITE**) sobre el mismo si el tipo de lockeo lo permite.

Para ejecutar estas funciones, el Kernel va a recibir el *Contexto de Ejecución* de la CPU y la función como *motivo*. El accionar de cada función va a ser el siguiente:

- **F_OPEN**: Esta función será la encargada de abrir el archivo pasado por parámetro (y crearlo en caso de que no exista). 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:

    - Modo Lectura: Al llegar este pedido se deberá validar si existe un lock de escritura activo
        - En caso de existir, se debe bloquear el proceso y esperar a que finalice dicha operación.
        - En caso de no existir pueden darse dos casos:
            - Que ya exista un lock de lectura en curso, de ser así se debe agregar al proceso como "participante" de dicho lock y el mismo finalizará cuando se ejecuten todos los **F_CLOSE** de los participantes.
            - Que no exista un lock de lectura. En este caso, se debe crear dicho lock como único participante.
    - Modo Escritura: Al llegar este pedido se creará un lock de escritura exclusivo para el proceso. Si ya existe un lock activo se bloqueará el proceso encolando dicho lock.
- **F_CLOSE**: Esta función será la encargada de cerrar el archivo pasado por parámetro. En caso que dicho archivo se encuentre abierto en Modo Lectura reducirá la cantidad de participantes en uno del lock (en caso que la cantidad llegue a 0 se cerrará dicho lock dando lugar a que otro se active en caso que haya encolados). Por otro lado, si dicho archivo se encuentra abierto en Modo Escritura se liberará el lock actual dando lugar al siguiente lock en la cola.
- **F_SEEK**: Actualiza el puntero del archivo en la tabla de archivos abiertos del proceso 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 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, en caso de que el proceso haya solicitado un lock de escritura, solicitará al módulo File System que escriba en el archivo desde la **dirección física** de memoria recibida por parámetro. 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. En caso de que el proceso haya solicitado un lock de lectura, se deberá cancelar la operación y enviar el proceso a EXIT con motivo de INVALID_WRITE.

Nota: Es importante aclarar que para las operaciones de F_READ y F_WRITE siempre se van a pasar tamaños y punteros válidos correspondientes al primer byte de un bloque. No se van a ingresar operaciones de File System sin haber abierto el archivo previamente con F_OPEN.

Funciones por consola

El Kernel dispondrá de una consola donde se permitirá la ejecución de 6 tipos de mensajes diferentes que se especifican a continuación:

- Iniciar proceso: Se encargará de ejecutar un nuevo proceso en base a un archivo dentro del file system de linux. Dicho mensaje se encargará de la creación del proceso (PCB) y dejará el

mismo en el estado NEW.
Nomenclatura: **INICIAR_PROCESO [PATH] [SIZE] [PRIORIDAD]**

- Finalizar proceso: Se encargará de finalizar un proceso que se encuentre dentro del sistema. Este mensaje se encargará de realizar las mismas operaciones como si el proceso llegara a EXIT por sus caminos habituales (deberá liberar recursos, archivos y memoria).
  Nomenclatura: **FINALIZAR_PROCESO [PID]**
- Detener planificación: Este mensaje se encargará de pausar la planificación de corto y largo plazo. El proceso que se encuentra en ejecución ***NO*** es desalojado, pero una vez que salga de EXEC se va a pausar el manejo de su motivo de desalojo. De la misma forma, los procesos bloqueados van a pausar su transición a la cola de Ready.
  Nomenclatura: **DETENER_PLANIFICACION**
- Iniciar planificación: Este mensaje se encargará de retomar (en caso que se encuentre pausada) la planificación de corto y largo plazo. En caso que la planificación no se encuentre pausada, se debe ignorar el mensaje.
  Nomenclatura: **INICIAR_PLANIFICACION**
- Modificar grado multiprogramación: Permitirá la actualización del grado de multiprogramación configurado inicialmente por archivo de configuración. En caso que dicho valor sea inferior al actual ***NO*** se debe desalojar ni finalizar los procesos.
  Nomenclatura: **MULTIPROGRAMACION [VALOR]**
- Listar procesos por estado: Se encargará de mostrar por consola el listado de los estados con los procesos que se encuentran dentro de cada uno de ellos.
  Nomenclatura: **PROCESO_ESTADO**

Se debe tener en cuenta que frente a un fallo en la escritura de un comando en consola el sistema debe permanecer estable sin reacción alguna.

