

# *Master of Files*

*I'm querying your blocks*



*Twisting your data and smashing your duplicates*

Cátedra de Sistemas Operativos - UTN FRBA

Trabajo práctico Cuatrimestral

- 2C2025 -

Versión 1.1

# Índice

<b>Índice</b>	<b>2</b>
<b>Historial de Cambios</b>	<b>4</b>
<b>Objetivos del Trabajo Práctico</b>	<b>5</b>
Características	5
Evaluación del Trabajo Práctico	5
Deployment y Testing del Trabajo Práctico	6
<b>Definición del Trabajo Práctico</b>	<b>7</b>
¿Qué es el trabajo práctico y cómo empezamos?	7
Arquitectura del sistema	8
Aclaración importante	8
<b>Módulo: Query Control</b>	<b>9</b>
Lineamiento e Implementación	9
Logs mínimos y obligatorios	9
Archivo de Configuración	9
Ejemplo de Archivo de Configuración	10
<b>Módulo: Master</b>	<b>11</b>
Lineamiento e Implementación	11
Planificación	11
Algoritmos	11
FIFO	12
Prioridades con desalojo y aging	12
Aging	12
Desconexión de Query Control	12
Conexión y desconexión de Workers	12
Logs mínimos y obligatorios	12
Archivo de Configuración	13
Ejemplo de Archivo de Configuración	13
<b>Módulo: Worker</b>	<b>14</b>
Lineamiento e Implementación	14
Query Interpreter	14
Instrucciones	15
CREATE	15
TRUNCATE	15
WRITE	15
READ	15
TAG	15
COMMIT	16
FLUSH	16
DELETE	16
END	16
Ejemplo de archivo de Query	16
Memoria Interna	16
Algoritmos de Reemplazo de páginas	17

Logs mínimos y obligatorios	17
Archivo de Configuración	18
Ejemplo de Archivo de Configuración	18
<b>Módulo: Storage</b>	<b>19</b>
Lineamiento e Implementación	19
Estructura de directorios y archivos	19
Archivo superblock.config	19
Archivo bitmap.bin	19
Archivo blocks_hash_index.config	20
Función de Hash	20
Directorio ‘physical_blocks’	20
Directorio ‘files’	21
Inicialización	22
Estructura interna	22
Operaciones	23
Creación de File	23
Truncado de archivo	23
Tag de File	23
Commit de un Tag	23
Escritura de Bloque	24
Lectura de Bloque	24
Eliminar un tag	24
Errores en las operaciones	24
File / Tag inexistente	24
File / Tag preexistente	24
Escritura no permitida	24
Lectura o escritura fuera de límite	25
Retardos en las operaciones	25
Logs mínimos y obligatorios	25
Archivo de Configuración	26
Ejemplo de Archivo de Configuración	26
<b>Descripción de las entregas</b>	<b>27</b>
Check de Control Obligatorio 1: Conexión inicial	27
Check de Control Obligatorio 2: Ejecución Básica	27
Check de Control Obligatorio 3: Comenzando a Persistir	28
Entregas Finales	28

## **Historial de Cambios**

v1.0 (02/09/2025) Publicación del enunciado

v1.1 (03/10/2025) Ajustes de redacción y aclaraciones

- *Se agrega que todos los módulos reciban el archivo de config como parámetro de ejecución.*
- *Se ajustaron algunos logs obligatorios para facilitar su implementación.*
- *Se corrigieron algunos logs obligatorios donde debería decir File en lugar de archivo.*

# 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.

Debido al fin académico del trabajo práctico, los conceptos que se verán 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.

## Características

- Modalidad: grupal (5 integrantes  $\pm 0$ ) y obligatorio
- Fecha de comienzo: 02/09/2025
- Fecha de primera entrega: 29/11/2025
- Fecha de segunda entrega: 06/12/2025
- Fecha de tercera entrega: 20/12/2025
- 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 o logs de su funcionamiento de la forma más clara posible, para ello se les proveerá en cada módulo un listado de logs mínimos y obligatorios.

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.

## 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:** Contendrá la definición funcional y aspectos de implementación técnica obligatorios del módulo en cuestión. La no inclusión de alguno de los puntos especificados en este título puede conllevar a la desaprobación del trabajo práctico.
- **Archivos de Configuración:** Se describirán los parámetros mínimos requeridos para ajustar el comportamiento de cada módulo ante cada prueba, sin recomilar. Durante la evaluación, deberá alcanzar con detener la ejecución, modificar el archivo de configuración y volver a ejecutar el módulo. En caso de que el grupo requiera de algún parámetro extra, podrá agregarlo.

Cada módulo contará con un listado de **logs mínimos y obligatorios** los cuales deberán realizarse utilizando la biblioteca de [so-commons-library](#) provista por la cátedra y los mismos deberán estar como LOG\_LEVEL\_INFO, pudiendo ser extendidos por necesidad del grupo utilizando LOG\_LEVEL\_DEBUG.

**En caso de no cumplir con los logs mínimos y/o no guardarlos en archivo, *se considerará que el TP no es apto para ser evaluado* y por consecuencia el mismo estará *desaprobado*.**

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](#), dado que las justificaciones deberán ser expuestas en el coloquio.

## ¿Qué es el trabajo práctico y cómo empezamos?

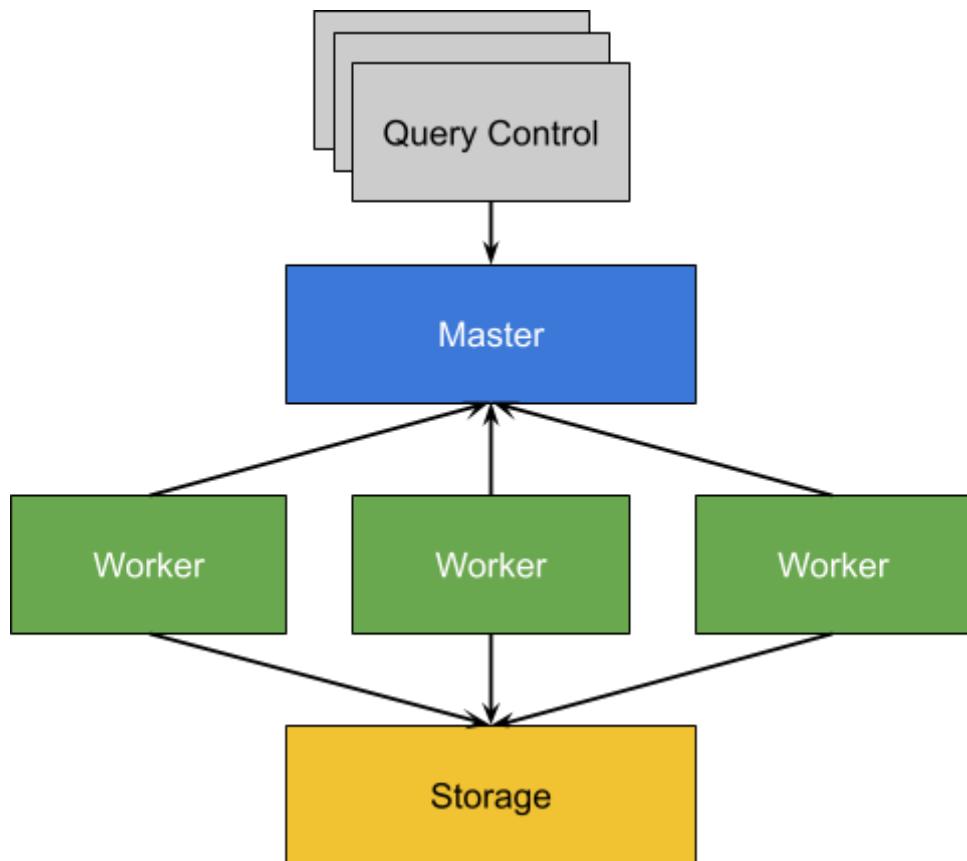
El objetivo del trabajo práctico consiste en desarrollar una solución que permita la simulación de un sistema distribuido, donde los grupos tendrán que planificar procesos, resolver peticiones al sistema, y administrar de manera adecuada una memoria bajo los esquemas explicados en sus correspondientes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

Este trabajo práctico correrá una serie de módulos, los cuales deberán ser capaces de correr en diferentes computadoras y/o máquinas virtuales.



**NOTA:** El sentido de las flechas indica dependencias entre módulos. Ejemplo: al momento de iniciar la ejecución del Worker es necesario contar con el Master y el Storage previamente iniciados.

## Aclaración importante

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

# Módulo: Query Control

El módulo Query Control es el encargado de enviar a ejecutar una Query al sistema. Dicha Query estará representada por un archivo que incluirá una serie de instrucciones relativas al manejo de archivos (de ahora en más, “**Files**”) que serán operados en el sistema.

## Lineamiento e Implementación

Cada Query Control al momento de iniciarse deberá poder recibir como mínimo los siguientes parámetros de ejecución<sup>1</sup>:

- Archivo de configuración, el cual se encuentra detallado más abajo.
- Archivo de Query, es el nombre del archivo.
- Prioridad, siendo un valor numérico mayor o igual a 0.

```
➔ ~ ./bin/query [archivo_config] [archivo_query] [prioridad]
```

Una vez iniciado el módulo, el mismo deberá conectarse al Módulo Master, enviarle el nombre del **archivo\_query** y la prioridad, y deberá quedarse a la espera de los mensajes que el módulo Master le envíe. Estos mensajes podrán contener información leída por la Query o el aviso de finalización de la Query.

## Logs mínimos y obligatorios

**Conexión al master:** “## Conexión al Master exitosa. IP: <ip>, Puerto: <puerto>”

**Envío de Query:** “## Solicitud de ejecución de Query: <archivo\_query>, prioridad: <prioridad>”

**Lectura de File:** “## Lectura realizada: File <File:Tag>, contenido: <CONTENIDO>”

**Finalización de la Query:** “## Query Finalizada - <MOTIVO>”

## Archivo de Configuración

Campo	Tipo	Descripción
IP_MASTER	String	IP del módulo Master
PUERTO_MASTER	Numérico	Puerto del módulo Master
LOG_LEVEL	String	Nivel de detalle máximo a mostrar. Compatible con <a href="#">log_level_from_string()</a>

---

<sup>1</sup> Se recomienda consultar la guía de [argumentos para el main](#).

## Ejemplo de Archivo de Configuración

```
IP_MASTER=127.0.0.1  
PUERTO_MASTER=9001  
LOG_LEVEL=INFO
```

## Módulo: Master

El módulo **Master** será el encargado de la gestión de las diferentes peticiones (Queries) que se envían desde los módulos Query Control.

### Lineamiento e Implementación

Al momento de iniciarse deberá poder recibir como mínimo el siguiente parámetro de ejecución:

- Archivo de configuración, el cual se encuentra detallado más abajo.

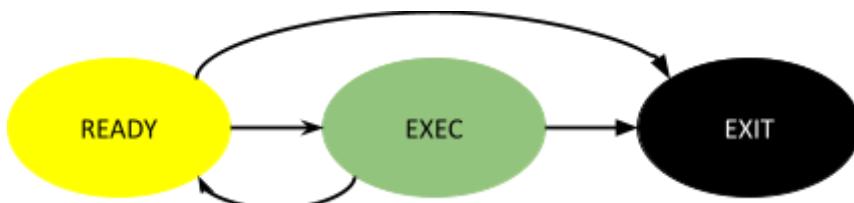
```
➔ ~ ./bin/master [archivo_config]
```

Para gestionar estas Queries, el módulo Master dispondrá de un puerto en el cual escuchará las conexiones de los Query Control. Estas conexiones **se deberán mantener** hasta que la Query enviada por el Query Control finalice.

Durante la ejecución de las Queries, el módulo Master recibirá mensajes de las lecturas realizadas por el Worker correspondiente, los cuales deberán reenviarse al Query Control asociado.

### Planificación

El Módulo Master será el encargado de planificar las Queries que le lleguen de parte de los Query Control. Las mismas se pueden encontrar en alguno de los siguientes estados:



Al momento de recibir una Query por parte de un Query Control, se le asignará un identificador único en todo el sistema, que será un valor autoincremental iniciado en 0.

Una vez hecho esto, se la dejará en el estado READY, donde estará a la espera de ser enviada a un Worker en base al algoritmo de planificación elegido. Al momento de que se seleccione un Worker, la Query pasará del estado READY al estado EXEC, y se procederá a esperar los mensajes de lectura recibidos para ser reenviados al Query Control, o eventualmente el resultado final de la ejecución.

La cantidad de Workers conectados nos indicará el grado de multiprocesamiento del sistema. El proceso Master será el responsable de indicarle a cada Worker qué Query deberá ejecutar.

### Algoritmos

Los algoritmos a utilizar en la planificación de corto plazo son:

- FIFO
- Prioridades con desalojo y aging

## FIFO

Las ejecuciones de las Queries se irán asignando a cada Worker por orden de llegada. Una vez que todos los Workers tengan una Query asignada se encolará el resto de las Queries en el estado READY.

## Prioridades con desalojo y aging

Las Queries se enviarán a ejecutar según su prioridad, siendo 0 la prioridad más alta (a mayor número, menor prioridad).

En caso que los Workers estuvieran ocupados y una Query nueva tuviera más prioridad que alguna de las que estuviera en ejecución, deberá ser *desalojada* aquella que tuviera la menor prioridad.

Para desalojar una Query de un Worker, se deberá solicitar al Worker su desalojo. Este último le devolverá al Master el Program Counter (PC) para que éste luego sepa desde dónde reanudarlo.

Al momento de volver a planificar una Query previamente desalojada, se deberá enviar el PC previamente recibido para poder retomar en el lugar correcto.

## Aging

Para que una Query de baja prioridad no sufra starvation, se implementará un esquema de aging tal que se aumente la prioridad de aquellas Queries que estén en READY por un período prolongado de tiempo. A cada Query se le irá reduciendo en 1 (uno) su número de prioridad cada vez que se cumpla cierto intervalo en milisegundos en READY (definido por archivo de configuración), hasta que la misma sea seleccionada para ejecutar.

## Desconexión de Query Control

Ante la desconexión de un módulo Query Control, la Query enviada por el mismo deberá iniciar el proceso de cancelación inmediatamente. En el caso de que la Query se encuentre en READY, la misma se deberá enviar a EXIT directamente. En caso de que la Query se encuentre en EXEC, se deberá notificar al Worker que la está ejecutando que debe desalojar dicha Query y una vez recibido su contexto se enviará la Query a EXIT.

## Conexión y desconexión de Workers

Los módulos Worker se podrán conectar y desconectar en tiempo de ejecución. En caso de que un nuevo Worker se conecte al Master, se deberá planificar la siguiente Query según el algoritmo configurado.

Al momento de que un Worker se desconecte, la Query que se encontraba en ejecución en dicho Worker se finalizará con error y se notificará al Query Control correspondiente.

## Logs mínimos y obligatorios

**Conexión de Query Control:** “## Se conecta un Query Control para ejecutar la Query <PATH\_QUERY> con prioridad <PRIORIDAD> - Id asignado: <QUERY\_ID>. Nivel multiprocesamiento <CANTIDAD>”

**Conexión de Worker:** “## Se conecta el Worker <WORKER\_ID> - Cantidad total de Workers: <CANTIDAD>”

**Desconexión de Query Control:** “## Se desconecta un Query Control. Se finaliza la Query <QUERY\_ID> con prioridad <PRIORIDAD>. Nivel multiprocesamiento <CANTIDAD>”

**Desconexión de Worker:** “## Se desconecta el Worker <WORKER\_ID> - Se finaliza la Query <QUERY\_ID> - Cantidad total de Workers: <CANTIDAD> ”

**Envío de Query a Worker:** “## Se envía la Query <QUERY\_ID> (<PRIORIDAD>) al Worker <WORKER\_ID>”

**Desalojo de Query en Worker:** “## Se desaloja la Query <QUERY\_ID> (<PRIORIDAD>) del Worker <WORKER\_ID> - Motivo: <DESCONEXION / PRIORIDAD>”

**Cambio de prioridad de Query:** “##<QUERY\_ID> Cambio de prioridad: <PRIORIDAD\_ANTERIOR> - <PRIORIDAD\_NUEVA>”

**Finalización de Query en Worker:** “## Se terminó la Query <QUERY\_ID> en el Worker <WORKER\_ID>”

**Envío de lectura de Query a Query Control:** “## Se envía un mensaje de lectura de la Query <QUERY\_ID> en el Worker <WORKER\_ID> al Query Control”

## Archivo de Configuración

Campo	Tipo	Descripción
PUERTO_ESCUCHA	Numérico	Puerto al cual se deberán conectar los procesos Query Control y Workers
ALGORITMO_PLANIFICACION	String	Algoritmo de planificación a utilizar FIFO / PRIORIDADES
TIEMPO_AGING	Numérico	Tiempo en milisegundos que deberán pasar antes de que se le reduzca el número de prioridad. El valor 0 (cero) implicará “sin aging”
LOG_LEVEL	String	Nivel de detalle máximo a mostrar. Compatible con <a href="#">log_level_from_string()</a>

### Ejemplo de Archivo de Configuración

```
PUERTO_ESCUCHA=9001
ALGORITMO_PLANIFICACION=PRIORIDADES
TIEMPO_AGING=2500
LOG_LEVEL=INFO
```

## Módulo: Worker

El Módulo Worker en el contexto de nuestro trabajo práctico será el encargado de ejecutar las diferentes Queries, de una a la vez.

