GOD

Go Direction

El proceso era lento…

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

-2C2024 -
Versión 1.2

Índice

Historial de Cambios

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

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

- *Se actualiza información sobre cómo ejecutan los I/O las distintas peticiones simultáneas*
- *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 (2X/10/2024) Detalle de cambios:

- *Se agrega especificación de cómo iniciar la compactación tanto en Kernel como en Memoria.*

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: 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.
- 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.

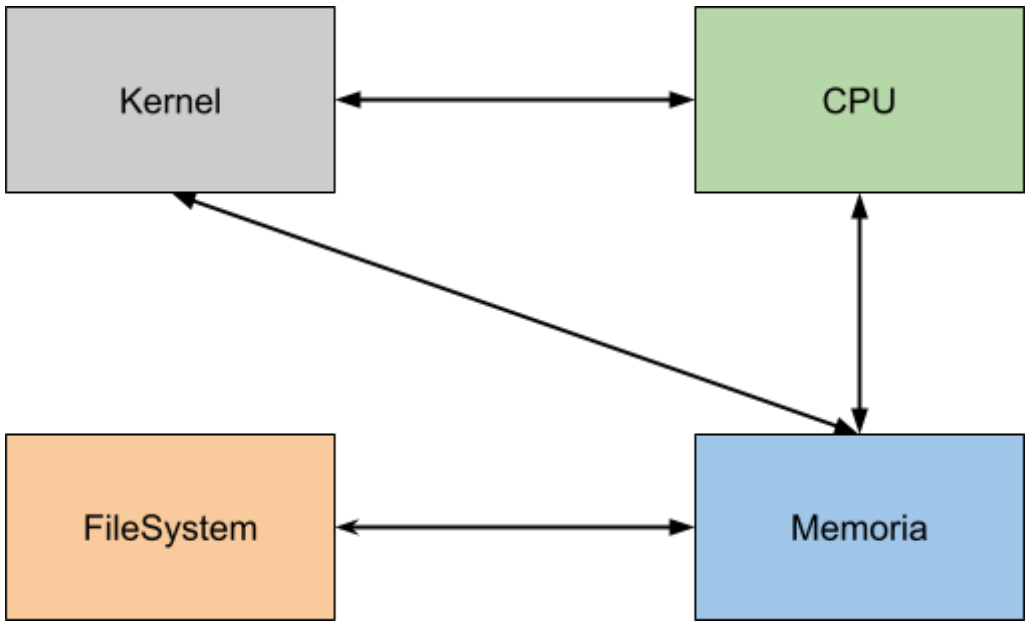
¿Qué es el trabajo práctico y cómo empezamos?

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



Módulos

Todos los módulos de la arquitectura serán API's que se deberán exponer a través del protocolo HTTP. Esta arquitectura nos permitirá una comunicación en ambos sentidos pudiendo modularizar y atomizar las operaciones que se realicen.

De esta manera, cada acción de "Módulo A debe comunicar a Módulo B" define una API que debe ser creada en el Módulo B mientras que cada acción de "Módulo B debe comunicar a Módulo A" define una API que debe ser creada en el Módulo A.

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 log/slog sugerida por la cátedra y los mismos deberán estar como `Logger.Info()`, pudiendo ser extendidos por necesidad del grupo utilizando `Logger.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 creen dentro de este sistema simulado, planificando su ejecución con diferentes algoritmos y permitiendo simular una entrada / salida la cual se solicitará mediante syscalls, para lograr su tarea se conectará con los módulos CPU y Memoria.

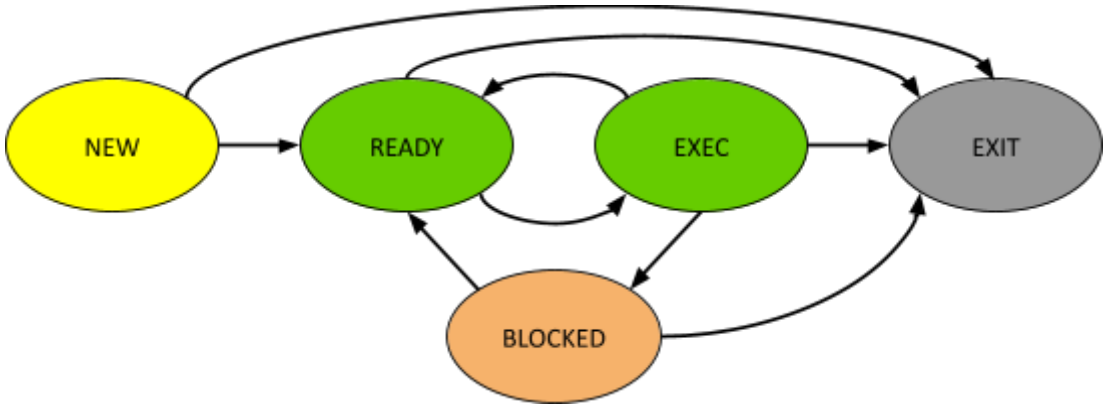
Para las conexiones con ambos módulos se utilizarán distintas API's definidas en cada uno de los módulos destinos a razón de una API por tipo de pedido/solicitud.

