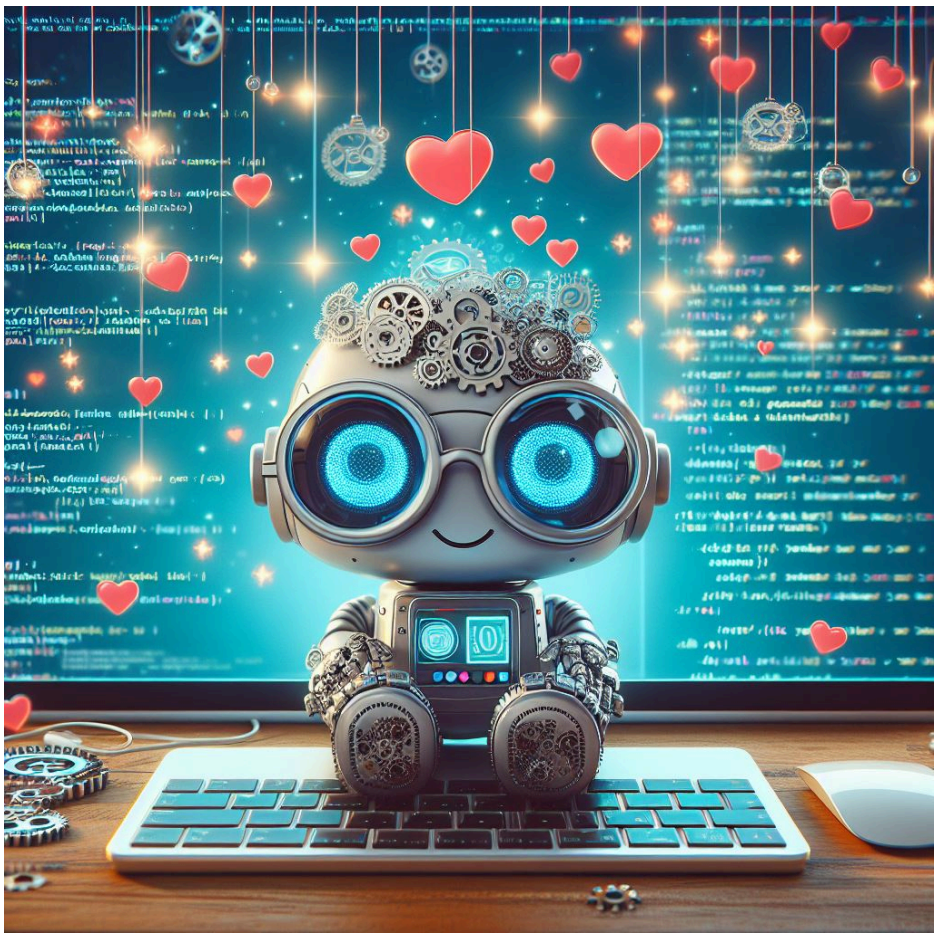
A.L.GO

Actualizando lentamente a GO



El proceso será lento…

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

-1C2024 -
Versión 1.1

Índice

Historial de Cambios

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

v1.1 (02/06/2024) Se actualiza información sobre cómo ejecutan los I/O las distintas peticiones simultáneas

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 del lenguaje Golang.

Características

- Modalidad: grupal (5 integrantes ± 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.
- Comunicación entre procesos: La comunicación entre procesos se realizará por medio de API's. A lo largo del trabajo práctico se definirán algunas de ellas que ***deben ser respetadas***. Todas aquellas definiciones que no se encuentren explícitamente definidas, es decisión del grupo como hacerlas y definirlas.

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.