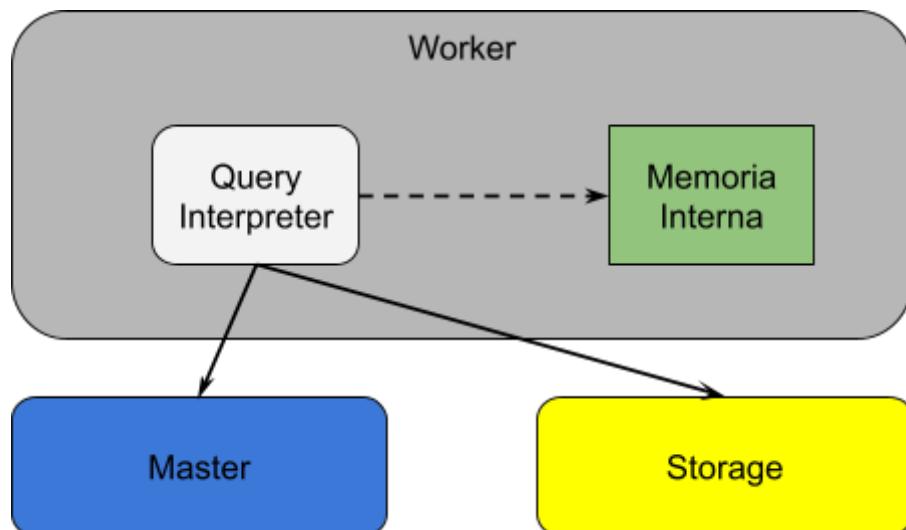
### Lineamiento e Implementación

Cada Worker al momento de iniciarse deberá poder recibir como parámetro el archivo de configuración y un identificador propio del Worker, el cuál se encuentra detallado más abajo

```
→ ~ ./bin/worker [archivo_config] [ID Worker]
```

Al momento de iniciar el Worker, el mismo deberá conectarse al módulo Storage y una vez realizada la conexión deberá conectarse al módulo Master indicando su ID a fin de que el Master pueda identificarlo.

Para poder ejecutar las Queries el módulo Worker contará con una Memoria Interna y un Query Interpreter.



### Query Interpreter

El componente de Query Interpreter, será el encargado de ir ejecutando cada una de las **instrucciones** de las Queries. Estas acciones podrán interactuar directamente sobre el componente de Memoria Interna y/o comunicarse con el Módulo de Storage.

Para esto, el Worker tendrá un path indicado en el archivo de configuración donde se encontrarán todas las Queries. Al recibir del Master la solicitud de ejecutar una nueva Query, deberá buscar el archivo en este Path para parsearlo a partir del PC indicado. De ahí en más, se deberán ejecutar una por una las instrucciones, respetando el retardo configurado.

Los parámetros de las instrucciones se separan por espacios. En el caso de los tags de los Files<sup>2</sup>, siempre se separarán del nombre del File por “:”.

El Query interpreter admite las siguientes instrucciones:

## Instrucciones

### CREATE

Formato: CREATE <NOMBRE\_FILE>:<TAG>

La instrucción CREATE solicitará al módulo Storage la creación de un nuevo File con el Tag recibido por parámetro y con tamaño 0.

### TRUNCATE

Formato: TRUNCATE <NOMBRE\_FILE>:<TAG> <TAMAÑO>

La instrucción TRUNCATE solicitará al módulo Storage la modificación del tamaño del File y Tag indicados, asignando el tamaño recibido por parámetro (deberá ser múltiplo del tamaño de bloque).

### WRITE

Formato: WRITE <NOMBRE\_FILE>:<TAG> <DIRECCIÓN BASE> <CONTENIDO>

La instrucción WRITE escribirá en la Memoria Interna los bytes correspondientes a partir de la dirección base del File:Tag. En caso de que la Memoria Interna no cuente con todas las páginas necesarias para satisfacer la operación, deberá solicitar el contenido faltante al módulo Storage.

### READ

Formato: READ <NOMBRE\_FILE>:<TAG> <DIRECCIÓN BASE> <TAMAÑO>

La instrucción READ leerá de la Memoria Interna los bytes correspondientes a partir de la dirección base del File y Tag pasados por parámetro, y deberá enviar dicha información al módulo Master.

En caso de que la Memoria Interna no cuente con todas las páginas necesarias para satisfacer la operación, deberá solicitar el contenido faltante al módulo Storage.

### TAG

Formato: TAG <NOMBRE\_FILE\_ORIGEN>:<TAG\_ORIGEN>  
<NOMBRE\_FILE\_DESTINO>:<TAG\_DESTINO>

La instrucción TAG solicitará al módulo Storage la creación un nuevo File:Tag a partir del File y Tag origen pasados por parámetro.

---

<sup>2</sup> Para diferenciar los archivos nativos de linux de los archivos que va a gestionar nuestro Filesystem, en todo el enunciado vamos a nombrarlos como “archivos” a los primeros y como “Files” a los segundos

## COMMIT

Formato: COMMIT <NOMBRE\_FILE>:<TAG>

La instrucción COMMIT, le indicará al Storage que no se realizarán más cambios sobre el File y Tag pasados por parámetro.

## FLUSH

Formato: FLUSH <NOMBRE\_FILE>:<TAG>

Persistirá todas las modificaciones realizadas en Memoria Interna de un File:Tag en el Storage.

Nota: Esta instrucción también deberá ser ejecutada implícitamente bajo las siguientes situaciones:

- Previo a la ejecución de un COMMIT.
- Previo a realizar el desalojo de la Query del Worker (para todos los File:Tag eventualmente modificados)

## DELETE

Formato: DELETE <NOMBRE\_FILE>:<TAG>

La instrucción DELETE solicitará al módulo Storage la eliminación del File:Tag correspondiente.

## END

Formato: END

Esta instrucción da por finalizada la Query y le informa al módulo Master el fin de la misma.

## Ejemplo de archivo de Query

```
1  CREATE MATERIAS:BASE
2  TRUNCATE MATERIAS:BASE 1024
3  WRITE MATERIAS:BASE 0 SISTEMAS_OPERATIVOS
4  FLUSH MATERIAS:BASE
5  COMMIT MATERIAS:BASE
6  READ MATERIAS:BASE 0 8
7  TAG MATERIAS:BASE MATERIAS:V2
8  DELETE MATERIAS:BASE
9  WRITE MATERIAS:V2 0 SISTEMAS_OPERATIVOS_2
10 COMMIT MATERIAS:V2
11 END
```

## Memoria Interna

El componente de la Memoria Interna estará implementado mediante un único `malloc()` que contendrá la información que el Query Interpreter necesite leer y/o escribir a lo largo de las diferentes ejecuciones.

El tamaño de la Memoria Interna vendrá delimitado por archivo de configuración y no se podrá cambiar en tiempo de ejecución.

Se utilizará un esquema de *paginación simple* a demanda con memoria virtual, donde el tamaño de página estará definido por el tamaño de bloque del Storage, este dato se debe obtener al momento de hacer el Handshake con el Storage. De esta manera, se deberá mantener una tabla de páginas por cada File:Tag de los que tenga al menos una página presente.

Para cada referencia lectura o escritura del Query Interpreter a una página de la Memoria Interna, se deberá esperar un tiempo definido por archivo de configuración (RETARDO\_MEMORIA).

### Algoritmos de Reemplazo de páginas

En caso de que la Memoria Interna esté llena y deba cargar una nueva página, se deberá elegir una víctima para ser reemplazada.

La elección de la víctima se podrá realizar mediante los algoritmos LRU o CLOCK-M definido por archivo de configuración y el mismo no se modificará en tiempo de ejecución. La víctima podrá ser una página correspondiente a cualquier File:Tag que se encuentre presente en la Memoria Interna del Worker. Si la página víctima se encuentra modificada, se deberá previamente impactar el cambio en el módulo Storage, teniendo en cuenta que los números de página están directamente relacionados al número del bloque lógico dentro del File:Tag.

## Logs mínimos y obligatorios

**Recepción de Query:** “## Query <QUERY\_ID>: Se recibe la Query. El path de operaciones es: <PATH\_QUERY>”

**Fetch Instrucción:** “## Query <QUERY\_ID>: FETCH - Program Counter: <PROGRAM\_COUNTER> - <INSTRUCCIÓN><sup>3</sup>”.

**Instrucción realizada:** “## Query <QUERY\_ID>: - Instrucción realizada: <INSTRUCCIÓN><sup>4</sup>”

**Desalojo de Query:** “## Query <QUERY\_ID>: Desalojada por pedido del Master”

**Lectura/Escritura Memoria:** “Query <QUERY\_ID>: Acción: <LEER / ESCRIBIR> - Dirección Física: <DIRECCION\_FISICA> - Valor: <VALOR LEIDO / ESCRITO>”.

**Asignar Marco:** “Query <QUERY\_ID>: Se asigna el Marco: <NUMERO\_MARCO> a la Página: <NUMERO\_PAGINA> perteneciente al - File: <FILE> - Tag: <TAG>”.

**Liberación de Marco:** “Query <QUERY\_ID>: Se libera el Marco: <NUMERO\_MARCO> perteneciente al - File: <FILE> - Tag: <TAG>”.

**Reemplazo Algoritmo:** “## Query <QUERY\_ID>: Se reemplaza la página <File1:Tag1>/<NUM\_PAG1> por la <File2:Tag2><NUM\_PAG2>”

---

<sup>3</sup> Los parámetros no se deben loguear

<sup>4</sup> Los parámetros no se deben loguear

**Bloque faltante en Memoria:** “Query <QUERY\_ID>: - Memoria Miss - File: <FILE> - Tag: <TAG> - Página: <NUMERO\_PAGINA>”

**Bloque ingresado en Memoria:** “Query <QUERY\_ID>: - Memoria Add - File: <FILE> - Tag: <TAG> - Página: <NUMERO\_PAGINA> - Marco: <NUMERO\_MARCO>”

## Archivo de Configuración

Campo	Tipo	Descripción
IP_MASTER	String	IP del módulo Master
PUERTO_MASTER	Numérico	Puerto del módulo Master
IP_STORAGE	String	IP del módulo Storage
PUERTO_STORAGE	Numérico	Puerto del módulo Storage
TAM_MEMORIA <sup>5</sup>	Numérico	Tamaño expresado en bytes de la Memoria Interna.
RETARDO_MEMORIA	Numérico	Tiempo en milisegundos que deberá esperarse ante cada lectura y/o escritura en la Memoria.
ALGORITMO_REEMPLAZO	String	Algoritmo de reemplazo para las páginas. LRU / CLOCK-M
PATH_QUERIES	String	Path donde se encuentran los archivos de Queries
LOG_LEVEL	String	Nivel de detalle máximo a mostrar. Compatible con <a href="#">log_level_from_string()</a>

## Ejemplo de Archivo de Configuración

```
IP_MASTER=127.0.0.1
PUERTO_MASTER=9001
IP_STORAGE=127.0.0.1
PUERTO_STORAGE=9002
TAM_MEMORIA=4096
RETARDO_MEMORIA=1500
ALGORITMO_REEMPLAZO=LRU
PATH_SCRIPTS=/home/utnso/queries
LOG_LEVEL=INFO
```

---

<sup>5</sup> El tamaño de memoria siempre va a ser un múltiplo de tamaño de página.

# Módulo: Storage

Este módulo representará un sistema de archivos (File System) sobre el que los Workers pueden operar de forma concurrente.

## Lineamiento e Implementación

Al iniciar leerá su archivo de configuración, de ser necesario se crearán los archivos y estructuras necesarias para inicializar el FS, de lo contrario leerá los archivos ya existentes para poder atender las peticiones de los Workers. Creará un servidor multihilo y quedará a la espera de las conexiones de los Workers.

El módulo Storage al momento de iniciarse deberá poder recibir como mínimo el siguiente parámetro de ejecución:

- Archivo de configuración, el cual se encuentra detallado más abajo.

```
➔ ~ ./bin/storage [archivo_config]
```

## Estructura de directorios y archivos

El FS se montará sobre un path (ruta) definido por archivo de configuración que llamaremos raíz.

Este directorio raíz deberá contener 3 archivos nativos:

- Un archivo “superblock.config” que contendrá la información administrativa del FS.
- Un archivo “bitmap.bin” que indicará el estado de los bloques del FS (libre/ocupado).
- Un archivo “blocks\_hash\_index.config” que contendrá un hash por cada bloque reservado del FS.

A su vez, contendrá 2 directorios nativos:

- Un directorio “physical\_blocks”, donde se encontrará el espacio de bloques físicos del volumen del FS
- Un directorio nativo “files” donde se encontrará el espacio lógico del FS.

### Archivo superblock.config

Debe ser un archivo compatible con las config de la biblioteca commons y contendrá mínimamente:

- Tamaño del FS en bytes
- Tamaño de Bloque en bytes

Ejemplo:

```
FS_SIZE=4096  
BLOCK_SIZE=128
```

### Archivo bitmap.bin

El objetivo de este archivo es identificar los bloques físicos reservados en nuestro File System.

Indicará para cada bloque físico su estado (libre/ocupado) utilizando *un bit* por cada bloque (estructura conocida como bitarray).<sup>67</sup>

**Cualquier implementación que no resguarde el contenido del bitmap como array de bits es motivo de desaprobación del TP.**

### Archivo blocks\_hash\_index.config

El objetivo de este archivo es asociar cada bloque físico *ocupado* con un identificador único (hash), generado a partir del contenido del mismo. De esta manera, dos bloques lógicos con el mismo contenido podrán ser asociados al mismo bloque físico al tener el mismo hash.

Un ejemplo de este archivo es el siguiente:

```
4d186321c1a7f0f354b297e8914ab240=block0000
ea5cb8ff0abf6b4ca0080069daaeada0=block0001
67d6c28fac7541d9ce1f46ba4f84e149=block0002
749dfe7c0cd3b291ec96d0bb8924cb46=block0003
5058f1af8388633f609cadb75a75dc9d=block0004
```

### Función de Hash

Se utilizará el algoritmo Message-Digest Algorithm 5 (MD5), el cual, en base a un contenido (input), genera un resultado (digest) de 128 bits, representados por una cadena de 32 caracteres hexadecimales.

Para realizar dicho cálculo, se debe utilizar la función [crypto\\_md5\(\)](#) de la biblioteca commons.

### Directorio ‘physical\_blocks’

Este directorio representará cada bloque físico del FS como un archivo de tamaño fijo definido por archivo de superbloque, por ejemplo:

```
/home/utnso/storage/
↳ ./physical_blocks/
↳   ./block0000.dat
↳   ./block0001.dat
↳   ./block0002.dat
↳   ./block0003.dat
↳   ./block0004.dat
...
↳   ./block9997.dat
↳   ./block9998.dat
↳   ./block9999.dat
```

<sup>6</sup> Deberá ser compatible con el formato de bitarray usado por la biblioteca commons.

<sup>7</sup> Se recomienda investigar el uso de la función [mmap\(\)](#) para facilitar su implementación.

## Directorio ‘files’

Este directorio representará cada archivo dentro de nuestro FS, que a partir de este momento lo llamaremos **File**, como un directorio nativo, siendo el nombre del directorio el nombre que tendrá el *File* dentro del FS. Teniendo esto en cuenta, no podrán haber múltiples *Files* con el mismo nombre dentro de este FS.

Cada directorio dentro del *File* contendrá un subdirectorio nativo por cada **Tag** del mismo dentro del FS, siendo el nombre del directorio el nombre que tendrá el *Tag* dentro del FS. Similarmente a los *Files*, no podrán haber múltiples *Tags* con el mismo nombre para un mismo File.

```
/home/utnso/storage/
    ↳ ./physical_blocks # physical space
    ↳ ./files/ # logical space
        ↳ ./arch1/
            ↳ ./tag_1_0_0/
                ↳ ./metadata.config
                ↳ ./logical_blocks/
                    ↳ ./000000.dat
                    ↳ ./000001.dat
                    ↳ ./000002.dat
            ↳ ./tag_2_0_0/
                ↳ ./metadata.config
                ↳ ./logical_blocks/
                    ↳ ./000000.dat
                    ↳ ./000001.dat
                    ↳ ./000002.dat
```

Dentro de cada uno de estos directorios de tags tendrá un archivo compatible con las config de la biblioteca commons llamado “metadata.config”, el cuál contendrá el tamaño del File:Tag, el estado del File:Tag (el cual podrá ser WORK\_IN\_PROGRESS o COMMITED) y la lista ordenada de los números de bloques físicos del FS que pertenecen al File:Tag, por ejemplo:

```
TAMAÑO=160
BLOCKS=[17,2,5,10,8]
ESTADO=WORK_IN_PROGRESS
```

Por último, con el objetivo de tener una visión más directa de los bloques lógicos del File:Tag, se deberá tener también un directorio nativo “logical\_blocks” cuyo contenido será de un archivo nativo por cada bloque lógico del File:Tag. Cada uno de estos archivos nativos será un hard link<sup>8</sup> al bloque físico que corresponde del FS.

---

<sup>8</sup> Se recomienda investigar el uso de la función [link\(\)](#)

## Inicialización

Al momento de inicializar el FS lo primero que se debe verificar es el valor FRESH\_START del archivo de configuración. Este valor nos indicará si se debe formatear el volumen (inimando un FS nuevo) o si se quiere mantener el contenido preexistente.

Al inicializar desde cero el FS (FRESH\_START=TRUE), el único archivo nativo que necesitamos tener obligatoriamente es el archivo `superblock.config`. Ese archivo nos indicará los datos necesarios para poder formatear nuestro FS, eliminando previamente los demás archivos que componen nuestro FS en caso de existir.

También se deberá crear un primer File llamado “initial\_file” confirmado con un único Tag “BASE”, cuyo contenido será un (1) bloque lógico con el bloque físico nro 0 (cero) asignado, completando el bloque con el carácter 0, por ejemplo: “00000...”. Dicho File/Tag no se podrá borrar.