Detección y resolución de Deadlocks

Frente a acciones donde los procesos lleguen al estado Block (por recursos o archivos) se debe realizar una detección automática de deadlock (proceso bloqueados entre sí). En caso que se detecte afirmativamente un deadlock se debe informar el mismo por consola.

En dicho mensaje se debe informar cuales son los procesos (PID) que se encuentran en deadlocks, cuales son los recursos (o archivos) tomados y cuales son los recursos (o archivos) requeridos (por los que fueron bloqueados).

La resolución de deadlocks se realizará de forma manual por la consola del Kernel utilizando el mensaje “Finalizar proceso” donde el proceso finalizado deberá liberar los recursos (o archivos) tomados. Una vez realizada esta acción se debe volver a realizar una nueva detección de deadlock.

Logs mínimos y obligatorios

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

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

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

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

Fin de Quantum: “PID: <PID> - Desalojado por fin de Quantum”

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

Page Fault: “Page Fault PID: <PID> - Pagina: <Página>”

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>”

Proceso de detección de deadlock: “ANÁLISIS DE DETECCIÓN DE DEADLOCK”

Detección de deadlock (por cada proceso dentro del deadlock): “Deadlock detectado: <PID> - Recursos en posesión <RECURSO_1>, <RECURSO_2>, <RECURSO_N> - Recurso requerido: <RECURSO_REQUERIDO>”

Pausa planificación: “PAUSA DE PLANIFICACIÓN”

Inicio de planificación: “INICIO DE PLANIFICACIÓN”

Listar procesos por estado: "Estado: <NOMBRE_ESTADO> - Procesos: <PID_1>, <PID_2>, <PID_N>" (por cada estado)

Cambio de Grado de Multiprogramación: “Grado Anterior: <GRADO_ANTERIOR> - Grado Actual: <GRADO_ACTUAL>”

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 FileSystem
PUERTO_FILESYSTEM	Numérico	Puerto al cual se deberá conectar con FileSystem
IP_CPU	String	IP a la cual se deberá conectar con la CPU
PUERTO_CPU_DISPATCH	Numérico	Puerto de *dispatch* al cual se deberá conectar con la CPU
PUERTO_CPU_INTERRUPT	Numérico	Puerto de *interrupt* al cual se deberá conectar con la CPU
ALGORITMO_PLANIFICACION	String	Define el algoritmo de planificación de corto plazo. (FIFO / RR / PRIORIDADES)
QUANTUM	Numérico	Tiempo en milisegundos del quantum para utilizar bajo el algoritmo RR
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
GRADO_MULTIPROGRAMACION_INI	Numérico	Grado de multiprogramación inicial del módulo

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_DISPATCH=8006
PUERTO_CPU_INTERRUPT=8007
```

```
ALGORITMO_PLANIFICACION=PRIORIDADES
QUANTUM=2000
RECURSOS=[RA,RB,RC]
INSTANCIAS_RECURSOS=[1,2,1]
GRADO_MULTIPROGRAMACION_INI=10
```

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, ejecutará un ciclo de instrucción simplificado que cuenta con los pasos: Fetch, Decode, Execute y Check Interrupt.

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). Para ello simulará la existencia de una MMU.

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**), peticiones al Kernel, necesitar ser bloqueado, o deber ser desalojado (**interrupción**).

Lineamiento e Implementación

Al iniciar, la CPU se deberá conectar al módulo **Memoria** y realizará un *handshake*[1] inicial para recibir de la memoria la información relevante que le permita traducir las *direcciones lógicas* en *direcciones físicas*, esta información debería incluir al menos el *tamaño de página*.

Quedará a la espera de las conexiones por parte del **Kernel**, tanto por el puerto *dispatch* como por el puerto *interrupt*. Una vez conectado, el **Kernel** le enviará el **Contexto de Ejecución** para ejecutar a través del puerto *dispatch*. 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 tendrán los siguientes nombres: AX, BX, CX, DX. Estos registros almacenarán valores enteros no signados de 4 bytes (*uint32_t*). Estos valores deberán devolverse al Kernel a través de la conexión de *dispatch* como parte del **Contexto de Ejecución** del proceso.

Ciclo de Instrucción

Fetch

La primera etapa del ciclo consiste en buscar la próxima instrucción a ejecutar. En este trabajo práctico cada instrucción deberá ser pedida al módulo Memoria utilizando el *Program Counter* (también llamado *Instruction Pointer*) que representa el número de instrucción a buscar relativo al proceso en ejecución. Al finalizar el ciclo, este último deberá ser actualizado (sumarle 1) si corresponde.

[1] No es estrictamente necesario que se realice un proceso de handshake pero es una buena práctica que se recomienda desde la cátedra.

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.

Ejemplos de instrucciones a interpretar