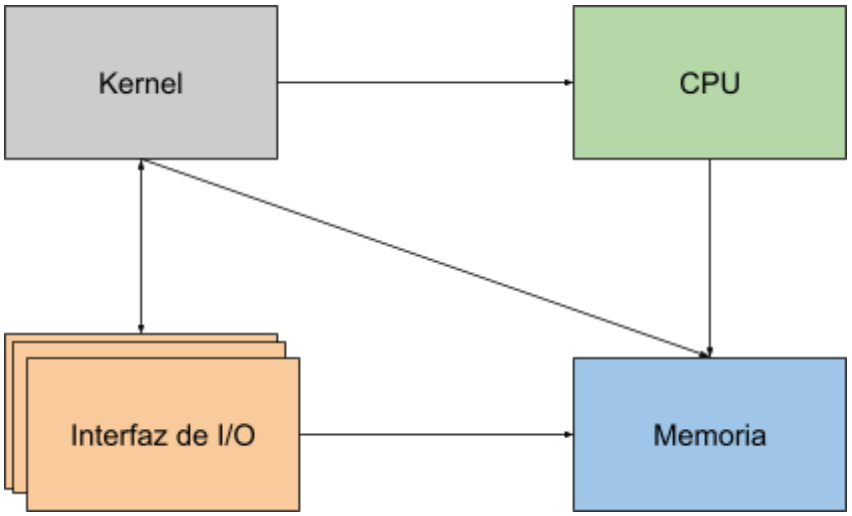
¿Qué es el trabajo práctico y cómo empezamos?

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 api.

Lineamiento e Implementación

El módulo Kernel será el encargado de iniciar los procesos del sistema, para ello contará con una serie de API's definida por la cátedra que permitirá las siguientes operaciones:

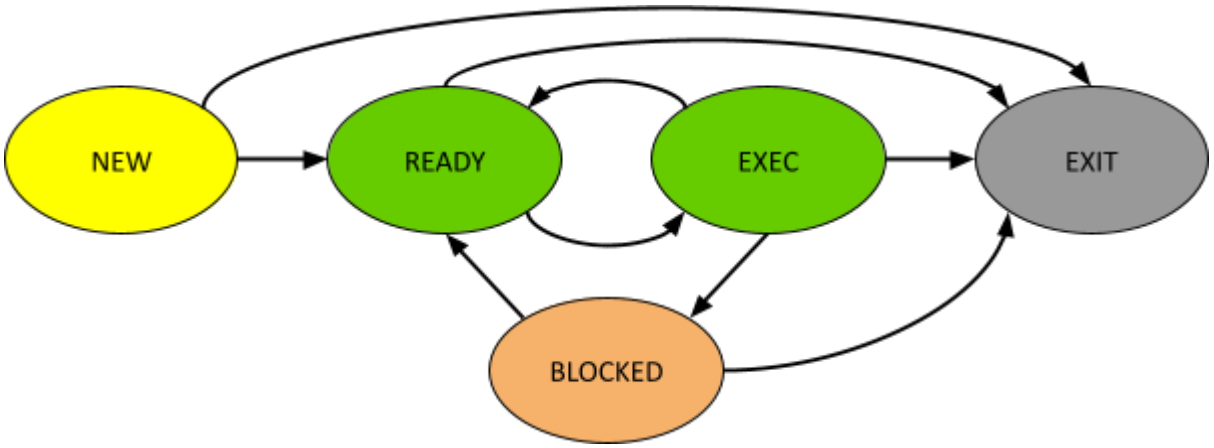
- Iniciar proceso
- Finalizar proceso
- Estado proceso
- Iniciar planificación
- Detener planificación
- Listar procesos

Además, el Kernel será el encargado de gestionar las peticiones contra la Memoria (a través de una conexión de *dispatch* y otra de *interrupt*) 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 api 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.

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 api 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.

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)T

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, además de la ip y puerto donde estén corriendo. Estos datos se recibirán cuando las interfaces se conecten por primera vez 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.

API's

Las siguientes API's deben ser implementadas cumpliendo la estructura indicada por la cátedra. El caso de no cumplir dicho requerimiento es motivo de desaprobación.

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.

Endpoint: PUT /process

Body:

```
{
    "pid": 0
    "path": "string"
}
```

Response:

```
{
    "pid": 0
}
```

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).

Endpoint: DELETE /process/{pid}

Sin Body y Response vacío.

Estado Proceso

Se encargará de retornar el estado en el que se encuentra un pid determinado.

Endpoint: GET /process/{pid}

Sin Body.

Response:

```
{
    "state": "string", // READY, EXEC, BLOCK, FIN
}
```

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.

Endpoint: PUT /plani

Sin Body y Response vacío.

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.

Endpoint: DELETE /plani

Sin Body y Response vacío.

Listar procesos

Se encargará de mostrar por consola y retornar por la api el listado de procesos que se encuentran en el sistema con su respectivo estado dentro de cada uno de ellos.