## Estructura interna

Un ejemplo de cómo se vería la estructura completa de directorios y archivos nativos de nuestro TP sería el siguiente:

```
/home/utnso/storage/
↳ ./superblock.config
↳ ./bitmap.bin
↳ ./blocks_hash_index.config
↳ ./physical_blocks/
↳   ./block0000.dat
↳   ./block0001.dat
↳   ./block0002.dat
↳   ./block0003.dat
↳   ./block0004.dat
↳ ./files/
↳   ./initial_file/
↳     ./BASE/
↳       ./metadata.config
↳       ./logical_blocks/
↳         ./000000.dat # hard link a ‘block0000.dat’
↳ ./arch1/
↳   ./tag_1_0_0/
↳     ./metadata.config
↳     ./logical_blocks/
↳       ./000000.dat # hard link a ‘block0003.dat’
↳       ./000001.dat # hard link a ...
↳       ./000002.dat
↳   ./tag_2_0_0/
↳     ./metadata.config
↳     ./logical_blocks/
↳       ./000000.dat
↳       ./000001.dat
↳       ./000002.dat
↳   ./tag_3_0_0/
↳     ./metadata.config
↳     ./logical_blocks/
↳       ./000000.dat
↳       ./000001.dat
```

```
↳ ./000002.dat  
↳ ./000003.dat
```

## Operaciones

El storage ofrecerá una serie de operaciones, susceptibles de ser solicitadas por los Workers. Las mismas no deberán ser confundidas con las instrucciones de una Query a ser interpretadas por los Workers. Las instrucciones ejecutadas en los Workers implicarán la ejecución de una o más operaciones del Storage.

### Creación de File

Esta operación creará un nuevo File dentro del FS. Para ello recibirá el nombre del File y un Tag inicial para crearlo.

Deberá crear el archivo de metadata en estado WORK\_IN\_PROGRESS y no asignarle ningún bloque.

### Truncado de archivo

Esta operación se encargará de modificar el tamaño del File:Tag especificados agrandando o achicando el tamaño del mismo para reflejar el nuevo tamaño deseado (actualizando la metadata necesaria).

Al incrementar el tamaño del File, se le asignarán tantos bloques lógicos (hard links) como sea necesario. Inicialmente, todos ellos deberán apuntar el bloque físico nro 0.

Al reducir el tamaño del File, se deberán desasignar tantos bloques lógicos como sea necesario (empezando por el final del archivo). Si el bloque físico al que apunta el bloque lógico eliminado no es referenciado por ningún otro File:Tag, deberá ser marcado como libre en el bitmap<sup>9</sup>.

### Tag de File

Esta operación creará una copia completa del directorio nativo correspondiente al Tag de origen en un nuevo directorio correspondiente al Tag destino y modificará en el archivo de metadata del Tag destino para que el mismo se encuentre en estado WORK\_IN\_PROGRESS.

### Commit de un Tag

Confirmará un File:Tag pasado por parámetro. En caso de que un Tag ya se encuentre confirmado, esta operación no realizará nada. Para esto se deberá actualizar el archivo metadata del Tag pasando su estado a “COMMITTED”.

Se deberá, por cada bloque lógico, buscar si existe algún bloque físico que tenga el mismo contenido (utilizando el hash y archivo blocks\_hash\_index.config). En caso de encontrar uno, se deberá liberar el bloque físico actual y reapuntar el bloque lógico al bloque físico pre-existente. En caso contrario, simplemente se agregará el hash del nuevo contenido al archivo blocks\_hash\_index.config.

---

<sup>9</sup> Para la implementación de esta parte se recomienda consultar la documentación de la syscall [stat\(2\)](#).

## **Escritura de Bloque**

Esta operación recibirá el contenido de un bloque lógico de un File:Tag y guardará los cambios en el bloque físico correspondiente, siempre y cuando el File:Tag no se encuentre en estado COMMITED y el bloque lógico se encuentre asignado.

Si el bloque lógico a escribir fuera el único referenciando a su bloque físico asignado, se escribirá dicho bloque físico directamente. En caso contrario, se deberá buscar un nuevo bloque físico, escribir en el mismo y asignarlo al bloque lógico en cuestión.

## **Lectura de Bloque**

Dado un File:Tag y número de bloque lógico, la operación de lectura obtendrá y devolverá el contenido del mismo.

## **Eliminar un tag**

Esta operación eliminará el directorio correspondiente al File:Tag indicado. Al realizar esta operación, si el bloque físico al que apunta cada bloque lógico eliminado no es referenciado por ningún otro File:Tag, deberá ser marcado como libre en el bitmap<sup>10</sup>.

## **Errores en las operaciones**

A fin de unificar los posibles errores que se puedan dar a lo largo de la ejecución del módulo storage, vamos a detallar los mismos a continuación. Es posible que algunos errores no apliquen para ciertas operaciones.

Cualquier error que se de en la ejecución de una query, será motivo de finalización de la misma y se informará al Worker correspondiente el motivo del error.

### **File / Tag inexistente**

Una operación quiere realizar una acción sobre un File:Tag que no existe (salvo la operación de CREATE que crea un nuevo File:Tag).

Una operación quiere realizar una acción sobre un tag que no existe, salvo la operación de TAG que crea un nuevo Tag.

### **File / Tag preexistente**

La operación CREATE o la operación TAG intentan crear un un File:Tag que ya existen.

### **Espacio Insuficiente**

Al intentar asignar un nuevo bloque físico, no se encuentra ninguno disponible.

### **Escritura no permitida**

Una query intenta escribir o truncar un File:Tag que se encuentre en estado COMMITED.

---

<sup>10</sup> Para la implementación de esta parte se recomienda consultar la documentación de la syscall [stat\(2\)](#).

## Lectura o escritura fuera de límite

Una query intenta leer o escribir por fuera del tamaño del File:Tag.

## Retardos en las operaciones

Para todas las operaciones se deberá esperar un tiempo determinado por el valor del archivo de configuración RETARDO\_OPERACION.

Adicionalmente, por cada bloque que se quiera leer y/o escribir se deberá esperar el tiempo determinado por el valor del archivo de configuración RETARDO\_ACCESO\_BLOQUE.

## Logs mínimos y obligatorios

**Conexión de Worker:** “##Se conecta el Worker <WORKER\_ID> - Cantidad de Workers: <CANTIDAD>”

**Desconexión de Worker:** “##Se desconecta el Worker <WORKER\_ID> - Cantidad de Workers: <CANTIDAD>”

**File Creado:** “##<QUERY\_ID> - File Creado <NOMBRE\_FILE>:<TAG>”

**File Truncado:** “##<QUERY\_ID> - File Truncado <NOMBRE\_FILE>:<TAG> - Tamaño: <TAMAÑO>”

**Creación de Tag:** “##<QUERY\_ID> - Tag creado <NOMBRE\_FILE>:<TAG>”

**Commit de Tag:** “##<QUERY\_ID> - Commit de File:Tag <NOMBRE\_FILE>:<TAG>”

**Eliminación de Tag:** “##<QUERY\_ID> - Tag Eliminado <NOMBRE\_FILE>:<TAG>”

**Bloque Lógico Leído:** “##<QUERY\_ID> - Bloque Lógico Leído <NOMBRE\_FILE>:<TAG> - Número de Bloque: <BLOQUE>”

**Bloque Lógico Escrito:** “##<QUERY\_ID> - Bloque Lógico Escrito <NOMBRE\_FILE>:<TAG> - Número de Bloque: <BLOQUE>”

**Bloque Físico Reservado:** “##<QUERY\_ID> - Bloque Físico Reservado - Número de Bloque: <BLOQUE>”

**Bloque Físico Liberado:** “##<QUERY\_ID> - Bloque Físico Liberado - Número de Bloque: <BLOQUE>”

**Hard Link Agregado:** “##<QUERY\_ID> - <NOMBRE\_FILE>:<TAG> Se agregó el hard link del bloque lógico <BLOQUE\_LOGICO> al bloque físico <BLOQUE\_FISICO>”

**Hard Link Eliminado:** “##<QUERY\_ID> - <NOMBRE\_FILE>:<TAG> Se eliminó el hard link del bloque lógico <BLOQUE\_LOGICO> al bloque físico <BLOQUE\_FISICO>”

**Deduplicación de Bloque:** “##<QUERY\_ID> - <NOMBRE\_FILE>:<TAG> Bloque Lógico <BLOQUE> se reasigna de <BLOQUE\_FISICO\_ACTUAL> a <BLOQUE\_FISICO\_CONFIRMADO>”

## Archivo de Configuración

Campo	Tipo	Descripción
PUERTO_ESCUCHA	Numérico	Puerto al cual se deberán conectar los procesos worker
FRESH_START	Booleano	Declara si se debe realizar una limpieza e inicialización de cero del FS.
PUNTO_MONTAJE	String	PATH donde se encontrará el punto de montaje de nuestra Storage con todas sus estructuras adentro
RETARDO_OPERACION	Numérico	Tiempo en milisegundos que se deberá esperar ante cada petición.
RETARDO_ACCESO_BLOQUE	Numérico	Tiempo en milisegundos que se deberá esperar luego de cada acceso a bloque.
LOG_LEVEL	String	Nivel de detalle máximo a mostrar. Compatible con <a href="#">log_level_from_string()</a>

### Ejemplo de Archivo de Configuración

```
PUERTO_ESCUCHA=9002
FRESH_START=TRUE
PUNTO_MONTAJE=/home/utnso/storage
RETARDO_OPERACION=8000
RETARDO_ACCESO_BLOQUE=4000
LOG_LEVEL=INFO
```

## 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.

### Check de Control Obligatorio 1: Conexión inicial

**Fecha:** 13/09/2025

#### 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 y compartir información. Por ejemplo:
  - El path de un archivo de Query se envía desde el Query Control es capaz de ser recibido por el Master.
  - El Master es capaz de enviarle el path de la query al Worker.
  - El Worker es capaz de solicitarle a Storage el tamaño de bloque.

### Check de Control Obligatorio 2: Ejecución Básica

**Fecha:** 11/10/2025

#### Objetivos:

- **Módulo Query Control: Completo**
- **Módulo Master:**
  - Permitir la conexión de múltiples Workers y múltiples Query Control
  - Planificación bajo algoritmo FIFO
- **Módulo Worker:**
  - Lectura de archivos de Queries
  - Es capaz de interpretar y ejecutar todas las instrucciones
  - Creación de tablas de páginas
- **Módulo Storage:**
  - Creación de estructura de archivos y directorios bajo FRESH\_START
  - Todas las operaciones se responden con un valor fijo definido por el grupo<sup>11</sup>

#### Carga de trabajo estimada:

- **Módulo Query Control:** 10%
- **Módulo Master:** 35%
- **Módulo Worker:** 35%
- **Módulo Storage:** 20%

<sup>11</sup> Se recomienda investigar el concepto de [Mock Object](#)

## Check de Control Obligatorio 3: Comenzando a Persistir

**Fecha:** 08/11/2025

### Objetivos:

- **Módulo Master:**
  - Completo
- **Módulo Worker:**
  - Completo
- **Módulo Storage:**
  - Implementada la creación de File:Tag
  - Las operaciones de lectura y escritura devuelven valores reales y son persistidas en el FS.
  - Manejo de espacio libre sin deduplicación (Chequeo de MD5)

### Carga de trabajo estimada:

- **Módulo Master:** 20%
- **Módulo Worker:** 40%
- **Módulo Storage:** 40%

## Entregas Finales

**Fechas:** 29/11/2025, 06/12/2025, 20/12/2025

### 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.

# **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

<b>Índice</b>	<b>2</b>
<b>Historial de Cambios</b>	<b>4</b>
<b>Objetivos del Trabajo Práctico</b>	<b>1</b>
Características	1
Evaluación del Trabajo Práctico	1
Deployment y Testing del Trabajo Práctico	2
Aclaraciones	2
<b>Definición del Trabajo Práctico</b>	<b>3</b>
¿Qué es el trabajo práctico y cómo empezamos?	3
Arquitectura del sistema	4
Distribución Recomendada	4
Aclaración Importante	5
<b>Módulo: Consola</b>	<b>6</b>
Lineamiento e Implementación	6
Archivo de pseudocódigo	6
Ejemplo de líneas a parsear	7
Archivo de configuración	7
Ejemplo de Archivo de Configuración	7
<b>Módulo: Kernel</b>	<b>8</b>
Lineamiento e Implementación	8
Diagrama de estados	8
PCB	9
Planificador de Largo Plazo	9
Planificador de Corto Plazo	9
Manejo de Recursos	10
Manejo de Memoria	10
Compactación de Memoria	11
Manejo de File System	11
Logs mínimos y obligatorios	13
Archivo de configuración	14
Ejemplo de Archivo de Configuración	15
<b>Módulo: CPU</b>	<b>16</b>
Lineamiento e Implementación	16
Ciclo de Instrucción	17
Fetch	17
Decode	17
Execute	17
MMU	18
Logs mínimos y obligatorios	19
Archivo de configuración	19
Ejemplo de Archivo de Configuración	19
<b>Módulo: Memoria</b>	<b>20</b>
Lineamiento e Implementación	20

Estructuras	20
Esquema de memoria	20
Comunicación con Kernel, CPU y File System	20
Inicialización del proceso	20
Finalización de proceso	21
Acceso a espacio de usuario	21
Creación de Segmento	21
Eliminación de Segmento	21
Compactación de Segmentos	21
Logs mínimos y obligatorios	22
Archivo de configuración	22
Ejemplo de Archivo de Configuración	23
<b>Módulo: File System</b>	<b>24</b>
Lineamiento e Implementación	24
Comunicación con Kernel y Memoria	24
Abrir Archivo	24
Crear Archivo	24
Truncar Archivo	25
Leer Archivo	25
Escribir Archivo	25
Persistencia	25
Logs mínimos y obligatorios	26
Archivo de configuración	26
Ejemplo de Archivo de Configuración	27
<b>Descripción de las entregas</b>	<b>1</b>
Checkpoint 1: Conexión Inicial	1
Checkpoint 2: Avance del Grupo	1
Checkpoint 3: Obligatorio - Presencial	2
Checkpoint 4: Avance del Grupo	2
Checkpoint 5: Entregas Finales	3
<b>Anexo: Implementación de File System</b>	<b>1</b>
Estructuras	1
Superbloque	1
Ejemplo de Superbloque	1
Bitmap de Bloques	1
Archivo de Bloques	1
FCB	1
Ejemplo de FCB	2

## **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  $\pm 0$ ) 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](#).

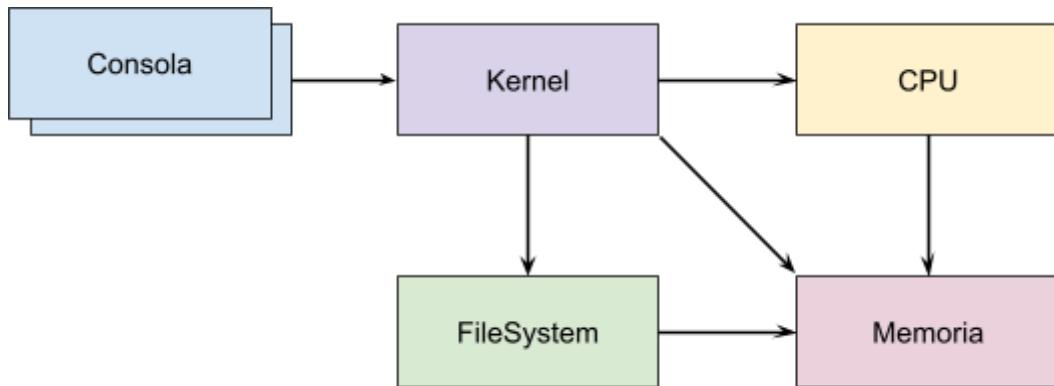
### ¿Qué es el trabajo práctico y cómo empezamos?

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*<sup>1</sup> 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%**

---

<sup>1</sup>Estos *procesos* son “símbólicos”, 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)<sup>2</sup>.

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).

---

<sup>2</sup> Se recomienda utilizar los parámetros argc y argv de la función main ayudándose con esta [guía](#).

### Ejemplo de líneas a parsear

```
1 SET AX HOLA
2 MOV_OUT 120 AX
3 WAIT DISCO
4 I/O 10
5 SIGNAL DISCO
6 MOV_IN BX 120
7 F_OPEN ARCHIVO
8 YIELD
9 F_TRUNCATE ARCHIVO 64
10 F_SEEK ARCHIVO 10
11 CREATE_SEGMENT 1 128
12 F_WRITE ARCHIVO 4 4
13 F_READ ARCHIVO 16 4
14 DELETE_SEGMENT 1
15 F_CLOSE ARCHIVO
16 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.

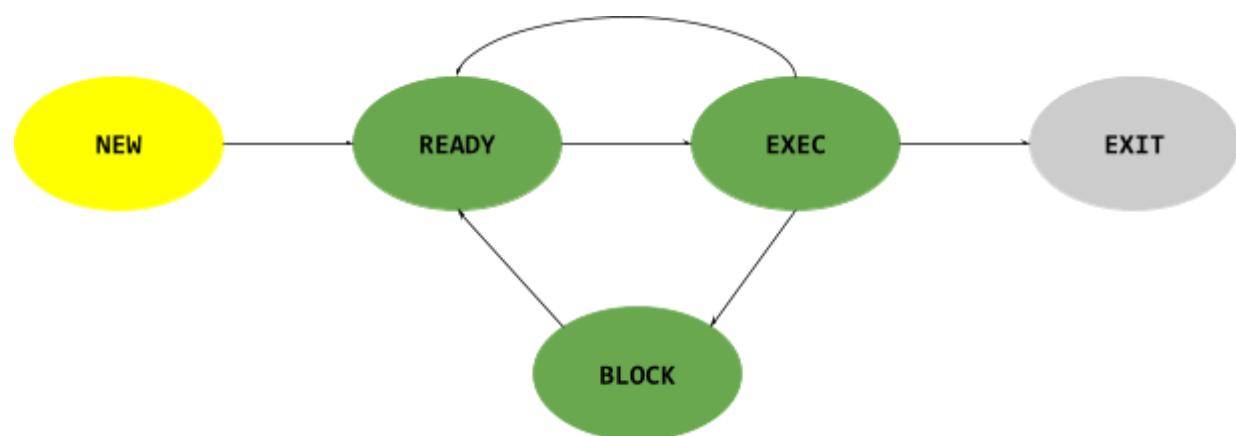
Cualquier fallo en las comunicaciones deberá ser informado mediante un mensaje apropiado en el archivo de log y el sistema deberá finalizar su ejecución.

## 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/O 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.<sup>3</sup>
- **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)

---

<sup>3</sup> 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<sup>4</sup> 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<sup>5</sup>.

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:

---

<sup>4</sup> No se tiene una implementación de privilegios como tal, pero se respeta lo visto en la teoría.

<sup>5</sup> 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 dispondrá 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_MEMORY=127.0.0.1
PUERTO_MEMORY=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<sup>6</sup>, 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<sup>7</sup> (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 TEXT0111  
SET AX UN06
```

---

<sup>6</sup> A diferencia de la realidad donde el Fetch busca la instrucción en memoria, en esta simplificación, buscaremos las instrucciones dentro del PCB.