```
SET AX 1
SET BX 1
SUM AX BX
SUB AX BX
MOV_IN DX 0
MOV_OUT 0 CX
SLEEP 10
JNZ AX 4
WAIT RECURSO_1
SIGNAL RECURSO_1
F_OPEN ARCHIVO W
F_TRUNCATE ARCHIVO 64
F_SEEK ARCHIVO 8
F_WRITE ARCHIVO 0
F_READ ARCHIVO 0
F_CLOSE ARCHIVO
EXIT
```

Las instrucciones detalladas previamente son a modo de ejemplo, su ejecución no necesariamente sigue alguna lógica ni funcionamiento correcto. Ninguna instrucción contendrá errores sintácticos ni semánticos.

Execute

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

- **SET** (Registro, Valor)**:** Asigna al registro el valor pasado como parámetro.
- **SUM** (Registro Destino, Registro Origen): Suma ambos registros y deja el resultado en el Registro Destino.
- **SUB** (Registro Destino, Registro Origen): Resta al Registro Destino el Registro Origen y deja el resultado en el Registro Destino.
- **JNZ** (Registro, Instrucción): Si el valor del registro es distinto de cero, actualiza el *program counter* al número de instrucción pasada por parámetro.
- **SLEEP** (Tiempo): Esta instrucción representa una syscall bloqueante. Se deberá devolver el **Contexto de Ejecución** actualizado al **Kernel** junto a la cantidad de segundos que va a bloquearse el proceso.
- **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.
- **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.
- **F_OPEN** (Nombre Archivo, Modo Apertura): Esta instrucción solicita al kernel que abra el archivo pasado por parámetro con el modo de apertura indicado.
- **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): Esta instrucción solicita al Kernel que se lea del archivo indicado y se escriba en la dirección física de Memoria la información leída.
- **F_WRITE** (Nombre Archivo, Dirección Lógica): Esta instrucción solicita al Kernel que se escriba en el archivo indicado la información que 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.
- **EXIT**: Esta instrucción representa la syscall de finalización del proceso. Se deberá devolver el **Contexto de Ejecución** actualizado al Kernel para su finalización.

Cabe aclarar que **todos** los valores a leer/escribir en memoria serán numéricos enteros no signados de **4 bytes**, considerar el uso de *uint32_t*.

Check Interrupt

En este momento, se deberá chequear si el **Kernel** nos envió una *interrupción* al PID que se está ejecutando, en caso afirmativo, se devuelve el **Contexto de Ejecución** actualizado al Kernel con *motivo* de la interrupción. Caso contrario, se descarta la interrupción.

Cabe aclarar que en todos los casos el **Contexto de Ejecución** debe ser devuelto a través de la conexión de *dispatch*, quedando la conexión de *interrupt* dedicada solamente a recibir mensajes de interrupción.

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 **Paginación**. Por lo tanto, las direcciones lógicas se compondrán de la siguiente manera:

[número_pagina | desplazamiento]

Estas traducciones, en los ejercicios prácticos que se ven en clases y se toman en los parciales, normalmente se hacen en binario. Como en el lenguaje C los números enteros se operan independientemente de su base numérica, la operatoria puede desarrollarse de la siguiente forma:

número_página = *floor*(dirección_lógica / tamaño_página)

desplazamiento = dirección_lógica - **número_página** * tamaño_página

Como las tablas de páginas están presentes en el módulo Memoria, para conseguir el número de **marco** correspondiente a la página se deberá solicitar el mismo a dicho módulo para traducción.

Page Fault

Durante la traducción de dirección lógica a física, al momento de solicitar el marco asociado a una página, el módulo **Memoria** puede devolvernos 2 resultados posibles:

- Número de marco: En este caso finalizamos la traducción y continuamos con la ejecución.
- Page Fault: En este caso deberemos devolver el **contexto de ejecución** al Kernel *sin actualizar el valor del program counter*. Deberemos indicarle al Kernel qué número de página fue el que generó el page fault para que éste resuelva el mismo.

Logs mínimos y obligatorios

Fetch Instrucción: `"PID: <PID> - FETCH - Program Counter: <PROGRAM_COUNTER>".`

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

Obtener Marco: `"PID: <PID> - OBTENER MARCO - Página: <NUMERO_PAGINA> - Marco: <NUMERO_MARCO>".`

Page Fault: `"Page Fault PID: <PID> - Página: <Página>".`

Lectura/Escritura Memoria: `"PID: <PID> - Acción: <LEER / ESCRIBIR> - Dirección Física: <DIRECCION_FISICA> - Valor: <VALOR LEIDO / ESCRITO>".`

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_DISPATCH	Numérico	Puerto en el cual se escuchará la conexión del Kernel para mensajes de dispatch
PUERTO_ESCUCHA_INTERRUPT	Numérico	Puerto en el cual se escuchará la conexión del Kernel para mensajes de interrupciones

Ejemplo de Archivo de Configuración