Endpoint: GET /process

Sin Body.

Response:

```
[
```

```
    {
        "pid": 0,
        "state": "string" // READY, EXEC, BLOCK, FIN
    }
]
```

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 <COLA>: [<LISTA DE PIDS>]"

Archivo de configuración

Campo	Tipo	Descripción
port	Numérico	Puerto en el cual se levantará la API del Kernel
ip_memory	String	IP a la cual se deberá conectar con la Memoria
port_memory	Numérico	Puerto al cual se deberá conectar con la Memoria
ip_cpu	String	IP a la cual se deberá conectar con la CPU
port_cpu	Numérico	Puerto al cual se deberá conectar con la CPU
planning_algorithm	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
resources	Lista	Lista ordenada de los nombres de los recursos compartidos del sistema
resource_instances	Lista	Lista ordenada de la cantidad de unidades por recurso

Campo	Tipo	Descripción
multiprogramming	Numérico	Grado de multiprogramación del módulo

Ejemplo de Archivo de Configuración

```
{
    "port": 8001,
    "ip_memory": "127.0.0.1",
    "port_memory": 8002,
    "ip_cpu": "127.0.0.1",
    "port_cpu": 8006,
    "planning_algorithm": "VRR",
    "quantum": 2000,
    "resources": ["RA","RB","RC"],
    "resource_instances": [1,2,1],
    "multiprogramming": 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**), necesitar ser bloqueado, 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.

```
SET AX 1
SET BX 1
SET PC 5
SUM AX BX
SUB AX BX
MOV_IN EDX ECX
MOV_OUT EDX ECX
RESIZE 128
JNZ AX 4
COPY_STRING 8
IO_GEN_SLEEP Int1 10
IO_STDIN_READ Int2 EAX AX
IO_STDOUT_WRITE Int3 BX EAX
IO_FS_CREATE Int4 notas.txt
IO_FS_DELETE Int4 notas.txt
IO_FS_TRUNCATE Int4 notas.txt ECX
IO_FS_WRITE Int4 notas.txt AX ECX EDX
IO_FS_READ Int4 notas.txt BX ECX EDX
WAIT RECURSO_1
SIGNAL RECURSO_1
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.
- **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
- **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 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_SEEK** (Intefaz, Nombre Archivo, Registro Puntero):~~
- **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 que **una dirección lógica puede traducirse a más de una dirección física**.

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

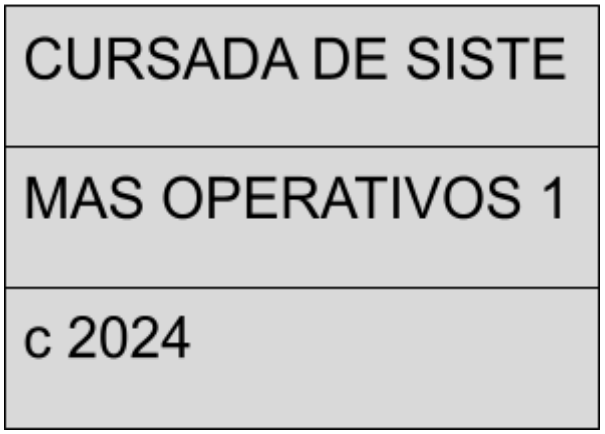
A la hora de **traducir direcciones lógicas a físicas**, la CPU debe tomar en cuenta que el esquema de memoria del sistema es de **Paginación**. Por lo tanto, las direcciones lógicas se compondrán de la siguiente manera:

[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 GO 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:



Con esto queremos decir que es posible que una traducción de memoria, pueda devolver más de una dirección física asociada.

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.