<sup>7</sup> 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<sup>8</sup> 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.

---

<sup>8</sup> 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:

[Nº 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 memoria 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_MEMORY	String	IP a la cual se deberá conectar con la Memoria
PUERTO_MEMORY	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/thread>

## 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<sup>9</sup>: 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.

---

<sup>9</sup> 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.

# **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

<b>Índice</b>	<b>2</b>
<b>Historial de Cambios</b>	<b>4</b>
<b>Objetivos del Trabajo Práctico</b>	<b>5</b>
Características	5
Evaluación del Trabajo Práctico	5
Deployment y Testing del Trabajo Práctico	6
Aclaraciones	6
<b>Definición del Trabajo Práctico</b>	<b>7</b>
¿Qué es el trabajo práctico y cómo empezamos?	7
Arquitectura del sistema	8
Distribución Recomendada	8
Aclaración Importante	8
<b>Módulo: Kernel</b>	<b>9</b>
Lineamiento e Implementación	9
Diagrama de estados	9
PCB	10
Planificador de Largo Plazo	10
Planificador de Corto Plazo	10
Manejo de Recursos	11
Manejo de Memoria	11
Creación de Procesos	11
Eliminación de Procesos	11
Page Fault	12
Manejo de File System	12
Funciones por consola	13
Detección y resolución de Deadlocks	14
Logs mínimos y obligatorios	14
Archivo de configuración	16
Ejemplo de Archivo de Configuración	16
<b>Módulo: CPU</b>	<b>18</b>
Lineamiento e Implementación	18
Ciclo de Instrucción	18
Fetch	18
Decode	19
Ejemplos de instrucciones a interpretar	19
Execute	19
Check Interrupt	20
MMU	20
Page Fault	21
Logs mínimos y obligatorios	21
Archivo de configuración	21
Ejemplo de Archivo de Configuración	21
<b>Módulo: Memoria</b>	<b>23</b>

<b>Lineamiento e Implementación</b>	<b>23</b>
Memoria de Instrucciones	23
Esquema de memoria	23
Estructuras	24
Comunicación con Kernel, CPU y File System	24
Creación de proceso	24
Finalización de proceso	24
Acceso a tabla de páginas	24
Acceso a espacio de usuario	24
Page Fault	24
Logs mínimos y obligatorios	25
Archivo de configuración	25
Ejemplo de Archivo de Configuración	26
<b>Módulo: File System</b>	<b>27</b>
Lineamiento e Implementación	27
Estructuras	27
FCB	27
Ejemplo de FCB	27
File Allocation Table	27
Archivo de bloques	28
Comunicación con Kernel y Memoria	28
Peticiones del módulo Kernel	28
Abrir Archivo	28
Crear Archivo	28
Truncar Archivo	29
Leer Archivo	29
Escribir Archivo	29
Peticiones del módulo Memoria	29
Iniciar Proceso	29
Finalizar Proceso	29
Logs mínimos y obligatorios	30
Archivo de configuración	30
Ejemplo de Archivo de Configuración	31
<b>Descripción de las entregas</b>	<b>32</b>
Checkpoint 1: Conexión Inicial	32
Checkpoint 2: Avance del Grupo	32
Checkpoint 3: Obligatorio - Presencial	33
Checkpoint 4: Avance del Grupo	33
Checkpoint 5: Entregas Finales	34

## **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  $\pm 0$ ) 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](#).

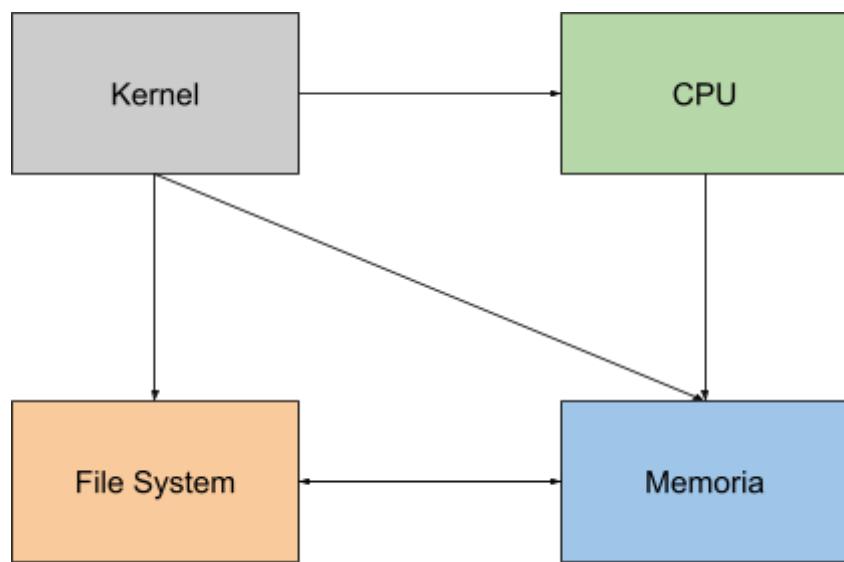
### ¿Qué es el trabajo práctico y cómo empezamos?

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 FileSystem**).*

*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:

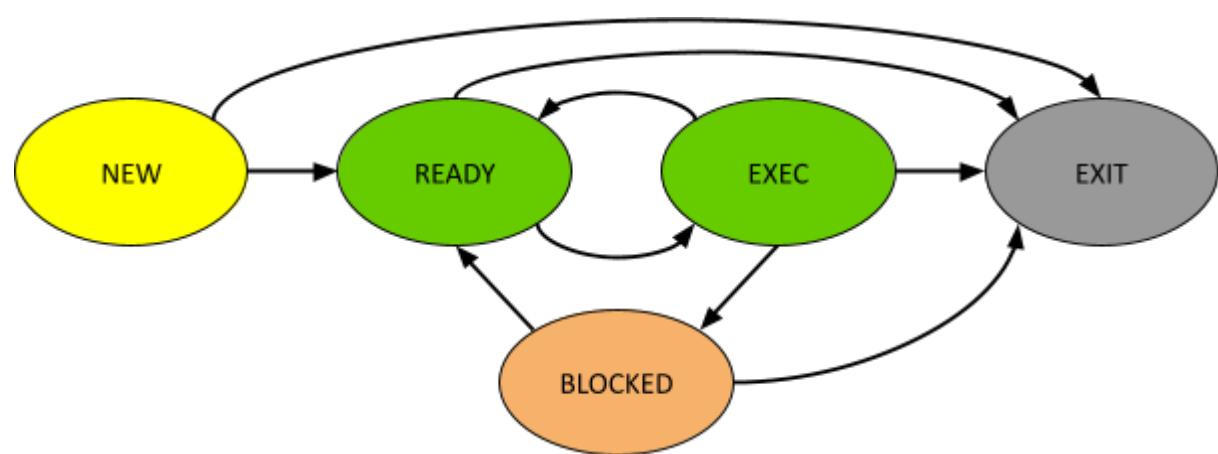
- Iniciar proceso
- Finalizar proceso
- Iniciar planificación
- Detener planificación
- Modificar grado multiprogramación
- Listar procesos por estado

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 <i>dispatch</i> al cual se deberá conectar con la CPU
PUERTO_CPU_INTERRUPT	Numérico	Puerto de <i>interrupt</i> 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*<sup>1</sup> 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.

---

<sup>1</sup> 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

```
1 SET AX 1
2 SET BX 1
3 SUM AX BX
4 SUB AX BX
5 MOV_IN DX 0
6 MOV_OUT 0 CX
7 SLEEP 10
8 JNZ AX 4
9 WAIT RECURSO_1
10 SIGNAL RECURSO_1
11 F_OPEN ARCHIVO W
12 F_TRUNCATE ARCHIVO 64
13 F_SEEK ARCHIVO 8
14 F_WRITE ARCHIVO 0
15 F_READ ARCHIVO 0
16 F_CLOSE ARCHIVO
17 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:

$$[\text{número\_pagina} \mid \text{desplazamiento}]$$

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:

$$\text{número\_página} = \text{floor}(\text{dirección\_lógica} / \text{tamaño\_página})$$

$$\text{desplazamiento} = \text{dirección_lógica} - \text{número_página} * \text{tamaño_página}$$

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<sup>2</sup>: 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 llamarse de igual manera que el nombre del archivo con la extensión fcb<sup>3</sup>, 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.

<sup>2</sup> 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.

<sup>3</sup> 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)`.

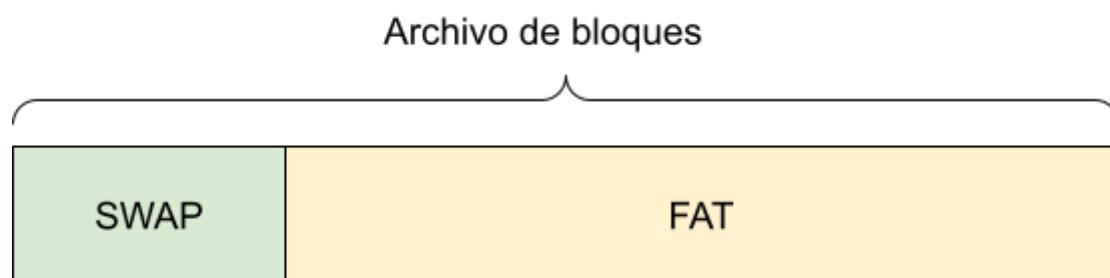
Si al iniciar el FS, no se encuentra el archivo de la tabla FAT, se deberá crear el mismo e inicializarlo con todos valores en 0 que representará que dicho bloque lógico está libre. Cabe resaltar que el bloque lógico 0 estará reservado<sup>4</sup> y no debe ser asignado a ningún archivo.

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<sup>5</sup>, 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.

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

<sup>5</sup> 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 <sup>6</sup>
RETARDO_ACCESO_BLOQUE	Numérico	Tiempo en milisegundos que se deberá esperar ante cada acceso a un bloque.

<sup>6</sup> 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_MEMORY=127.0.0.1
PUERTO_MEMORY=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 TPO: <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>

# **C Comenta**

***En los pasillos de la facu se comenta...***



*¿Esperaban que cambiemos algo?*

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

-1C2024 -  
Versión 1.3

# Índice

<b>Índice</b>	<b>2</b>
<b>Historial de Cambios</b>	<b>4</b>
<b>Objetivos del Trabajo Práctico</b>	<b>5</b>
Características	5
Evaluación del Trabajo Práctico	5
Deployment y Testing del Trabajo Práctico	6
Aclaraciones	6
<b>Definición del Trabajo Práctico</b>	<b>6</b>
¿Qué es el trabajo práctico y cómo empezamos?	7
Arquitectura del sistema	8
Distribución Recomendada	8
Aclaración Importante	8
<b>Módulo: Kernel</b>	<b>9</b>
Lineamiento e Implementación	9
Diagrama de estados	9
PCB	10
Planificador de Largo Plazo	10
Creación de Procesos	10
Eliminación de Procesos	10
Planificador de Corto Plazo	10
Manejo de Recursos	11
Manejo de Interfaces de I/O	11
Comandos por consola	12
Logs mínimos y obligatorios	13
Archivo de configuración	13
Ejemplo de Archivo de Configuración	14
<b>Módulo: CPU</b>	<b>15</b>
Lineamiento e Implementación	15
Registros de la CPU	15
Ciclo de Instrucción	16
Fetch	16
Decode	16
Ejemplos de instrucciones a interpretar	16
Execute	17
Check Interrupt	18
MMU	19
TLB	19
Logs mínimos y obligatorios	20
Archivo de configuración	20
Ejemplo de Archivo de Configuración	20

<b>Módulo: Memoria</b>	<b>22</b>
Lineamiento e Implementación	22
Memoria de Instrucciones	22
Esquema de memoria y Estructuras	22
Comunicación con Kernel, CPU e Interfaces de I/O	22
Creación de proceso	22
Finalización de proceso	22
Acceso a tabla de páginas	22
Ajustar tamaño de un proceso	23
Ampliación de un proceso	23
Reducción de un proceso	23
Acceso a espacio de usuario	23
Retardo en peticiones	23
Logs mínimos y obligatorios	23
Archivo de configuración	24
Ejemplo de Archivo de Configuración	24
<b>Módulo: Interfaz de I/O</b>	<b>25</b>
Lineamiento e Implementación	25
Interfaces Genéricas	25
Interfaces STDIN	25
Interfaces STDOUT	26
Interfaces DialFS	26
Sistema de archivos DialFS	26
Creación de archivos	27
Compactación	27
Logs mínimos y obligatorios	27
Archivo de configuración	28
Ejemplo de Archivo de Configuración	29
<b>Descripción de las entregas</b>	<b>30</b>
Check de Control Obligatorio 1: Conexión inicial	30
Check de Control Obligatorio 2: Planificación CP y Operaciones aritméticas	30
Check de Control Obligatorio 3: Memoria e interfaces de memoria	30
Entregas Finales	31

## **Historial de Cambios**

v1.0 (30/03/2024) *Release inicial del trabajo práctico*

v1.1 (18/04/2024)

- Agregada aclaración sobre manejo de interfaces I/O en Kernel
- Agregada nota al pie sobre connotación teórica de la instrucción RESIZE
- Agregados detalles sobre la implementación de Compactación (DialFS)

v1.2 (02/05/2024)

- Agregada más información sobre el funcionamiento de las Interfaces de I/O

v1.3 (02/06/2024)

- Agregadas aclaraciones sobre traducción de direcciones de memoria
- Agregadas aclaraciones sobre retardos en peticiones a memoria
- Arreglados errores de redacción en Módulo Interfaz de I/O
- Eliminado retardo innecesario en interfaces STDOUT

# 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  $\pm 0$ ) y obligatorio
- Fecha de comienzo: 30/03/2024
- Fecha de primera entrega: 13/07/2024
- Fecha de segunda entrega: 27/07/2024
- Fecha de tercera entrega: 03/08/2024
- 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](#).

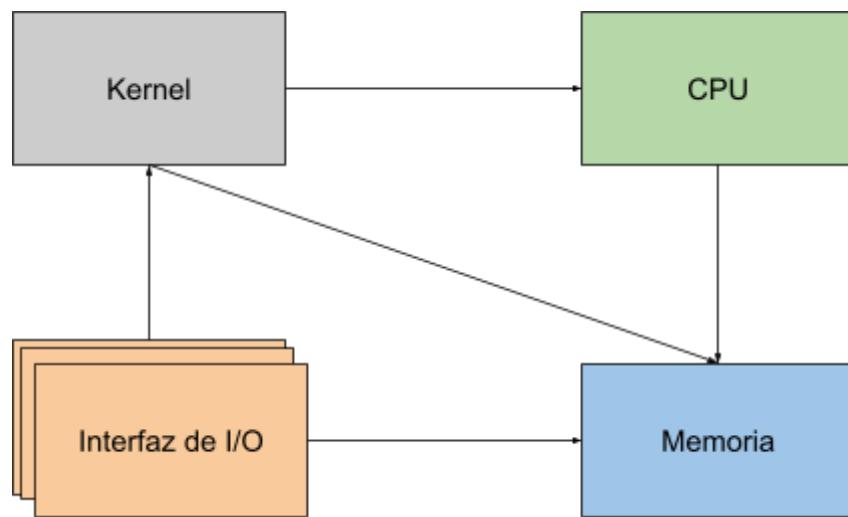
## **¿Qué es el trabajo práctico y cómo empezamos?**

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: **35%**
- CPU: **15%**
- Memoria: **20%**
- Interfaces de I/O: **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 Interfaces de I/O**).*

*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:

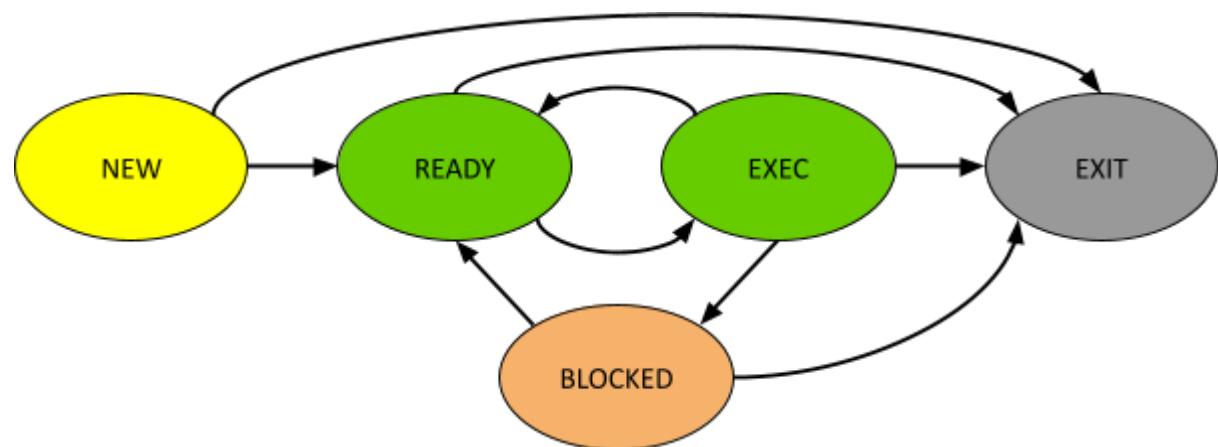
- Ejecutar Script de Operaciones
- Iniciar proceso
- Finalizar proceso
- Iniciar planificación
- Detener planificación
- Listar procesos por estado

Además de las operaciones de la consola interactiva, el Kernel será el encargado de gestionar las peticiones contra la Memoria y las interfaces de I/O (conectadas dinámicamente), por lo que deberá implementarse siguiendo una estrategia multihilo que permita la concurrencia de varias solicitudes desde o hacia diferentes módulos.

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 un 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 I/O, teniendo una por cada interfaz de I/O conectada y correspondientes a los recursos que administra el Kernel, teniendo una por cada recurso.



## 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.
- **Quantum:** Unidad de tiempo utilizada por el algoritmo de planificación VRR.
- **Registros de la CPU:** Estructura que contendrá los valores de los *registros de uso general* de la CPU.

## Planificador de Largo Plazo

El Kernel será el encargado de gestionar las peticiones a la memoria para la creación y eliminación de procesos. 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 cuyas operaciones corresponderán al archivo de pseudocódigo pasado por parámetro, todos los procesos iniciarán sin espacio reservado en memoria, por lo que solamente tendrán una tabla de páginas vacía.

En caso de que el grado de multiprogramación lo permita, los procesos creados podrán pasar de la cola de NEW a la cola de READY, caso contrario, se quedarán a la espera de que finalicen procesos que se encuentran en ejecución.

### Eliminación de Procesos

Ante la llegada de un proceso al estado de EXIT (ya sea por solicitud de la CPU, por un error, 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.

Al eliminar un proceso se deberá habilitar 1 espacio en el grado de multiprogramación, para que en caso de haber procesos encolados en NEW, el primero de estos pase a READY.

## Planificador de Corto Plazo

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

- **FIFO**
- **Round Robin**
- **Virtual Round Robin**

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 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 que el recurso exista, se deberá enviar el proceso a EXIT.

## Manejo de Interfaces de I/O

Todas las interfaces de I/O tendrán un único nombre que las identificará en el sistema, este nombre deberá ser consultado al momento de que una Interfaz se conecte al kernel. Como este nombré será único se utilizará en los programas para identificar a la Interfaz.

Al recibir una petición de I/O de parte de la CPU primero se deberá validar que exista y esté conectada la interfaz solicitada, en caso contrario, se deberá enviar el proceso a EXIT.

En caso de que la interfaz exista y esté conectada, se deberá validar que la interfaz admite la operación solicitada, en caso de que no sea así, se deberá enviar el proceso a EXIT.

De cumplirse todos los requisitos anteriores, el Kernel enviará el proceso al estado BLOCKED y a partir de este punto pueden darse 2 situaciones:

1. El caso en el que la interfaz de I/O esté libre: En este caso el Kernel deberá solicitar la operación al dispositivo correspondiente.
2. En el caso de que exista algún proceso haciendo uso de la Interfaz de I/O, el proceso que acaba de solicitar la operación de I/O deberá esperar la finalización del anterior antes de poder hacer uso de la misma.

Una vez la operación finalice, el Kernel recibirá una notificación y desbloqueará dicho proceso para que esté listo para continuar con su ejecución cuando le toque según el algoritmo.

Toda interfaz de I/O puede conectarse y desconectarse en tiempo de ejecución. No se evaluará el caso en que una interfaz se desconecte estando ocupada por un proceso.

## Comandos por consola

El Kernel dispondrá de una consola donde se permitirá la ejecución de los diferentes comandos que se especifican a continuación:

- Ejecutar Script de Comandos: Se encargará de abrir un archivo de comandos que se encontrará en la máquina donde corra el Kernel, el mismo contendrá una secuencia de comandos y se los deberá ejecutar uno a uno hasta finalizar el archivo.  
Nomenclatura: **EJECUTAR\_SCRIPT [PATH]**
- Iniciar proceso: Se encargará de iniciar un nuevo proceso cuyas instrucciones a ejecutar se encontrarán en el archivo <path> dentro del file system de la máquina donde corra la Memoria. Este mensaje se encargará de la creación del proceso (PCB) en estado NEW.  
Nomenclatura: **INICIAR\_PROCESO [PATH]**
- 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 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 el grado de multiprogramación del módulo: Se cambiará el grado de multiprogramación del sistema reemplazandolo por el valor indicado.  
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.

En caso de que se tengan más procesos ejecutando que lo que permite el grado de multiprogramación, no se tomarán acciones sobre los mismos y se esperará su finalización normal.

## 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\_INTERFACE / OUT\_OF\_MEMORY / INTERRUPTED\_BY\_USER>”

**Cambio de Estado:** “PID: <PID> - Estado Anterior: <ESTADO\_ANTERIOR> - Estado Actual: <ESTADO\_ACTUAL>”

**Motivo de Bloqueo:** “PID: <PID> - Bloqueado por: <INTERFAZ / NOMBRE\_RECURSO>”

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

**Ingreso a Ready:** “Cola Ready / Ready Prioridad: [<LISTA DE PIDS>]”

## Archivo de configuración

Campo	Tipo	Descripción
PUERTO_ESCUCHA	Numérico	Puerto en el cual se escucharán las conexiones a este módulo.
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_CPU	String	IP a la cual se deberá conectar con la CPU
PUERTO_CPU_DISPATCH	Numérico	Puerto de <i>dispatch</i> al cual se deberá conectar con la CPU
PUERTO_CPU_INTERRUPT	Numérico	Puerto de <i>interrupt</i> al cual se deberá conectar con la CPU

Campo	Tipo	Descripción
ALGORITMO_PLANIFICACION	String	Define el algoritmo de planificación de corto plazo. (FIFO / RR / VRR)
QUANTUM	Numérico	Tiempo en milisegundos del quantum para utilizar bajo el algoritmo RR o VRR
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	Numérico	Grado de multiprogramación del módulo

### Ejemplo de Archivo de Configuración