```
IP_MEMORIA=127.0.0.1
```

```
PUERTO_MEMORIA=8002
PUERTO_ESCUCHA_DISPATCH=8006
PUERTO_ESCUCHA_INTERRUPT=8007
```

Módulo: Memoria

Lineamiento e Implementación

La memoria será un proceso que estará dividido en 2 partes internamente, la primer parte y la primera que les recomendamos iniciar es la Memoria de Instrucciones, esta parte será la encargada de brindarle las instrucciones a ejecutar a la CPU y la segunda parte será el Espacio de Usuario, esta parte se deberá implementar utilizando paginación bajo demanda y que será el encargado de responder los pedidos realizados por la CPU para leer y/o escribir los datos del proceso en ejecución.

La memoria se conecta con el módulo File System para poder administrar el espacio de SWAP.

Memoria de Instrucciones

Esta parte de la memoria será la encargada de obtener de los archivos de pseudo código las instrucciones y de devolverlas a pedido a la CPU.

Al momento de recibir la creación de un proceso, la memoria de instrucciones deberá leer el archivo de pseudocódigo indicado y generar las estructuras que el grupo considere necesarias para poder devolver las instrucciones de a 1 a la CPU según ésta se las solicite por medio del Program Counter.

Ante cada petición se deberá esperar un tiempo determinado a modo de retardo en la obtención de la instrucción, y este tiempo, estará indicado en el archivo de configuración.

Esquema de memoria

El módulo memoria de este trabajo práctico implementará un esquema de **paginación bajo demanda** con asignación de marcos variable y reemplazo global. El algoritmo de reemplazo será definido por archivo de configuración, pudiendo ser **FIFO** o **LRU**.

Marco	P	M	Pos en SWAP
5	1	1	2048
6	1	0	3076
5	0	0	4096
…			

Las tablas de páginas forman parte del espacio de memoria Kernel del sistema y, a diferencia de la realidad, **no** deben guardarse en el espacio de memoria reservado para los procesos, sino que podrán ser estructuras auxiliares.

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.
- Las Tablas de páginas.

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.

El tamaño de la memoria **siempre** será un múltiplo del tamaño de página.

Comunicación con Kernel, CPU y File System

Creación de proceso

El módulo deberá crear las estructuras administrativas necesarias y notificar al módulo FS para que nos asigne un bloque de SWAP para cada página del proceso.

Finalización de proceso

Al ser finalizado un proceso, se debe liberar su espacio de memoria y notificar al módulo FS que marque como disponibles las páginas correspondientes al proceso en la partición de SWAP.

Acceso a tabla de páginas

El módulo deberá responder el número de marco correspondiente a la página consultada. Para este caso, si la página correspondiente no se encuentra en memoria se deberá responder con un **Page Fault**.

Acceso a espacio de usuario

El módulo deberá realizar lo siguiente:

- Ante un pedido de lectura, devolver el valor que se encuentra a partir de la dirección física pedida.
- Ante un pedido de escritura, escribir lo indicado a partir de la dirección física pedida. En caso satisfactorio se responderá un mensaje de ‘OK’.

Cada petición tendrá un tiempo de espera en milisegundos definido por archivo de configuración.

Page Fault

El módulo deberá solicitar al módulo File System la página correspondiente y escribirla en la memoria principal.

En caso de que la memoria principal se encuentre llena, se deberá seleccionar una página víctima utilizando el algoritmo de reemplazo. Si la víctima se encuentra modificada, se deberá previamente escribir en SWAP.

Logs mínimos y obligatorios

Creación / destrucción de Tabla de Páginas: "`PID: <PID> - Tamaño: <CANTIDAD_PAGINAS>`"

Acceso a Tabla de Páginas: "`PID: <PID> - Pagina: <PAGINA> - Marco: <MARCO>`"

Acceso a espacio de usuario: "`PID: <PID> - Accion: <LEER / ESCRIBIR> - Direccion fisica: <DIRECCION_FISICA>`"

Reemplazo Página: "`REEMPLAZO - Marco: <NRO_MARCO> - Page Out: <PID>-<NRO_PAGINA> - Page In: <PID>-<NRO_PAGINA>`"

Lectura de Página de SWAP: "`SWAP IN - PID: <PID> - Marco: <MARCO> - Page In: <PID>-<NRO_PAGINA>`"

Escritura de Página en SWAP: "`SWAP OUT - PID: <PID> - Marco: <MARCO> - Page Out: <PID>-<NRO_PAGINA>`"

Archivo de configuración

Campo	Tipo	Descripción
PUERTO_ESCUCHA	Numérico	Puerto en el cual se escuchará la conexión de módulo.
IP_FILESYSTEM	String	IP a la cual se deberá conectar con FileSystem
PUERTO_FILESYSTEM	Numérico	Puerto al cual se deberá conectar con FileSystem
TAM_MEMORIA	Numérico	Tamaño expresado en bytes del espacio de usuario de la memoria.
TAM_PAGINA	Numérico	Tamaño de las páginas en bytes.
PATH_INSTRUCCIONES	String	Carpeta donde se encuentran los archivos de pseudocódigo.
RETARDO_RESPUESTA	Numérico	Tiempo en milisegundos que se deberá esperar antes de responder a las solicitudes de CPU y FS.
ALGORITMO_REEMPLAZO	String	Algoritmo de reemplazo de páginas (FIFO / LRU).

Ejemplo de Archivo de Configuración