Al iniciar el módulo Kernel, se creará un proceso inicial para que éste lo planifique. Para poder inicializarlo se requerirá que el Kernel recibirá dos parámetros (argumentos): 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).

```
./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.



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.

- **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, la memoria puede retornar dos valores:

- No tiene espacio total para almacenar el proceso: En este caso se deberá esperar a la finalización de otro proceso para volver a intentar inicializarlo.
- No tiene espacio continuo pero si en huecos: Se deberá solicitar a memoria la realización del proceso de compactación. Para esto, el kernel debe pausar su planificación a corto plazo y una vez que no tenga un hilo en Exec solicitarle a memoria que realice la compactación. Una vez que la memoria responda que el proceso de compactación fue finalizado se debe volver a intentar iniciar el proceso y activar la planificación de corto plazo.

Al llegar un proceso a esta cola, si ya hay 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 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 y 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 máxima prioridad, se desempata por FIFO. Se pide implementar este esquema **con desalojo** de manera que si llega un hilo a la cola de READY con mayor prioridad que el hilo que está actualmente ejecutando, se deberá desalojar a este último.

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 **con desalojo** (mismo funcionamiento anteriormente explicado).
- 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. Esto aplica también al desalojo entre colas, es decir, al desalojar un hilo por la llegada de otro de más prioridad, se pondrá al final de su respectiva cola (al igual que si hubiera sido desalojado por quantum).

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, quedando a la espera de recibir (en una API expuesta por el Kernel) 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 al módulo CPU a través de otra API forzando el desalojo del mismo y por consiguiente la invocación a la API expuesta por el Kernel indicada en el párrafo anterior.

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[1] a ningún hilo.

[1] 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_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
sheduler_algorithm	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 slog.SetLogLoggerLevel()

Ejemplo de Archivo de Configuración

```
{
    "ip_memory": "127.0.0.1",
    "port_memory": 8002,
    "ip_cpu": "127.0.0.1",
    "port_cpu": 8002,
    "sheduler_algorithm": "CMN",
    "quantum": 2000,
    "log_level": "DEBUG"
}
```

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,** donde se informará 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.

```
SET AX 1
SET BX 1
SET PC 5
SUM AX BX
SUB AX BX
READ_MEM AX BX
WRITE_MEM AX BX
JNZ AX 4
LOG AX
MUTEX_CREATE RECURSO_1
MUTEX_LOCK RECURSO_1
MUTEX_UNLOCK RECURSO_1
DUMP_MEMORY
IO 1500
PROCESS_CREATE proceso1 256 1
THREAD_CREATE hilo1 3
THREAD_CANCEL 1
THREAD_JOIN 1
THREAD_EXIT
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 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 Go los números enteros se operan independientemente de su base numérica, las direcciones físicas se generarán como:

```
[Base + 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 interrupcion 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_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_kernel	String	IP a la cual se deberá conectar con la Kernel

Campo	Tipo	Descripción
port_kernel	Numérico	Puerto al cual se deberá conectar con la Kernel
port	Numérico	Puerto en el cual se escuchará las conexiones
log_level	String	Nivel de detalle máximo a mostrar. Compatible con slog.SetLogLoggerLevel()

Ejemplo de Archivo de Configuración