```

PUERTO_ESCUCHA=8003
IP_MEMORIA=127.0.0.1
PUERTO_MEMORIA=8002
IP_CPU=127.0.0.1
PUERTO_CPU_DISPATCH=8006
PUERTO_CPU_INTERRUPT=8007
ALGORITMO_PLANIFICACION=VRR
QUANTUM=2000
RECURSOS=[RA, RB, RC]
INSTANCIAS_RECURSOS=[1, 2, 1]
GRADO_MULTIPROGRAMACION=10

```

# Módulo: CPU

El módulo CPU en nuestro contexto de TP lo que va a hacer es simular los pasos del ciclo de instrucción de una CPU real, de una forma mucho más simplificada.

## Lineamiento e Implementación

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 a alguna interfaz de I/O que interactúe 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), solicitar una llamada al Kernel, o deber ser desalojado (**interrupción**).

## Registros de la CPU

En la implementación de nuestra CPU, vamos a utilizar una serie de registros para poder modelar la operatoria de una CPU real, es decir, vamos a contar con registros similares a los vistos en Arquitectura de Computadores y algunos registros creados por nosotros mismos a fin de poder facilitar las pruebas.

En la siguiente tabla está el detalle de los registros que deberá tener nuestra CPU, es decir, estará detallado el tamaño del mismo y que tipo de dato se recomienda para su implementación:

Registro	Tamaño	Tipo de Dato	Descripción
PC	4 bytes	uint32_t	Program Counter, indica la próxima instrucción a ejecutar
AX	1 byte	uint8_t	Registro Numérico de propósito general
BX	1 byte	uint8_t	Registro Numérico de propósito general
CX	1 byte	uint8_t	Registro Numérico de propósito general
DX	1 byte	uint8_t	Registro Numérico de propósito general
EAX	4 bytes	uint32_t	Registro Numérico de propósito general
EBX	4 bytes	uint32_t	Registro Numérico de propósito general
ECX	4 bytes	uint32_t	Registro Numérico de propósito general
EDX	4 bytes	uint32_t	Registro Numérico de propósito general

SI	4 bytes	uint32_t	Contiene la dirección lógica de memoria de origen desde donde se va a copiar un string.
DI	4 bytes	uint32_t	Contiene la dirección lógica de memoria de destino a donde se va a copiar un string.

## 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.

### 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

**El siguiente ejemplo es solo de las instrucciones, no representa un programa funcional.**

```

1  SET AX 1
2  SET BX 1
3  SET PC 5
4  SUM AX BX
5  SUB AX BX
6  MOV_IN EDX ECX
7  MOV_OUT EDX ECX
8  RESIZE 128
9  JNZ AX 4
10 COPY_STRING 8
11 IO_GEN_SLEEP Int1 10
12 IO_STDIN_READ Int2 EAX AX
13 IO_STDOUT_WRITE Int3 BX EAX
14 IO_FS_CREATE Int4 notas.txt
15 IO_FS_DELETE Int4 notas.txt
16 IO_FS_TRUNCATE Int4 notas.txt ECX
17 IO_FS_WRITE Int4 notas.txt AX ECX EDX
18 IO_FS_READ Int4 notas.txt BX ECX EDX
19 WAIT RECURSO_1
20 SIGNAL RECURSO_1

```

Las instrucciones detalladas previamente son a modo de ejemplo, su ejecución no necesariamente sigue alguna lógica ni funcionamiento correcto. Al momento de realizar las pruebas, 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.
- **MOV\_IN** (Registro Datos, Registro Dirección): Lee el valor de memoria correspondiente a la Dirección Lógica que se encuentra en el Registro Dirección y lo almacena en el Registro Datos.
- **MOV\_OUT** (Registro Dirección, Registro Datos): Lee el valor del Registro Datos y lo escribe en la dirección física de memoria obtenida a partir de la Dirección Lógica almacenada en el Registro Dirección.
- **SUM** (Registro Destino, Registro Origen): Suma al Registro Destino el Registro Origen 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.
- **RESIZE** (Tamaño): Solicitará a la Memoria ajustar el tamaño del proceso al *tamaño* pasado por parámetro. En caso de que la respuesta de la memoria sea *Out of Memory*, se deberá devolver el **contexto de ejecución** al Kernel informando de esta situación.<sup>1</sup>
- **COPY\_STRING** (Tamaño): Toma del string apuntado por el registro SI y copia la cantidad de bytes indicadas en el parámetro *tamaño* a la posición de memoria apuntada por el registro DI.
- **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.
- **IO\_GEN\_SLEEP** (Interfaz, Unidades de trabajo): Esta instrucción solicita al Kernel que se envíe a una interfaz de I/O a que realice un sleep por una cantidad de *unidades de trabajo*.
- **IO\_STDIN\_READ** (Interfaz, Registro Dirección, Registro Tamaño): Esta instrucción solicita al Kernel que mediante la interfaz ingresada se lea desde el STDIN (Teclado) un valor cuyo *tamaño* está delimitado por el valor del *Registro Tamaño* y el mismo se guarde a partir de la *Dirección Lógica almacenada en el Registro Dirección*.
- **IO\_STDOUT\_WRITE** (Interfaz, Registro Dirección, Registro Tamaño): Esta instrucción solicita al Kernel que mediante la interfaz seleccionada, se lea desde la posición de memoria

---

<sup>1</sup> En un sistema real, esto normalmente sería una “syscall”, pero en este trabajo lo realizamos de forma directa para que solamente el módulo Memoria intervenga en la implementación.

indicada por la *Dirección Lógica almacenada en el Registro Dirección*, un tamaño indicado por el *Registro Tamaño* y se imprima por pantalla.

- **IO\_FS\_CREATE** (Interfaz, Nombre Archivo): Esta instrucción solicita al Kernel que mediante la interfaz seleccionada, se cree un archivo en el FS montado en dicha interfaz.
- **IO\_FS\_DELETE** (Interfaz, Nombre Archivo): Esta instrucción solicita al Kernel que mediante la interfaz seleccionada, se elimine un archivo en el FS montado en dicha interfaz
- **IO\_FS\_TRUNCATE** (Interfaz, Nombre Archivo, Registro Tamaño): Esta instrucción solicita al Kernel que mediante la interfaz seleccionada, se modifique el tamaño del archivo en el FS montado en dicha interfaz, actualizando al valor que se encuentra en el registro indicado por *Registro Tamaño*.
- **IO\_FS\_WRITE** (Interfaz, Nombre Archivo, Registro Dirección, Registro Tamaño, Registro Puntero Archivo): Esta instrucción solicita al Kernel que mediante la interfaz seleccionada, se lea desde Memoria la cantidad de bytes indicadas por el *Registro Tamaño* a partir de la dirección lógica que se encuentra en el *Registro Dirección* y se escriban en el archivo a partir del valor del Registro Puntero Archivo.
- **IO\_FS\_READ** (Interfaz, Nombre Archivo, Registro Dirección, Registro Tamaño, Registro Puntero Archivo): Esta instrucción solicita al Kernel que mediante la interfaz seleccionada, se lea desde el archivo a partir del valor del Registro Puntero Archivo la cantidad de bytes indicada por *Registro Tamaño* y se escriban en la Memoria a partir de la dirección lógica indicada en el *Registro Dirección*.
- **EXIT**: Esta instrucción representa la syscall de finalización del proceso. Se deberá devolver el **Contexto de Ejecución** actualizado al Kernel para su finalización.

Es importante tener en cuenta las siguientes aclaraciones:

- Una dirección lógica se traduce a una dirección física, pero al copiar un string/registro a memoria, podría estar presente en más de una página (ver sección de MMU).
- Los registros utilizados en las operaciones tales como Registro Dirección, Registro Tamaño, Registro Puntero Archivo, etc siempre tendrán un valor previamente asignado con la instrucción **SET**.

### 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:

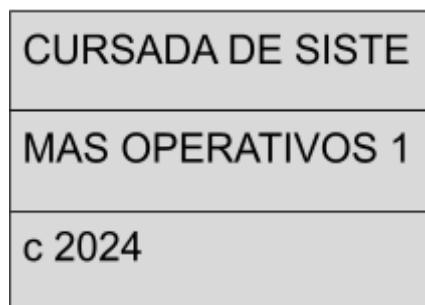
[número\_pagina | desplazamiento]

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

Es importante tener en cuenta en este punto que una Dirección Lógica va a pertenecer a una página en cuestión, pero el contenido a leer o escribir puede ser que ocupe más de una página, por ejemplo, supongamos que tenemos páginas de 16 bytes cada una y que quedemos escribir el texto: "Cursada de Sistemas Operativos 1c2024", esto si lo dividimos nos va a quedar ocupando 3 páginas, de manera que lo podemos representar fácilmente de la siguiente manera:



Es trabajo del grupo contemplar y manejar estos casos correctamente.

### TLB

Como las tablas de páginas están presentes en el módulo Memoria, se implementará una TLB para agilizar la traducción de las direcciones lógicas a direcciones físicas, para esto la TLB contará con la siguiente estructura [ pid | página | marco ], pudiendo agregar algún campo extra para facilitar la implementación de los algoritmos.

La cantidad de entradas y el algoritmo de reemplazo de la TLB se indicarán por archivo de configuración de la CPU. La cantidad de entradas de la TLB será un entero (pudiendo ser 0, lo cual la deshabilitará), mientras que los algoritmos podrán ser:

- FIFO
- LRU

Al momento de obtener el número de página se deberá consultar en la TLB si se tiene la información de la misma. En caso afirmativo (TLB Hit) se deberá devolver la dirección física (o frame) correspondiente, en caso contrario, se deberá informar el TLB Miss y se deberá consultar a la memoria para obtener el frame correspondiente a la página buscada. Por último, se agregará la nueva entrada a la TLB, si la misma se encuentra llena se deberá reemplazar siguiendo el algoritmo configurado pudiendo elegir como víctima cualquier otra entrada (incluso de otro proceso).

### Logs mínimos y obligatorios

**Fetch Instrucción:** "PID: <PID> - FETCH - Program Counter: <PROGRAM\_COUNTER>".

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

**TLB Hit:** "PID: <PID> - TLB HIT - Página: <NUMERO\_PAGINA>"

**TLB Miss:** "PID: <PID> - TLB MISS - Página: <NUMERO\_PAGINA>"

**Obtener Marco:** "PID: <PID> - OBTENER MARCO - Página: <NUMERO\_PAGINA> - Marco: <NUMERO\_MARCO>".

**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
CANTIDAD_ENTRADAS_TLB	Numérico	Cantidad de entradas que tendrá la TLB
ALGORITMO_TLB	String	Algoritmo de reemplazo de la TLB FIFO / LRU

## Ejemplo de Archivo de Configuración

```
IP_MEMORIA=127.0.0.1
PUERTO_MEMORIA=8002
PUERTO_ESCUCHA_DISPATCH=8006
PUERTO_ESCUCHA_INTERRUPT=8007
CANTIDAD_ENTRADAS_TLB=32
ALGORITMO_TLB=FIFO
```

# Módulo: Memoria

## Lineamiento e Implementación

### 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 y Estructuras

La memoria al trabajar bajo un esquema de **paginación simple** estará compuesta principalmente por 2 estructuras principales 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 e Interfaces de I/O

### Creación de proceso

Esta petición podrá venir solamente desde el módulo Kernel, y el módulo Memoria deberá crear las estructuras administrativas necesarias.

### Finalización de proceso

Esta petición podrá venir solamente desde el módulo Kernel. El módulo Memoria, al ser finalizado un proceso, debe liberar su espacio de memoria (marcando los frames como libres pero **sin sobreescribir su contenido**).

### Acceso a tabla de páginas

El módulo deberá responder el número de marco correspondiente a la página consultada.

## Ajustar tamaño de un proceso

Al llegar una solicitud de ajuste de tamaño de proceso (resize) se deberá cambiar el tamaño del proceso de acuerdo al nuevo tamaño. Se pueden dar 2 opciones:

Ampliación de un proceso

Se deberá ampliar el tamaño del proceso al final del mismo, pudiendo solicitarse múltiples páginas. Es posible que en un punto no se puedan solicitar más marcos ya que la memoria se encuentra llena, por lo que en ese caso se deberá contestar con un error de **Out Of Memory**.

Reducción de un proceso

Se reducirá el mismo desde el final, liberando, en caso de ser necesario, las páginas que ya no sean utilizadas (desde la última hacia la primera).

## Acceso a espacio de usuario

Esta petición puede venir tanto de la CPU como de un Módulo de Interfaz de I/O, es importante tener en cuenta que las peticiones pueden ocupar más de una página.

El módulo Memoria 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'.

## Retardo en peticiones

Ante cada una de las peticiones definidas anteriormente se deberá aplicar el tiempo de espera en milisegundos definido por archivo de configuración.

## 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>"

**Ampliación de Proceso:** "PID: <PID> - Tamaño Actual: <TAMAÑO\_ACTUAL> - Tamaño a Ampliar: <TAMAÑO\_A\_AMPLIAR>"

**Reducción de Proceso:** "PID: <PID> - Tamaño Actual: <TAMAÑO\_ACTUAL> - Tamaño a Reducir: <TAMAÑO\_A\_REDUCIR>"

**Acceso a espacio de usuario:** "PID: <PID> - Accion: <LEER / ESCRIBIR> - Direccion fisica: <DIRECCION\_FISICA>" - Tamaño <TAMAÑO A LEER / ESCRIBIR>

## 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_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.

### Ejemplo de Archivo de Configuración