```
PUERTO_ESCUCHA=8002
IP_FILESYSTEM=127.0.0.1
PUERTO_FILESYSTEM=8003
TAM_MEMORIA=4096
```

```
TAM_PAGINA=16
PATH_INSTRUCCIONES=/home/utnso/mappa-pruebas
RETARDO_RESPUESTA=1000
ALGORITMO_REEMPLAZO=FIFO
```

Módulo: File System

Lineamiento e Implementación

El módulo file system será el encargado de almacenar de manera persistente toda la información del trabajo práctico, para ello utilizará 2 particiones las cuales deberán estar implementadas dentro del mismo archivo, la primera será un espacio contiguo el cual contendrá las páginas de la memoria (para manejo de SWAP) y la otra partición se basará en un esquema similar a un esquema FAT para la administración de los archivos de los usuarios.

Para su implementación, deberá ser un proceso multihilo capaz de atender todas las peticiones de manera concurrente, a tales efectos, se recomienda implementar cada petición como una conexión nueva.

Estructuras

El módulo File System contará con 3 estructuras principales:

FCB

La estructura de *File Control Block* que se utilizará tomará como referencia una versión simplificada de un FCB de un sistema de archivos FAT.

- Nombre del archivo[2]: 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*.
- Bloque inicial: Apunta al primer bloque de datos del archivo.

Ejemplo de FCB