```
{
    "ip_memory": "127.0.0.1",
    "port_memory": 8002,
    "ip_kernel": "127.0.0.1",
    "port_kernel": 8002,
    "port": 8002,
    "log_level": "DEBUG"
}
```

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

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 **array de bytes**). Este representará el espacio de usuario de la misma, donde los procesos/hilos podrán leer y/o escribir.[2]
- La lista de particiones.

---

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

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.

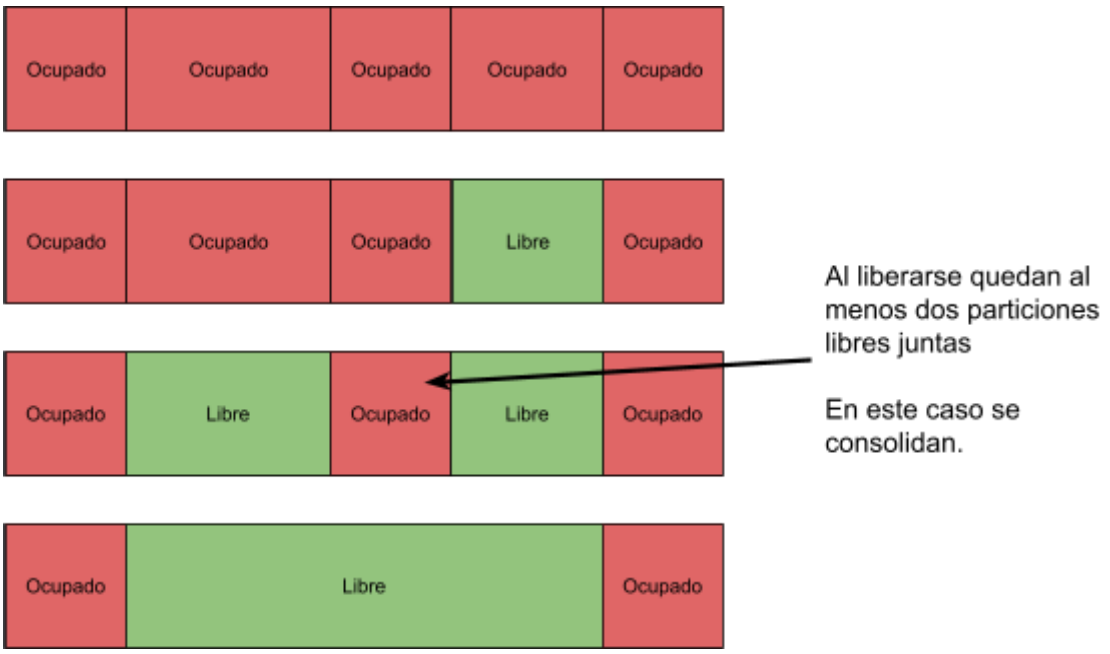
En caso de no encontrar un hueco libre:

- **Particiones Fijas**: Se le responderá al Kernel que el proceso no pudo ser inicializado.
- **Particiones Dinámicas**: Se deberá verificar si el total de los huecos libres permite la inclusión del espacio requerido. De ser así, **y solo así**, se deberá notificar al kernel de la necesidad de compactar para poder asignar la memoria solicitada al proceso. Una vez que el Kernel indique que se debe compactar, se inicializará dicho proceso. En caso que el total de los huecos libres no permita la inclusión del espacio requerido, se le responderá al Kernel que el proceso no pudo ser inicializado.

Finalización de proceso

Esta petición podrá venir 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:



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

Creación / destrucción de Proceso: "`## Proceso <Creado/Destruido> - PID: <PID> - Tamaño: <TAMAÑO>`"

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
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.
instruction_path	String	Carpeta donde se encuentran los archivos de pseudocódigo.
response_delay	Numérico	Tiempo en milisegundos que se deberá esperar antes de responder a las solicitudes de CPU.
ip_kernel	String	IP a la cual se deberá conectar con la Kernel
port_kernel	Numérico	Puerto al cual se deberá conectar con la Kernel
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
ip_filesystem	String	IP a la cual se deberá conectar con la Filesystem
port_filesystem	Numérico	Puerto al cual se deberá conectar con la Filesystem
scheme	String	Esquema de particiones de memoria a utilizar FIJAS / DINAMICAS
search_algorithm	String	Algoritmo de busqueda de huecos de memoria: FIRST / BEST / WORST
partitions	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 slog.SetLogLoggerLevel()

Ejemplo de Archivo de Configuración

```
{
    "port": 8002,
    "memory_size": 1024,
    "instruction_path": "/home/utnso/scripts-pruebas",
    "response_delay": 1000,
    "ip_kernel": "127.0.0.1",
    "port_kernel": 8001,
    "ip_cpu": "127.0.0.1",
    "port_cpu": 8004,
    "ip_filesystem": "127.0.0.1",
    "port_filesystem": 8006,
    "scheme": "FIJAS",
    "search_algorithm": "FIRST",
    "partitions": [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 iniciar el módulo se deberá validar que existan los archivos bitmap.dat y bloques.dat. En caso que no existan se deberán crear. Caso contrario se deberán tomar los ya existentes.

Para simular la latencia real de un File System, 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[3] 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:

```
{
    "index_block": 10,
    "size": 1024
}
```

[3] Se recomienda utilizar el tipo de dato uint32_t

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:

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 archivo de punteros y grabar todos los punteros 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 / INDICE> - Bloque File System <NUMERO_BLOQUE_FS>`"

Fin Petición: "`## Fin de solicitud - Archivo: <NOMBRE_ARCHIVO>`"

Archivo de configuración

Campo	Tipo	Descripción
port	Numérico	Puerto en el cual se escuchará la conexión de módulo.
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
mount_dir	String	Path a partir del cual van a encontrarse los archivos del FS.
block_size	Numérico	Tamaño de los bloques del FS

Campo	Tipo	Descripción
block_count	Numérico	Cantidad de bloques del FS
block_access_delay	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 slog.SetLogLoggerLevel()

Ejemplo de Archivo de Configuración

```
{
    "port": 8003,
    "ip_memory": "127.0.0.1",
    "port_memory": 8002,
    "mount_dir": "/home/utnso/mount_dir",
    "block_size": 64,
    "block_count": 1024,
    "block_access_delay": 15000,
    "log_level": "DEBUG"
}
```

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 la biblioteca estándar de Go, principalmente las funciones para slices, archivos de configuración y logs.
- Todos los módulos están creados y son capaces de exponer una API para poder comunicarse 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:**
  - Devuelve y actualiza contextos de ejecución
  - Para todas las demás peticiones responde OK en formato stub[4] (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 los demás peticiones responde OK en formato stub (Sin hacer nada)

Carga de trabajo estimada:

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

[4] 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.