```
PUERTO_ESCUCHA=8002
TAM_MEMORIA=4096
TAM_PAGINA=32
PATH_INSTRUCCIONES=/home/utnso/scripts-pruebas
RETARDO_RESPUESTA=1000
```

# Módulo: Interfaz de I/O

Las interfaces de I/O pueden ser varias, en la realidad las conocemos como Teclados, Mouse, Discos, Monitores o hasta Impresoras. Las mismas irán recibiendo desde Kernel distintas operaciones a realizar para determinado proceso, las atenderá una a la vez siguiendo el orden de llegada y le irán dando aviso a dicho módulo una vez se vayan completando.

## Lineamiento e Implementación

Cada interfaz en la vida real tiene diferentes velocidades, por lo que para simplificar esto en nuestro TP vamos a tener en la configuración del módulo el Tiempo de unidad de trabajo, este valor luego se va a multiplicar por otro valor que va a estar dado según el tipo de interfaz que tengamos, en este TP vamos a trabajar con 4 tipos de Interfaces: Genéricas, STDIN, STDOUT y DialFS.

Al iniciar una Interfaz de I/O la misma deberá recibir 2 parámetros:

- Nombre: Este nombre será único dentro del sistema y servirá como identificación de la Interfaz dentro del TP.
- Archivo de Configuración

## Interfaces Genéricas

Las interfaces genéricas van a ser las más simples, y lo único que van a hacer es que ante una petición van a esperar una cantidad de unidades de trabajo, cuyo valor va a venir dado en la petición desde el Kernel.

Las instrucciones que aceptan estas interfaces son:

- IO\_GEN\_SLEEP

Al leer el archivo de configuración solo le van a importar las propiedades de:

- TIPO\_INTERFAZ
- TIEMPO\_UNIDAD\_TRABAJO
- IP\_KERNEL
- PUERTO\_KERNEL

## Interfaces STDIN

Las interfaces STDIN no van a esperar ninguna unidad de trabajo, ya que las mismas van a quedarse esperando que el alumno ingrese un texto por teclado. Este texto se va a guardar en la memoria a partir de la o las direcciones físicas indicadas en la petición que recibió por parte del Kernel.

Las instrucciones que aceptan estas interfaces son:

- IO\_STDIN\_READ

Al leer el archivo de configuración solo le van a importar las propiedades de:

- TIPO\_INTERFAZ
- IP\_KERNEL
- PUERTO\_KERNEL
- IP\_MEMORIA
- PUERTO\_MEMORIA

## Interfaces STDOUT

Las interfaces STDOUT se conectan a memoria para leer el valor que se encuentra en la o las direcciones físicas pedidas y mostrar el resultado por pantalla.

La instrucción para realizar esto es:

- IO\_STDOUT\_WRITE

Al leer el archivo de configuración solo le van a importar las propiedades de:

- TIPO\_INTERFAZ
- IP\_KERNEL
- PUERTO\_KERNEL
- IP\_MEMORIA
- PUERTO\_MEMORIA

## Interfaces DialFS

Las interfaces DialFS son las más complejas de este trabajo práctico ya que las mismas interactúan con un sistema de archivos (filesystem) implementado por el grupo. Siempre va a consumir una unidad de TIEMPO\_UNIDAD\_TRABAJO.

En el caso de todas las peticiones que interactúen con el DialFS, la información a leer y/o escribir va a estar relacionada con una o más direcciones físicas de la memoria, por lo que el Módulo de Interfaz de I/O se va a tener que conectar con la memoria para pedirle o enviarle información.

Las instrucciones que aceptan estas interfaces son:

- IO\_FS\_CREATE
- IO\_FS\_DELETE
- IO\_FS\_TRUNCATE
- IO\_FS\_WRITE
- IO\_FS\_READ

## Sistema de archivos DialFS

Lo que busca este FS es ser una implementación simple que represente Asignación Contigua de bloques.

Para simplificar las estructuras que va a tener nuestro FS, vamos a contar con una serie de archivos para simular su funcionamiento.

El primero va a ser nuestro archivo de bloques (*bloques.dat*) el cual va a ser un archivo de tamaño definido mediante 2 parámetros del archivo de configuración: BLOCK\_SIZE y BLOCK\_COUNT, y el tamaño del archivo va a ser BLOCK\_SIZE \* BLOCK\_COUNT.

El segundo archivo va a ser un archivo que va a contener un bitmap<sup>2</sup> (*bitmap.dat*) indicando que bloques se encuentran libres y que bloques se encuentran ocupados dentro de nuestro FS, cualquier implementación que no utilice un bitmap será considerada una implementación equivocada y por consiguiente no se aprobará esa entrega del TP.

El tercer tipo de archivo va a ser un archivo de metadata<sup>3</sup>, del cual vamos a tener varios en nuestro FS y va a ser un archivo cuyo nombre va a ser el nombre del archivo en el FS, por ej *notas.txt* y su contenido va a tener el bloque en el cual empieza el archivo y el tamaño del archivo **en bytes**:

**BLOQUE\_INICIAL=25**  
**TAMANIO\_ARCHIVO=1024**

## Creación de archivos

Al momento de crearse un archivo, va a comenzar ocupando un bloque del FS aunque su tamaño sea 0 y luego el mismo se podrá extender o disminuir por medio de la función IO\_FS\_TRUNCATE.

## Compactación

Puede darse la situación que al momento de querer ampliar un archivo, dispongamos del espacio disponible pero el mismo no se encuentre contiguo, por lo que vamos a tener que compactar nuestro FS para agrupar los bloques de los archivos de manera tal que quede todo el espacio libre contiguo para el archivo que se desea truncar. Luego de compactar el FS, se deberá esperar un tiempo determinado por el valor de configuración de RETRASO\_COMPACTACION para luego continuar con la operación de ampliación del archivo.

Al leer el archivo de configuración solo le van a importar las propiedades de:

- TIPO\_INTERFAZ
- TIEMPO\_UNIDAD\_TRABAJO
- IP\_KERNEL
- PUERTO\_KERNEL
- IP\_MEMORIA
- PUERTO\_MEMORIA
- PATH\_BASE\_DIALFS
- BLOCK\_SIZE
- BLOCK\_COUNT
- RETRASO\_COMPACTACION

## Logs mínimos y obligatorios

**Todos - Operación:** “PID: <PID> - Operacion: <OPERACION\_A\_REALIZAR>”

---

<sup>2</sup> Se recomienda investigar la implementación de bitarray de las commons.

<sup>3</sup> Estos archivos de metadata tienen una similitud con los archivos de configuración de los módulos, por lo que recomendamos utilizar las commons propias de config para operarlos.

**DialFS - Crear Archivo:** "PID: <PID> - Crear Archivo: <NOMBRE\_ARCHIVO>"

**DialFS - Eliminar Archivo:** "PID: <PID> - Eliminar Archivo: <NOMBRE\_ARCHIVO>"

**DialFS - Truncar Archivo:** "PID: <PID> - Truncar Archivo: <NOMBRE\_ARCHIVO> - Tamaño: <TAMAÑO>"

**DialFS - Leer Archivo:** "PID: <PID> - Leer Archivo: <NOMBRE\_ARCHIVO> - Tamaño a Leer: <TAMAÑO> - Puntero Archivo: <PUNTERO\_ARCHIVO>"

**DialFS - Escribir Archivo:** "PID: <PID> - Escribir Archivo: <NOMBRE\_ARCHIVO> - Tamaño a Escribir: <TAMAÑO> - Puntero Archivo: <PUNTERO\_ARCHIVO>"

**DialFS - Inicio Compactación:** "PID: <PID> - Inicio Compactación."

**DialFS - Fin Compactación:** "PID: <PID> - Fin Compactación."

## Archivo de configuración

Campo	Tipo	Descripción
TIPO_INTERFAZ	String	Indica el tipo de Interfaz de I/O que estamos creando. GENERICA / STDIN / STDOUT / DIALFS
TIEMPO_UNIDAD_TRABAJO	Numérico	Tiempo en milisegundos que dura cada unidad de trabajo
IP_KERNEL	String	IP a la cual se deberá conectar con el Kernel
PUERTO_KERNEL	Numérico	Puerto al cual se deberá conectar con el Kernel
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
PATH_BASE_DIALFS	String	Path a partir del cual van a encontrarse los archivos de DialFS.
BLOCK_SIZE	Numérico	Tamaño de los bloques del FS
BLOCK_COUNT	Numérico	Cantidad de bloques del FS
RETRASO_COMPACTACION	Numérico	Tiempo en milisegundos que se espera a que finalice la compactación

## Ejemplo de Archivo de Configuración

```
TIPO_INTERFAZ=STDOUT
TIEMPO_UNIDAD_TRABAJO=250
IP_KERNEL=127.0.0.1
PUERTO_KERNEL=8003
IP_MEMORIA=127.0.0.1
PUERTO_MEMORIA=8002
PATH_BASE_DIALFS=/home/utnso/dialfs
BLOCK_SIZE=64
BLOCK_COUNT=1024
RETRASO_COMPACTACION=50000
```

## 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.

### Check de Control Obligatorio 1: Conexión inicial

**Fecha:** 20/04/2024

**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.

### Check de Control Obligatorio 2: Planificación CP y Operaciones aritméticas

**Fecha:** 25/05/2024

**Objetivos:**

- **Módulo Kernel:**
  - Es capaz de crear un PCB y planificarlo por FIFO y RR.
  - Es capaz de enviar un proceso a la CPU para que sea procesado.
- **Módulo CPU:**
  - Se conecta a Kernel y recibe un PCB.
  - Es capaz de conectarse a la memoria y solicitar las instrucciones.
  - Es capaz de ejecutar un ciclo básico de instrucción.
  - Es capaz de resolver las operaciones: SET, SUM, SUB, JNZ e IO\_GEN\_SLEEP.
- **Módulo Memoria:**
  - Se encuentra creado y acepta las conexiones.
  - Es capaz de abrir los archivos de pseudocódigo y envía las instrucciones al CPU.
- **Módulo Interfaz I/O:**
  - Se encuentra desarrollada la Interfaz Genérica.

### Check de Control Obligatorio 3: Memoria e interfaces de memoria

**Fecha:** 15/06/2024

**Objetivos:**

- **Módulo Kernel:**
  - Es capaz de planificar por VRR.
  - Es capaz de realizar manejo de recursos.
  - Es capaz de manejar el planificador de largo plazo
- **Módulo CPU:**
  - Es capaz de resolver las operaciones: MOV\_IN, MOV\_OUT, RESIZE, COPY\_STRING, IO\_STDIN\_READ, IO\_STDOUT\_WRITE.
- **Módulo Memoria:**
  - Se encuentra completamente desarrollada.
- **Módulo Interfaz I/O:**
  - Se encuentran desarrolladas las interfaces STDIN y STDOUT.

## Entregas Finales

**Fechas:** 13/07/2024 - 27/07/2024 - 03/08/2024

### 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.

# *The Last of C*

*Siempre puede haber una secuela...*



*Yo no confiaría en el nombre...*

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

-2C2024 -

Versión 1.2

# Índice

<b>Índice</b>	<b>2</b>
<b>Historial de Cambios</b>	<b>4</b>
<b>Objetivos del Trabajo Práctico</b>	<b>5</b>
Características	5
Evaluación del Trabajo Práctico	5
Deployment y Testing del Trabajo Práctico	6
Aclaraciones	6
<b>Definición del Trabajo Práctico</b>	<b>7</b>
¿Qué es el trabajo práctico y cómo empezamos?	7
Arquitectura del sistema	8
Aclaración importante	8
<b>Módulo: Kernel</b>	<b>9</b>
Lineamiento e Implementación	9
Diagrama de estados	9
PCB	10
TCB	10
Planificador de Largo Plazo	10
Creación de procesos	10
Finalización de procesos	10
Creación de hilos	10
Finalización de hilos	11
Planificador de Corto Plazo	11
FIFO	11
Prioridades	11
Colas Multinivel	11
Ejecución	11
Syscalls	12
Procesos	12
Threads	12
Mutex	12
Memory	13
Entrada Salida	13
Errores	13
Logs mínimos y obligatorios	13
Archivo de configuración	14
Ejemplo de Archivo de Configuración	14
<b>Módulo: CPU</b>	<b>15</b>
Lineamiento e Implementación	15
Registros de la CPU	15
Ciclo de Instrucción	16
Fetch	16
Decode	16
Ejemplos de instrucciones a interpretar	16

Execute	17
Check Interrupt	18
MMU	18
Logs mínimos y obligatorios	18
Archivo de configuración	18
Ejemplo de Archivo de Configuración	19
<b>Módulo: Memoria</b>	<b>20</b>
Lineamiento e Implementación	20
Memoria de Sistema	20
Contextos de ejecución	20
Archivos de pseudocódigo	20
Memoria de Usuario	20
Esquema de memoria y Estructuras	20
Comunicación con CPU	21
Obtener contexto de ejecución	21
Actualizar contexto de ejecución	21
Obtener instrucción	21
READ MEM	21
WRITE MEM	21
Retardo en peticiones	21
Comunicación con Kernel	21
Creación de proceso	21
Finalización de proceso	22
Creación de hilo	23
Finalización de hilo	23
Memory Dump	23
Logs mínimos y obligatorios	23
Archivo de configuración	24
Ejemplo de Archivo de Configuración	24
<b>Módulo: File System</b>	<b>26</b>
Lineamiento e Implementación	26
Esquema de archivos	26
Creación de Archivos	26
Logs mínimos y obligatorios	27
Archivo de configuración	27
Ejemplo de Archivo de Configuración	28
<b>Descripción de las entregas</b>	<b>29</b>
Check de Control Obligatorio 1: Conexión inicial	29
Check de Control Obligatorio 2: Módulos Kernel y CPU Completos	29
Check de Control Obligatorio 3: Módulo Memoria Completa	30
Entregas Finales	30

## **Historial de Cambios**

v1.0 (24/08/2024) *Release inicial del trabajo práctico.*

v1.1 (16/06/2024) *Detalle de cambios:*

- Arreglados problemas de redacción donde en varios lados hicimos referencias a procesos y era a hilos.
- Agregados Base y Límite al set de registros.
- Agregados nuevos logs mínimos y obligatorios en memoria.

v1.2 (22/10/2024) *Detalle de cambios:*

- Se aclara que la memoria no se debe compactar.

# 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  $\pm 0$ ) y obligatorio
- Fecha de comienzo: 25/08/2024
- Fecha de primera entrega: Sábado 30 de Noviembre
- Fecha de segunda entrega: Sábado 07 de Diciembre
- Fecha de tercera entrega: Sábado 21 de Diciembre
- 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](#).

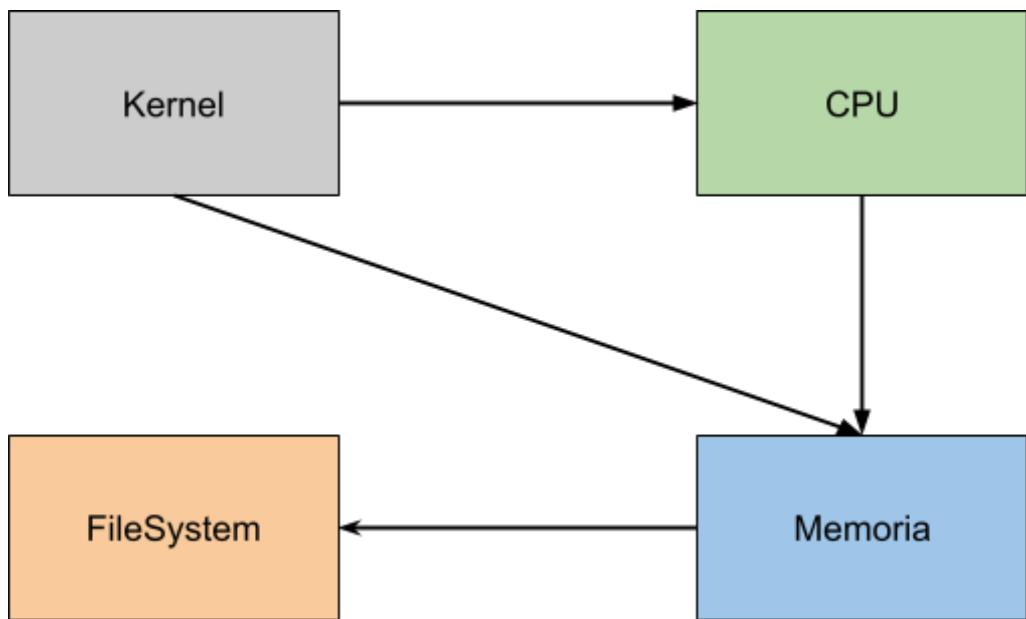
### ¿Qué es el trabajo práctico y cómo empezamos?

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



## Aclaración importante

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

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** los cuales deberán realizarse utilizando la biblioteca de so-commons-library provista por la cátedra y los mismos deberán estar como `LOG_LEVEL_INFO`, pudiendo ser extendidos por necesidad del grupo utilizando `LOG_LEVEL_DEBUG`.

En caso de no cumplir con los logs mínimos y/o tener los logs impresos por pantalla, **se considerará que el TP no es apto para ser evaluado** y por consecuencia el mismo estará **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.

## Lineamiento e Implementación

El módulo Kernel será el encargado de la gestión de los procesos e hilos que se crean dentro de este sistema simulado, planificando su ejecución con diferentes algoritmos y permitiendo simular una entrada/salida, la cual se solicitará mediante syscalls y para lograr su tarea mantendrá una conexión constante con CPU. Además, realizará conexiones efímeras para interactuar con el módulo Memoria.

Para la conexión con la CPU, se tendrá 2 sockets: en el primero enviará los hilos a procesar y se quedará esperando la respuesta de la CPU a dicha solicitud; y el segundo socket será utilizado para enviarle las通知aciones de interrupciones al momento de trabajar con algoritmos que requieran interrumpir a la CPU.

Para las conexiones con el módulo Memoria, el Kernel **deberá crear una nueva conexión para cada petición** que requiera hacerle a la Memoria.

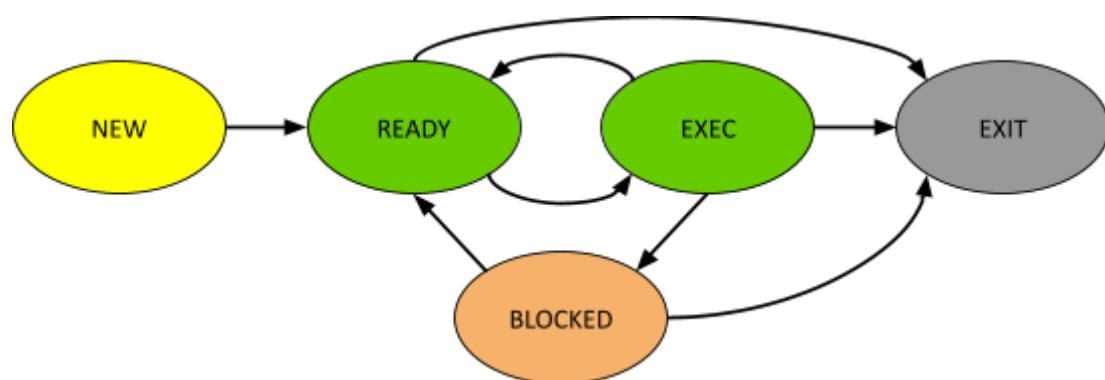
Al iniciar el módulo Kernel, se creará un proceso inicial para que esté lo planifique y para poder inicializarlo, se requerirá entonces que este módulo reciba dos argumentos adicionales del main<sup>1</sup>: el nombre del archivo de pseudocódigo que deberá ejecutar y el tamaño del proceso para ser inicializado en Memoria, el TID 0 creado por este proceso tendrá la prioridad máxima 0 (cero).

Ejemplo de comando de ejecución:

```
./bin/kernel [archivo_pseudocodigo] [tamanio_proceso] [...args]
```

## Diagrama de estados

El kernel utilizará un diagrama de 5 estados para la planificación de los procesos e hilos.



<sup>1</sup> <https://docs.utnso.com.ar/guias/programacion/main>

## PCB

El PCB será la estructura base que utilizaremos dentro del Kernel para administrar cada uno de los procesos. El mismo deberá contener como mínimo los datos definidos a continuación, que representan la información administrativa.

- **PID:** Identificador del proceso (deberá ser un número entero, único en todo el sistema).
- **TIDs:** Lista de los identificadores de los hilos asociados al proceso (deberá ser un número entero, autoincremental dentro del mismo proceso, siendo el primer hilo del proceso el TID 0).
- **Mutex:** Lista de los mutex creados para el proceso a lo largo de la ejecución de sus hilos.

## TCB

El TCB será la estructura base que utilizaremos dentro del Kernel para administrar los hilos de los diferentes procesos.

- **TID:** Identificador del hilo.
- **Prioridad:** Es la prioridad del hilo dentro del sistema.

## Planificador de Largo Plazo

El Kernel será el encargado de gestionar las peticiones a la memoria para la creación y eliminación de procesos e hilos.

### Creación de procesos

Se tendrá una cola NEW que será administrada estrictamente por FIFO para la creación de procesos.

Al llegar un nuevo proceso a esta cola y la misma esté vacía se enviará un pedido a Memoria para inicializar el mismo. Si la respuesta es positiva, se crea el hilo 0 de ese proceso, se lo pasa al estado READY y se sigue la misma lógica con el proceso que sigue. Si la respuesta es negativa (ya que la Memoria no tiene espacio suficiente para inicializarlo) se deberá esperar la finalización de otro proceso para volver a intentar inicializarlo.

Al llegar un proceso a esta cola y haya otros esperando, el mismo simplemente se encola.

### Finalización de procesos

Al momento de finalizar un proceso, el Kernel deberá informar a la Memoria la finalización del mismo y luego de recibir la confirmación por parte de la Memoria deberá liberar su PCB asociado e intentar inicializar uno de los que estén esperando en estado NEW si los hubiere.

### Creación de hilos

Para la creación de hilos, el Kernel deberá informar a la Memoria y luego ingresarlo directamente a la cola de READY correspondiente, según su nivel de prioridad.

## Finalización de hilos

Al momento de finalizar un hilo, el Kernel deberá informar a la Memoria la finalización del mismo, liberar su TCB asociado y deberá mover al estado READY a todos los hilos que se encontraban bloqueados por ese TID. De esta manera, se desbloquean aquellos hilos bloqueados por THREAD\_JOIN o por mutex tomados por el hilo finalizado (en caso que hubiera).

## Planificador de Corto Plazo

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

- **FIFO**
- **Prioridades**
- **Colas multinivel**

### FIFO

Se elegirá al siguiente hilo a ejecutar según su orden de llegada a READY.

### Prioridades

Se elegirá al siguiente hilo a ejecutar según cual tenga el número de prioridad más bajo, siendo 0 la máxima prioridad. En caso de tener varios hilos con la misma prioridad más alta, se desempata por FIFO. Se pide implementar este esquema **sin desalojo**.

### Colas Multinivel

Se elegirá al siguiente hilo a ejecutar según el siguiente esquema de colas multinivel:

- Se tendrá una cola por cada nivel de prioridad existente entre los hilos del sistema.
- El algoritmo entre colas es de prioridades **sin desalojo**.
- Cada cola implementará un algoritmo Round Robin con un quantum (Q) definido por archivo de configuración.
- Al llegar un hilo a ready se posiciona siempre al final de la cola que le corresponda.

## Ejecución

Una vez seleccionado el siguiente hilo a ejecutar, se lo transicionará al estado **EXEC** y se enviará al módulo CPU el TID y su PID asociado a ejecutar a través del puerto de *dispatch*, quedando a la espera de recibir dicho TID después de la ejecución junto con un *motivo* por el cual fue devuelto.

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

Al recibir el TID del hilo en ejecución, en caso de que el motivo de devolución implique replanificar, se seleccionará el siguiente hilo a ejecutar según indique el algoritmo. Durante este período la CPU se quedará esperando.

## Syscalls

### Procesos

Dentro de las syscalls que se pueden atender referidas a procesos, tendremos 2 instrucciones **PROCESS\_CREATE** y **PROCESS\_EXIT**.

**PROCESS\_CREATE**, esta syscall recibirá 3 parámetros de la CPU, el primero será el nombre del archivo de pseudocódigo que deberá ejecutar el proceso, el segundo parámetro es el tamaño del proceso en Memoria y el tercer parámetro es la prioridad del hilo main (TID 0). El Kernel creará un nuevo PCB y un TCB asociado con TID 0 y lo dejará en estado NEW.

**PROCESS\_EXIT**, esta syscall finalizará el PCB correspondiente al TCB que ejecutó la instrucción, enviando todos sus TCBs asociados a la cola de EXIT. Esta instrucción sólo será llamada por el TID 0 del proceso y le deberá indicar a la memoria la finalización de dicho proceso.

### Threads

Las syscalls que puede atender el kernel referidas a threads son 4: **THREAD\_CREATE**, **THREAD\_JOIN**, **THREAD\_CANCEL** y **THREAD\_EXIT**.

**THREAD\_CREATE**, esta syscall recibirá como parámetro de la CPU el nombre del archivo de pseudocódigo que deberá ejecutar el hilo a crear y su prioridad. Al momento de crear el nuevo hilo, deberá generar el nuevo TCB con un TID autoincremental y poner al mismo en el estado READY.

**THREAD\_JOIN**, esta syscall recibe como parámetro un TID, mueve el hilo que la invocó al estado BLOCK hasta que el TID pasado por parámetro finalice. En caso de que el TID pasado por parámetro no exista o ya haya finalizado, esta syscall no hace nada y el hilo que la invocó continuará su ejecución.

**THREAD\_CANCEL**, esta syscall recibe como parámetro un TID con el objetivo de finalizarlo pasando al mismo al estado EXIT. Se deberá indicar a la Memoria la finalización de dicho hilo. En caso de que el TID pasado por parámetro no exista o ya haya finalizado, esta syscall no hace nada. Finalmente, el hilo que la invocó continuará su ejecución.

**THREAD\_EXIT**, esta syscall finaliza al hilo que lo invocó, pasando el mismo al estado EXIT. Se deberá indicar a la Memoria la finalización de dicho hilo.

### Mutex

Las syscalls que puede atender el kernel referidas a threads son 3: **MUTEX\_CREATE**, **MUTEX\_LOCK**, **MUTEX\_UNLOCK**. Para las operaciones de **MUTEX\_LOCK** y **MUTEX\_UNLOCK** donde no se cumpla que el recurso exista, se deberá enviar el hilo a EXIT.

**MUTEX\_CREATE**, crea un nuevo mutex para el proceso sin asignar<sup>2</sup> a ningún hilo.

---

<sup>2</sup> Que un mutex esté sin asignar es el equivalente a que un semáforo contador tenga un valor de 1.

**MUTEX\_LOCK**, se deberá verificar primero que exista el mutex solicitado y en caso de que exista y el mismo no se encuentre tomado se deberá asignar dicho mutex al hilo correspondiente. En caso de que el mutex se encuentre tomado, el hilo que realizó **MUTEX\_LOCK** se bloqueará en la cola de bloqueados correspondiente a dicho mutex.

**MUTEX\_UNLOCK**, se deberá verificar primero que exista el mutex solicitado y esté tomado por el hilo que realizó la syscall. En caso de que corresponda, se deberá desbloquear al primer hilo de la cola de bloqueados de ese mutex y le asignará el mutex al hilo recién desbloqueado. Una vez hecho esto, se devuelve la ejecución al hilo que realizó la syscall **MUTEX\_UNLOCK**. En caso de que el hilo que realiza la syscall no tenga asignado el mutex, no realizará ningún desbloqueo.

## Memory

En este apartado solamente se tendrá la instrucción **DUMP\_MEMORY**. Esta syscall le solicita a la memoria, junto al PID y TID que lo solicitó, que haga un Dump del proceso.

Esta syscall bloqueará al hilo que la invocó hasta que el módulo memoria confirme la finalización de la operación, en caso de error, el **proceso** se enviará a EXIT. Caso contrario, el hilo se desbloquea normalmente pasando a READY.

## Entrada Salida

Para la implementación de este trabajo práctico, el módulo Kernel simulará la existencia de un único dispositivo de Entrada Salida, el cual atenderá las peticiones bajo el algoritmo FIFO. Para “utilizar” esta interfaz, se dispone de la syscall **IO**. Esta syscall recibe como parámetro la cantidad de milisegundos que el hilo va a permanecer haciendo la operación de entrada/salida.

## Errores

Al recibir el TID del hilo en ejecución con motivo de Segmentation Fault, se procederá a finalizar el proceso, siguiendo el mismo comportamiento que si llegara una syscall **PROCESS\_EXIT**.

## Logs mínimos y obligatorios

**Syscall recibida:** “## (<PID>:<TID>) - Solicitó syscall: <NOMBRE\_SYSCALL>”

**Creación de Proceso:** “## (<PID>:0) Se crea el proceso - Estado: NEW”

**Creación de Hilo:** “## (<PID>:<TID>) Se crea el Hilo - Estado: READY”

**Motivo de Bloqueo:** “## (<PID>:<TID>) - Bloqueado por: <PTHREAD\_JOIN / MUTEX / IO>”

**Fin de IO:** “## (<PID>:<TID>) finalizó IO y pasa a READY”

**Fin de Quantum:** “## (<PID>:<TID>) - Desalojado por fin de Quantum”

**Fin de Proceso:** “## Finaliza el proceso <PID>”

**Fin de Hilo:** “## (<PID>:<TID>) Finaliza el hilo”

## 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_CPU	String	IP a la cual se deberá conectar con la CPU
PUERTO_CPU_DISPATCH	Numérico	Puerto de <i>dispatch</i> al cual se deberá conectar con la CPU
PUERTO_CPU_INTERRUPT	Numérico	Puerto de <i>interrupt</i> al cual se deberá conectar con la CPU
ALGORITMO_PLANIFICACION	String	Define el algoritmo de planificación de corto plazo. (FIFO / PRIORIDADES / CMN)
QUANTUM	Numérico	Tiempo en milisegundos del quantum para utilizar bajo el algoritmo RR en Colas Multinivel
LOG_LEVEL	String	Nivel de detalle máximo a mostrar. Compatible con <a href="#">log_level_from_string()</a>

### Ejemplo de Archivo de Configuración

```
IP_MEMORIA=127.0.0.1
PUERTO_MEMORIA=8002
IP_CPU=127.0.0.1
PUERTO_CPU_DISPATCH=8006
PUERTO_CPU_INTERRUPT=8007
ALGORITMO_PLANIFICACION=CMN
QUANTUM=2000
LOG_LEVEL=TRACE
```

# Módulo: CPU

El módulo CPU en nuestro contexto de TP lo que va a hacer es simular los pasos del ciclo de instrucción de una CPU real, de una forma mucho más simplificada.

## Lineamiento e Implementación

El módulo **CPU** es el encargado de interpretar y ejecutar las instrucciones de los *Contextos de Ejecución* recibidos por parte de la **Memoria**. Para ello, ejecutará un ciclo de instrucción simplificado que cuenta con los pasos: Fetch, Decode, Execute y Check Interrupt.

Al momento de recibir un **TID y PID** de parte del Kernel la CPU deberá solicitarle el *contexto de ejecución* correspondiente a la Memoria para poder iniciar su ejecución.

A la hora de ejecutar instrucciones que requieran interactuar directamente con la 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 hilo, se irá actualizando su **Contexto de Ejecución**, que luego será devuelto a la **Memoria** bajo los siguientes escenarios: finalización del mismo (**PROCESS\_EXIT** o **THREAD\_EXIT**), ejecutar una llamada al Kernel (syscall), deber ser desalojado (**interrupción**) o por la ocurrencia de un error Segmentation Fault.

## Registros de la CPU

En la implementación de nuestra CPU, vamos a utilizar una serie de registros para poder modelar la operatoria de una CPU real, es decir, vamos a contar con registros similares a los vistos en Arquitectura de Computadores y algunos registros creados por nosotros mismos a fin de poder facilitar las pruebas.

En la siguiente tabla está el detalle de los registros que deberá tener nuestra CPU, es decir, estará detallado el tamaño del mismo y que tipo de dato se recomienda para su implementación:

Registro	Tamaño	Tipo de Dato	Descripción
PC	4 bytes	uint32_t	Program Counter, indica la próxima instrucción a ejecutar
AX	4 bytes	uint32_t	Registro Numérico de propósito general
BX	4 bytes	uint32_t	Registro Numérico de propósito general
CX	4 bytes	uint32_t	Registro Numérico de propósito general
DX	4 bytes	uint32_t	Registro Numérico de propósito general
EX	4 bytes	uint32_t	Registro Numérico de propósito general
FX	4 bytes	uint32_t	Registro Numérico de propósito general

GX	4 bytes	uint32_t	Registro Numérico de propósito general
HX	4 bytes	uint32_t	Registro Numérico de propósito general
Base	4 bytes	uint32_t	Indica la dirección base de la partición del proceso
Límite	4 bytes	uint32_t	Indica el tamaño de la partición 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 hilo en ejecución.

### 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

El siguiente ejemplo es solo de las instrucciones, no representa un programa funcional.