```
NOMBRE_ARCHIVO=Notas1erParcialK9999
TAMANIO_ARCHIVO=64
BLOQUE_INICIAL=12
```

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, estos archivos deberán deberan llamarse de igual manera que el nombre del archivo con la extensión fcb[3], por ejemplo `Notas1erParcialK9999.fcb`

File Allocation Table

La File Allocation Table será un archivo binario que contendrá una entrada por cada bloque de la partición FAT, el tamaño de la misma sale de la resta de CANT_BLOQUES_TOTAL - CANT_BLOQUES_SWAP.

[2] 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.
[3] Las extensiones del archivo no afectan a su uso.

Cada entrada será representada por un dato de tipo `uint32_t`, es decir que el tamaño en bytes del archivo FAT será `(CANT_BLOQUES_TOTAL - CANT_BLOQUES_SWAP) * sizeof(uint32_t)`.

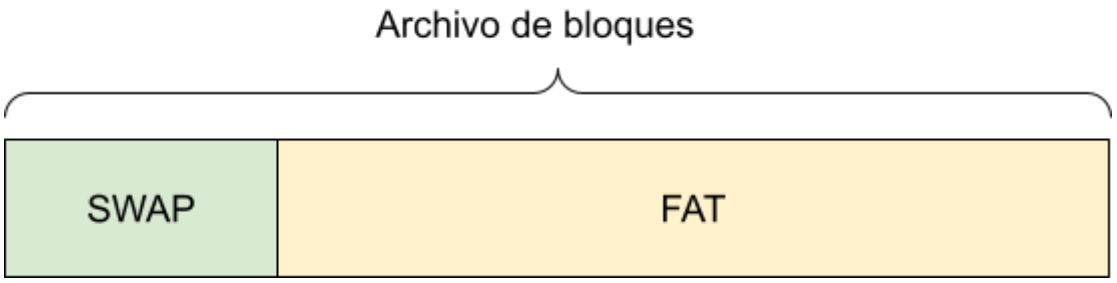
Si al iniciar el FS, no se encuentra el archivo de la tabla FAT, se deberá crear el mismo e inicializarlo con todos valores en 0 que representará que dicho bloque lógico está libre. Cabe resaltar que el bloque lógico 0 estará reservado[4] y no debe ser asignado a ningún archivo.

El valor correspondiente a un EOF (End of File) será representado como la constante `UINT32_MAX`.

Archivo de bloques

Este archivo es un archivo que contendrá todos los bloques de nuestro File System, el mismo se puede considerar como un array de bloques y deberá estar implementado como un archivo binario[5], cualquier implementación que permita leer el archivo con un editor de textos normal, será considerada una implementación errónea.

Este archivo estará dividido lógicamente en dos particiones donde la primera será el esquema SWAP de la memoria. De esta manera, el bloque 0 del sistema FAT iniciará al finalizar la partición de SWAP.



Comunicación con Kernel y Memoria

Peticiones del módulo Kernel

Abrir Archivo

La operación de abrir archivo consistirá en verificar que exista el FCB correspondiente al archivo.

- En caso de que exista deberá devolver el tamaño del archivo.
- En caso de que no exista, deberá informar que el archivo no existe.

Crear Archivo

En la operación crear archivo, se deberá crear un archivo FCB con tamaño 0 y sin bloque inicial.

---

[4] En un esquema FAT real suelen haber varios bloques reservados al comienzo de la partición, siendo el bloque 0 reservado para el Boot.

[5] Para su lectura se utilizarán funciones de la consola de linux tales como hexdump o xxd.

Para interpretar el contenido leído o a escribir desde C, es posible usar el inspector de memoria del debugger y las funciones de memoria de las commons.

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).

Siempre se van a poder truncar archivos para ampliarlos, no se realizará la prueba de llenar el FS.

Leer Archivo

Esta operación leerá la información correspondiente al bloque a partir del puntero.

La información se deberá enviar al módulo **Memoria** para ser escrita a partir de la dirección física recibida por parámetro, una vez recibida la confirmación por parte del módulo **Memoria**, se informará al módulo **Kernel** del éxito de la operación.

Escribir Archivo

Se deberá solicitar al módulo **Memoria** la información que se encuentra a partir de la dirección física recibida y se escribirá en el bloque correspondiente del archivo a partir del puntero recibido.

Nota: El tamaño de la información a leer/escribir de la memoria coincidirá con el tamaño del bloque / página. Siempre se leerá/escribirá un bloque completo, los punteros utilizados siempre serán el 1er byte del bloque o página.

Peticiones del módulo Memoria

Iniciar Proceso

El módulo **Memoria** solicitará que se reserve una cantidad de bloques de SWAP, como respuesta el módulo **File System** enviará el listado de los bloques reservados al proceso.

Al momento de reservar un bloque, el mismo deberá rellenarse con `'\0'`, el algoritmo y las estructuras necesarias para la gestión de estos bloques serán definidas por el grupo.

Finalizar Proceso

El módulo **Memoria** solicitará que se marquen como libres los bloques de SWAP que se envían como parámetro de la solicitud.

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>"

Lectura de Archivo: "Leer Archivo: <NOMBRE_ARCHIVO> - Puntero: <PUNTERO ARCHIVO> - Memoria: <DIRECCION MEMORIA>"

Escritura de Archivo: "Escribir Archivo: <NOMBRE_ARCHIVO> - Puntero: <PUNTERO ARCHIVO> - Memoria: <DIRECCION MEMORIA>"

Acceso a FAT: "Acceso FAT - Entrada: <NRO_ENTRADA_FAT> - Valor: <VALOR_ENTRADA_FAT>"

Acceso a Bloque Archivo: "Acceso Bloque - Archivo: <NOMBRE_ARCHIVO> - Bloque Archivo: <NUMERO_BLOQUE_ARCHIVO> - Bloque FS: <NUMERO_BLOQUE_FS>"

Acceso a Bloque SWAP: "Acceso SWAP: <NRO_BLOQUE>"

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_FAT	String	Path al archivo que contiene la tabla FAT
PATH_BLOQUES	String	Path al archivo que contiene los bloques.
PATH_FCB	String	Path de la carpeta que contiene los archivos FCB
CANT_BLOQUES_TOTAL	Numérico	Cantidad total de bloques del sistema
CANT_BLOQUES_SWAP	Numérico	Cantidad de bloques reservados para la partición de SWAP
TAM_BLOQUE	Numérico	Tamaño de cada bloque del file system[6]
RETARDO_ACCESO_BLOQUE	Numérico	Tiempo en milisegundos que se deberá esperar ante cada acceso a un bloque.

[6] El tamaño de bloque será siempre configurado igual al tamaño de página del módulo Memoria.

Campo	Tipo	Descripción
RETARDO_ACCESO_FAT	Numérico	Tiempo en milisegundos que se deberá esperar ante cada acceso a la tabla FAT.

Ejemplo de Archivo de Configuración