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 - Pagina: <NUMERO_PAGINA>”

TLB Miss: “PID: <PID> - TLB MISS - Pagina: <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
port	Numérico	Puerto en el cual se levantará la API del CPU
ip_memory	String	IP a la cual se deberá conectar con la Memoria
port_memory	Numérico	Puerto al cual se deberá conectar con la Memoria
number_felling_tlb	Numérico	Cantidad de entradas que tendrá la TLB
algorithm_tlb	String	Algoritmo de reemplazo de la TLB FIFO / LRU

Ejemplo de Archivo de Configuración

```
{
    "port": 8003,
    "ip_memory": "127.0.0.1",
    "port_memory": 8002,
    "number_felling_tlb": 32,
    "algorithm_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

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 **array de bytes**). Este representará el espacio de usuario de la misma, donde los procesos podrán leer y/o escribir.[1]
- 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.

[1] Para inicializar la memoria se debe utilizar SIN excepcion la siguiente función
make([]byte, TamMemoria)

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’.

Cada petición tendrá un 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
port	Numérico	Puerto en el cual se escuchará la conexión de módulo.
memory_size	Numérico	Tamaño expresado en bytes del espacio de usuario de la memoria.
page_size	Numérico	Tamaño de las páginas en bytes.
instructions_path	String	Carpeta donde se encuentran los archivos de pseudocódigo.
delay_response	Numérico	Tiempo en milisegundos que se deberá esperar antes de responder a las solicitudes de CPU y FS.

Ejemplo de Archivo de Configuración

```
{
    "port": 8002,
    "memory_size": 4096,
    "page_size": 16,
    "instructions_path": "/home/utnso/mappa-pruebas",
    "delay_response": 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 solamente 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:

- type
- unit_work_time
- ip_kernel
- port_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 dirección física indicada 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:

- type
- ip_kernel
- port_kernel
- ip_memory
- port_memory

Interfaces STDOUT

Las interfaces STDOUT se conectan a memoria para leer una dirección física y mostrar el resultado. Siempre van a consumir una unidad de ***unit_work_time***.

La instrucción para realizar esto es:

- IO_STDOUT_WRITE

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

- type
- unit_work_time
- ip_kernel
- port_kernel
- ip_memory
- port_memory

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 ***unit_work_time***.

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 dirección física de la memoria, por lo que el Módulo de Interfaz de I/O se va a tener que conectar con la memoria para pedirle información o para 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 archivo DialFS

Esta implementación de FS, lo que busca es ser una implementación sumamente simple que va a implementar un FS de Asignación Contigua.

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

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: **dialfs_block_size** y **dialfs_block_count**, y el tamaño del archivo va a ser **dialfs_block_size * dialfs_block_count**.

El segundo archivo va a ser un archivo que va a contener un bitmap (*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, 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**. El mismo será representado en formato JSON de la siguiente manera:

```
{
    "initial_block": 25,
    "size": 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 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 archivos de manera tal que quede todo el espacio libre luego del archivo que se desea truncar.

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

- type
- unit_work_time
- ip_kernel
- port_kernel
- ip_memory
- port_memory
- dialfs_path

- dialfs_block_size
- dialfs_block_count

Logs mínimos y obligatorios

Todos - Operación: "`PID: <PID> - Operacion: <OPERACION_A_REALIZAR>`"

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>`"

Archivo de configuración

Campo	Tipo	Descripción
port	Numérico	Puerto en donde se inicializará la API de la I/O
type	String	Indica el tipo de Interfaz de I/O que estamos creando. GENERICA / STDIN / STDOUT / DIALFS
unit_work_time	Numérico	Tiempo en milisegundos que dura cada unidad de trabajo
ip_kernel	Numérico	IP a la cual se deberá conectar con el Kernel
port_kernel	String	Puerto al cual se deberá conectar con el Kernel
ip_memory	String	IP a la cual se deberá conectar con la Memoria
port_memory	Numérico	Puerto al cual se deberá conectar con la Memoria
dialfs_path	String	Path a partir del cual van a encontrarse los archivos de DialFS.
dialfs_block_size	Numérico	Tamaño de los bloques del FS
dialfs_block_count	Numérico	Cantidad de bloques del FS

Ejemplo de Archivo de Configuración

```
{
    "port": 8005,
    "type": "STDOUT",
    "unit_work_time": 250,
    "ip_kernel": "127.0.0.1",
    "port_kernel": 8001,
    "ip_memory": "127.0.0.1",
    "port_memory": 8002,
    "dialfs_path": "/home/utnso/dialfs",
    "dialfs_block_size": 64,
    "dialfs_block_count": 1024
}
```

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.
- Todas las API del módulo Kernel definidas por la cátedra están creadas y retornan datos hardcodeados.
- Todos los módulos están creados y son capaces de inicializarse con al menos una API.

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.