```

1  SET AX 1
2  SET BX 1
3  SET PC 5
4  SUM AX BX
5  SUB AX BX
6  READ_MEM AX BX
7  WRITE_MEM AX BX
8  JNZ AX 4
9  LOG AX
10 MUTEX_CREATE RECURSO_1
11 MUXTEX_LOCK RECURSO_1
12 MUXTEX_UNLOCK RECURSO_1
13 DUMP_MEMORY
14 IO 1500
15 PROCESS_CREATE proceso1 256 1
16 THREAD_CREATE hilo1 3
17 THREAD_CANCEL 1
18 THREAD_JOIN 1
19 THREAD_EXIT
20 PROCESS_EXIT

```

Las instrucciones detalladas previamente son a modo de ejemplo, su ejecución no necesariamente sigue alguna lógica ni funcionamiento correcto. Al momento de realizar las pruebas, 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.
- **READ\_MEM** (Registro Datos, Registro Dirección): Lee el valor de memoria correspondiente a la dirección física obtenida a partir de la Dirección Lógica que se encuentra en el Registro Dirección y lo almacena en el Registro Datos.
- **WRITE\_MEM** (Registro Dirección, Registro Datos): Lee el valor del Registro Datos y lo escribe en la dirección física de memoria obtenida a partir de la Dirección Lógica almacenada en el Registro Dirección.
- **SUM** (Registro Destino, Registro Origen): Suma al Registro Destino el Registro Origen 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.
- **LOG** (Registro): Escribe en el archivo de log el valor del registro.

Las siguientes instrucciones se considerarán *Syscalls* ya que se deberá actualizar el contexto de ejecución en el módulo Memoria y se le deberá devolver el control al **Kernel** para que este actúe en consecuencia.

- **DUMP\_MEMORY**
- **IO** (Tiempo)
- **PROCESS\_CREATE** (Archivo de instrucciones, Tamaño, Prioridad del TID 0)
- **THREAD\_CREATE** (Archivo de instrucciones, Prioridad)
- **THREAD\_JOIN** (TID)
- **THREAD\_CANCEL** (TID)
- **MUTEX\_CREATE** (Recurso)
- **MUTEX\_LOCK** (Recurso)
- **MUTEX\_UNLOCK** (Recurso)
- **THREAD\_EXIT**
- **PROCESS\_EXIT**

Es importante tener en cuenta las siguientes aclaraciones:

- Los registros utilizados en las operaciones siempre tendrán un valor previamente asignado con la instrucción **SET**.
- Al finalizar el ciclo, el **PC** deberá ser actualizado sumándole 1 en caso de que éste no haya sido modificado por la instrucción.

## Check Interrupt

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

## MMU

A la hora de traducir **direcciones lógicas a físicas**, la CPU debe tomar en cuenta que el esquema de la memoria de este sistema será el de Asignación Contigua, por lo tanto las direcciones lógicas son el desplazamiento dentro de la partición en la que se encuentra el proceso.

Estas traducciones, por lo general en los ejercicios prácticos que se ven en clases y se toman en los parciales / finales se hacen en binario, pero como en el lenguaje C los números enteros se operan independientemente de su base numérica, las direcciones físicas se generarán como:

$$[\text{Base} + \text{desplazamiento}]$$

Se debe validar que las solicitudes se encuentren dentro de la partición asignada, es decir que sea menor al límite de la partición. De fallar dicha validación, ocurrirá un “Segmentation Fault”, en cuyo caso, se deberá actualizar el contexto de ejecución en Memoria y devolver el Tid al Kernel con motivo de Segmentation Fault.

## Logs mínimos y obligatorios

**Obtención de Contexto de Ejecución:** “## TID: <TID> - Solicito Contexto Ejecución”.

**Actualización de Contexto de Ejecución:** “## TID: <TID> - Actualizo Contexto Ejecución”.

**Interrupción Recibida:** “## Llega interrupción al puerto Interrupt”.

**Fetch Instrucción:** “## TID: <TID> - FETCH - Program Counter: <PROGRAM\_COUNTER>”.

**Instrucción Ejecutada:** “## TID: <TID> - Ejecutando: <INSTRUCCION> - <PARAMETROS>”.

**Lectura/Escritura Memoria:** “## TID: <TID> - Acción: <LEER / ESCRIBIR> - Dirección Física: <DIRECCION\_FISICA>”.

## 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

Campo	Tipo	Descripción
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
LOG_LEVEL	String	Nivel de detalle máximo a mostrar. Compatible con <a href="#">log_level_from_string()</a>

### Ejemplo de Archivo de Configuración

```
IP_MEMORIA=127.0.0.1
PUERTO_MEMORIA=8002
PUERTO_ESCUCHA_DISPATCH=8006
PUERTO_ESCUCHA_INTERRUPT=8007
LOG_LEVEL=TRACE
```

# Módulo: Memoria

Este módulo simulará diferentes aspectos de utilización de la Memoria de un sistema, administrando el pseudocódigo de los hilos que ejecutarán en el sistema, sus contextos de ejecución y simulando un esquema de administración de memoria de usuario.

## Lineamiento e Implementación

Al iniciar, esperará la conexión del módulo CPU y levantará un servidor *multihilo* para esperar las peticiones por parte del módulo Kernel. Estas peticiones se deberán atender de manera concurrente teniendo 1 hilo por cada petición, el cual deberá finalizar luego de notificar el resultado de la misma al módulo kernel y de cerrar la conexión.

Luego, para las eventuales peticiones al módulo FileSystem se **deberá crear una nueva conexión para cada una** que requiera hacerse.

Las estructuras de este módulo las dividiremos en dos categorías, Memoria del Sistema y Memoria de Usuario.

## Memoria de Sistema

### Contextos de ejecución

Se deberá almacenar, por cada PID del sistema, la parte del contexto de ejecución común para el proceso, en este caso, es la requerida para poder traducir las direcciones lógicas a físicas: **base** y **límite**.

Luego, por cada TID del sistema, se tendrán los registros de la CPU propios de cada hilo: **AX, BX, CX, DX, EX, FX, GX, HX y PC**. Siendo todos ellos inicializados en 0.

### Archivos de pseudocódigo

Por cada TID del sistema, se deberá leer su archivo de pseudocódigo y guardar de forma estructurada las instrucciones del mismo para poder devolverlas una a una a pedido de la CPU. Queda a criterio del grupo utilizar la estructura que crea conveniente para este caso de uso.

## Memoria de Usuario

### Esquema de memoria y Estructuras

La memoria al trabajar bajo dos esquemas diferentes. Al iniciar la memoria se definirá el esquema a utilizar mediante el archivo de configuración, estos esquemas pueden ser:

- Particiones fijas.
- Particiones dinámicas.

Ambos esquemas estarán compuestos por 2 estructuras principales las cuales son:

- Un espacio contiguo de memoria (representado por un **void\***). Este representará el espacio de usuario de la misma, donde los procesos/hilos podrán leer y/o escribir.
- La lista de particiones.

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.

## Comunicación con CPU

### Obtener contexto de ejecución

Deberemos devolverle el contexto de ejecución completo para el PID-TID pedido: **AX, BX, CD, DX, EX, FX, GX, HX, PC, base y límite**.

### Actualizar contexto de ejecución

Deberemos actualizar los registros de la CPU propios del hilo en su estructura correspondiente y responder la petición como OK.

### Obtener instrucción

Deberemos devolverle la instrucción correspondiente al hilo y al *Program Counter* recibido. Por ejemplo, si el hilo 3 del proceso 1 pide la instrucción número 4, deberemos devolver la 5ta instrucción del pseudocódigo correspondiente a ese hilo.

### READ MEM

Se deberá devolver el valor correspondiente a los primeros 4 bytes a partir del byte enviado como *dirección física* dentro de la Memoria de Usuario.

### WRITE MEM

Se escribirán los 4 bytes enviados a partir del byte enviado como *dirección física* dentro de la Memoria de Usuario y se responderá como OK.

### Retardo en peticiones

Ante cada una de las peticiones definidas anteriormente se deberá aplicar el tiempo de espera en milisegundos definido por archivo de configuración.

## Comunicación con Kernel

### Creación de proceso

Al momento de crear los procesos deberemos asignarles espacio en la memoria de usuario teniendo en cuenta qué esquema estamos utilizando, es por esto que vamos a diferenciar los dos esquemas:

**Particiones Fijas:** En este esquema la lista de particiones vendrá dada por archivo de configuración y la misma no se podrá alterar a lo largo de la ejecución.

**Particiones Dinámicas:** En este esquema la lista de particiones, va a iniciar como una única partición libre del tamaño total de la memoria y la misma se va a ir subdividiendo a medida que lleguen los pedidos de creación de los procesos, es por esto que la lista será dinámica.

En ambos esquemas, el hueco a asignar y/o fraccionar se deberá elegir utilizando alguna de las siguientes estrategias:

- First Fit
- Best Fit
- Worst Fit

En caso de encontrar un hueco libre, se le asignará la partición, se creará la estructura necesaria para administrar la Memoria de Sistema, y se responderá como OK al Kernel.

En caso de no encontrar un hueco libre, se le responderá al Kernel que el proceso no pudo ser inicializado. Para este trabajo práctico, no debe realizarse el proceso de compactación bajo ningún caso.

### Finalización de proceso

Esta petición podrá venir solamente desde el módulo Kernel. El módulo Memoria, al ser finalizado un proceso, debe liberar su espacio de memoria marcando la partición ocupada por ese proceso como libre. No es requerido sobreescribir el contenido de la partición. Así mismo, debe eliminar las estructuras correspondientes de Memoria del Sistema.

Solo para el esquema de Particiones Dinámicas, en caso de que la partición liberada tenga particiones libres aledañas, las mismas deberán consolidarse en una única nueva partición libre, como se ve en el siguiente ejemplo:



Al liberarse quedan al menos dos particiones libres juntas

En este caso se consolidan.

## Creación de hilo

Al momento de recibir la creación de un hilo, se deberán crear las estructuras correspondientes a la Memoria del Sistema con los valores iniciales, por lo tanto, se deberá leer el archivo de pseudocódigo para generar las estructuras que el grupo implementó para poder devolver las instrucciones de a 1 a la CPU según ésta se las solicite por medio del Program Counter. Finalmente, se responde como OK.

## Finalización de hilo

Al momento de recibir la finalización de un hilo, se deben eliminar sus estructuras correspondientes en Memoria del Sistema y responder como OK.

## Memory Dump

Al momento de recibir la operación de memory dump el módulo memoria deberá solicitar al módulo FileSystem la creación de un nuevo archivo con el tamaño total de la memoria reservada por el proceso y debe escribir en dicho archivo todo el contenido actual de la memoria del proceso. El archivo debe llamarse “<PID>-<TID>-<TIMESTAMP>.dmp”.

En caso de que el FileSystem responda con error, se devolverá el mismo mensaje al Kernel, en caso positivo, se responde como OK.

## Logs mínimos y obligatorios

**Conexión de Kernel<sup>3</sup>:** “## Kernel Conectado - FD del socket: <FD\_DEL\_SOCKET>”

**Creación / destrucción de Proceso:** “## Proceso <Creado/Destruido> - PID: <PID> - Tamaño: <TAMAÑO>”

---

<sup>3</sup> Los FD al ser un valor int, se pueden imprimir %d, para más información sobre los FD, ver la [Guía de Sockets](#)

**Creación / destrucción de Hilo:** “## Hilo <Creado/Destruido> - (PID:TID) - (<PID>:<TID>)”

**Solicitud / actualización de Contexto:** “## Contexto <Solicitado/Actualizado> - (PID:TID) - (<PID>:<TID>)”

**Obtener instrucción:** “## Obtener instrucción - (PID:TID) - (<PID>:<TID>) - Instrucción: <INSTRUCCIÓN> <...ARGS>”

**Escritura / lectura en espacio de usuario:** “## <Escritura/Lectura> - (PID:TID) - (<PID>:<TID>) - Dir. Física: <DIRECCIÓN\_FÍSICA> - Tamaño: <TAMAÑO>”

**Memory Dump:** “## Memory Dump solicitado - (PID:TID) - (<PID>:<TID>)”

## 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 el FileSystem
PUERTO_FILESYSTEM	Numérico	Puerto al cual se deberá conectar con el FileSystem
TAM_MEMORIA	Numérico	Tamaño expresado en bytes del espacio de usuario de la memoria.
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.
ESQUEMA	String	Esquema de particiones de memoria a utilizar FIJAS / DINAMICAS
ALGORITMO_BUSQUEDA	String	Algoritmo de búsqueda de huecos de memoria: FIRST / BEST / WORST
PARTICIONES	Lista	Lista ordenada con las particiones a generar en el algoritmo de Particiones Fijas
LOG_LEVEL	String	Nivel de detalle máximo a mostrar. Compatible con <a href="#">log_level_from_string()</a>

## Ejemplo de Archivo de Configuración

PUERTO\_ESCUCHA=8002

IP\_FILESYSTEM=127.0.0.1  
PUERTO\_FILESYSTEM=8003  
TAM\_MEMORIA=1024  
PATH\_INSTRUCCIONES=/home/utnso/scripts-pruebas  
RETARDO\_RESPUESTA=1000  
ESQUEMA=FIJAS  
ALGORITMO\_BUSQUEDA=FIRST  
PARTICIONES=[512, 16, 32, 16, 256, 64, 128]  
LOG\_LEVEL=TRACE

# Módulo: File System

El módulo **File System** será el encargado de persistir la información suministrada por el módulo **Memoria** por medio de una serie de archivos, estos archivos se encontrarán detallados a continuación.

## Lineamiento e Implementación

Al iniciarse el módulo se iniciará un servidor multihilo que quedará a la espera de las peticiones del módulo **Memoria**, las cuales las deberá atender de manera concurrente teniendo 1 hilo para cada petición recibida el cual deberá finalizar al momento de concluida la petición, luego de notificar el resultado de la operación a la memoria y de cerrar la conexión.

Ante *cada acceso a bloque* se deberá esperar un tiempo definido por archivo de configuración.

### Esquema de archivos

El File System implementará una simulación de un esquema de asignación de bloques indexado puro, donde el tamaño del puntero de bloque es de 4 bytes<sup>4</sup> para ello contará los siguientes archivos:

**bitmap.dat**: Este archivo representará el bitmap del FS y contendrá 1 bit por cada bloque del archivo donde si el valor del bit es 0 indicará que el bloque se encuentra vacío y si su valor es 1 indica que el bloque se encuentra ocupado, por lo tanto, el tamaño de este archivo deberá ser `BLOCK_COUNT / 8` (redondeado al valor superior). Este archivo se encontrará directamente en la carpeta `MOUNT_DIR`

**bloques.dat**: Este archivo será el archivo que contendrá tanto los bloques de datos como los bloques de índices de los diferentes archivos creados en el FS y su tamaño será `BLOCK_COUNT * BLOCK_SIZE`. Este archivo se encontrará directamente en la carpeta `MOUNT_DIR`

**Archivos de metadata**: Estos archivos se encontrarán dentro de la carpeta `/files`, a partir del punto de montaje (`MOUNT_DIR`) y se tendrá 1 archivo por cada archivo que se cree en el FS, donde su nombre va a ser el nombre del archivo en el FS, por ej, `1-0-12:51:59:331.dmp`. El contenido de estos archivos consistirá de el tamaño en bytes del archivo (`SIZE`) y el número de bloque que corresponde al bloque de índices (`INDEX_BLOCK`), un ejemplo del contenido puede ser el siguiente:

```
SIZE=4  
INDEX_BLOCK=10
```

### Creación de Archivos

La creación de los archivos de DUMP de memoria se crearán al recibir desde Memoria la petición de creación de un nuevo archivo de DUMP. En la petición deberá venir el nombre del archivo, el tamaño y el contenido a grabar en el mismo.

La creación se dividirá en los siguientes pasos:

---

<sup>4</sup> Se recomienda utilizar el tipo de dato `uint32_t`

1. Verificar que se cuenta con el espacio disponible, en caso de que **no** se cuente con el espacio disponible, se deberá informar a memoria del error y finalizar la creación del archivo en este punto.
2. Reservar el bloque de índice y los bloques de datos correspondientes en el bitmap
3. Crear el archivo de metadata con los datos requeridos y el siguiente formato: <PID>-<TID>-<TIMESTAMP>.dmp.
4. Acceder al bloque de punteros y grabar todos los bloques reservados.
5. Acceder bloque a bloque e ir escribiendo el contenido de la memoria.

Esta operación resulta en que se deberán tener N+1 accesos a bloques, donde N es el número de bloques de datos del archivo y los retardos deberán realizarse luego de acceder a cada bloque.

## Logs mínimos y obligatorios

**Creación Archivo:** “## Archivo Creado: <NOMBRE\_ARCHIVO> - Tamaño: <TAMAÑO\_ARCHIVO>”

**Asignación de bloque:** “## Bloque asignado: <NUMERO\_BLOQUE> - Archivo: <NOMBRE\_ARCHIVO> - Bloques Libres: <CANTIDAD\_BLOQUES\_LIBRES>”

**Acceso a Bloque:** “## Acceso Bloque - Archivo: <NOMBRE\_ARCHIVO> - Tipo Bloque: <DATOS / ÍNDICE> - Bloque File System <NUMERO\_BLOQUE\_FS>”

**Fin Petición:** “## Fin de solicitud - Archivo: <NOMBRE\_ARCHIVO>”

## Archivo de configuración

Campo	Tipo	Descripción
PUERTO_ESCUCHA	Numérico	Puerto en el cual se escuchará la conexión de módulo.
MOUNT_DIR	String	Path a partir del cual van a encontrarse los archivos del FS.
BLOCK_SIZE	Numérico	Tamaño de los bloques del FS
BLOCK_COUNT	Numérico	Cantidad de bloques del FS
RETARDO_ACCESO_BLOQUE	Numérico	Tiempo en milisegundos que se deberá esperar luego de cada acceso a bloques (de datos o punteros)
LOG_LEVEL	String	Nivel de detalle máximo a mostrar. Compatible con <a href="#">log_level_from_string()</a>

### Ejemplo de Archivo de Configuración

```
PUERTO_ESCUCHA=8003
MOUNT_DIR=/home/utnso/mount_dir
BLOCK_SIZE=64
BLOCK_COUNT=1024
RETARDO_ACCESO_BLOQUE=15000
LOG_LEVEL=TRACE
```

## 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.

### Check de Control Obligatorio 1: Conexión inicial

**Fecha:** 07/09/2024

**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.

### Check de Control Obligatorio 2: Módulos Kernel y CPU Completos

**Fecha:** 12/10/2024

**Objetivos:**

- **Módulo Kernel:**
  - Completo
- **Módulo CPU:**
  - Completo
- **Módulo Memoria:**
  - Crea, actualiza y devuelve a CPU los contextos de ejecución.
  - Devuelve las instrucciones pedidas por CPU.
  - Para todas las demás peticiones responde OK en formato stub<sup>5</sup> (sin hacer nada).

**Carga de trabajo estimada:**

- **Módulo Kernel:** 50%
- **Módulo CPU:** 25%
- **Módulo Memoria:** 25%

### Check de Control Obligatorio 3: Módulo Memoria Completa

**Fecha:** 09/11/2024

**Objetivos:**

- **Módulo Memoria:**
  - Completo
- **Módulo Filesystem:**
  - Inicialización de estructuras
  - Para todas las peticiones responde OK en formato stub (sin hacer nada).

**Carga de trabajo estimada:**

- **Módulo Memoria:** 75%
- **Módulo Filesystem:** 25%

<sup>5</sup> <https://martinfowler.com/articles/mocksArentStubs.html#TheDifferenceBetweenMocksAndStubs>

## Entregas Finales

**Fechas:** 30/11/2024, 07/12/2024, 21/12/2024

### 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.