```
IP_MEMORIA=127.0.0.1
PUERTO_MEMORIA=8002
PUERTO_ESCUCHA=8003
PATH_FAT=/home/utnso/fs/fat.dat
PATH_BLOQUES=/home/utnso/fs/bloques.dat
PATH_FCB=/home/utnso/fs/fcbs
CANT_BLOQUES_TOTAL=1024
CANT_BLOQUES_SWAP=64
TAM_BLOQUE=1024
RETARDO_ACCESO_BLOQUE=2500
RETARDO_ACCESO_FAT=500
```

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: 09/09/2023

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: 30/09/2023

Objetivos:

- **Módulo Kernel:**
  - Crea las conexiones con la Memoria, la CPU y el File System.
  - Inicia la consola interactiva y permite iniciar y finalizar procesos.
  - Generar estructuras base para administrar los PCB y sus estados.
  - Planificador de Largo Plazo funcionando, respetando el grado de multiprogramación.
  - Planificador de Corto Plazo funcionando con algoritmo FIFO.
- **Módulo CPU:**
  - Lee el archivo de configuración.
  - Se conecta a Memoria y espera conexiones del Kernel.
  - Ejecuta las instrucciones SET, SUM, SUB, y EXIT.
- **Módulo Memoria:**
  - Crea las estructuras necesarias para leer el archivo de pseudocódigo.
  - Es capaz de responder a la petición de instrucciones por parte del módulo CPU.
- **Módulo File System:**
  - Es capaz de leer su archivo de configuración y se conecta a los demás módulos.

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: 21/10/2023

Objetivos:

- **Realizar pruebas mínimas en un entorno distribuido.**
- **Módulo Kernel:**
  - Planificador de Corto plazo funcionando para los algoritmos RR y Prioridades.
  - Manejo de estado de bloqueado sin Page Fault.
- **Módulo CPU:**
  - Implementa ciclo de instrucción completo.
  - Interpreta todas las operaciones.
  - Ejecuta correctamente SLEEP, WAIT y SIGNAL.
- **Módulo Memoria:**
  - Espera las peticiones de los demás módulos y responde con mensajes genéricos.
- **Módulo File System:**
  - Responde de manera genérica a los mensajes de Kernel y Memoria
  - Levanta los archivos de Bloques, FAT y FCBs

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: 11/11/2023

Objetivos:

- **Realizar pruebas mínimas en un entorno distribuido.**
- **Módulo Kernel**
  - Recibe las funciones de la CPU.
  - Solicita a la memoria la creación y/o finalización de procesos.
  - 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 tablas de Páginas
  - Responde los mensajes de CPU y Kernel con datos reales.
  - Respuesta Page Fault de manera genérica sin acciones.
- **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: 02/12/2023 - 09/12/2023 - 16/12/2023

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