Episodio IX

The Rise of Gopher

I've brought peace, freedom, justice and security to my new Operating System

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

-1C2025 -
Versión 1.3

Índice

Historial de Cambios

v1.0 (08/04/2025) Publicación del enunciado

v1.1 (15/04/2025) Detalle de cambios:

- *Se agrega argumento al iniciar el CPU*
- *Se explica cómo identificar los sockets desde el Kernel*
- *Se corrigen detalles de redacción que generaban confusión en planificadores*
- *Se ajustan configs*
- *Se eliminan estructuras innecesarias en memoria.*

v1.2 (01/05/2025) Detalle de cambios:

- *Se agregan argumentos faltantes en instrucciones de CPU*
- *Se agrega parámetro a la config del Kernel y de Memoria*
- *Se corrigen detalles de redacción en Kernel, CPU y Memoria*

v1.3 (04/06/2025) Detalle de cambios:

- *Se corrigen detalles de redacción en Kernel*
    - *Se aclara el comportamiento de Proceso más chico primero*
    - *Se aclara el comportamiento de NEW y SUSP. READY.*
    - *Se aclara el comportamiento esperado ante desconexión de IO.*
- *Se ajusta redacción en Memoria + SWAP*
    - *Se agrega aclaración del archivo de SWAP.*

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: 08/04/2025
- Fecha de primera entrega: 12/07/2025
- Fecha de segunda entrega: 19/07/2025
- Fecha de tercera entrega: 02/08/2025
- 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 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?

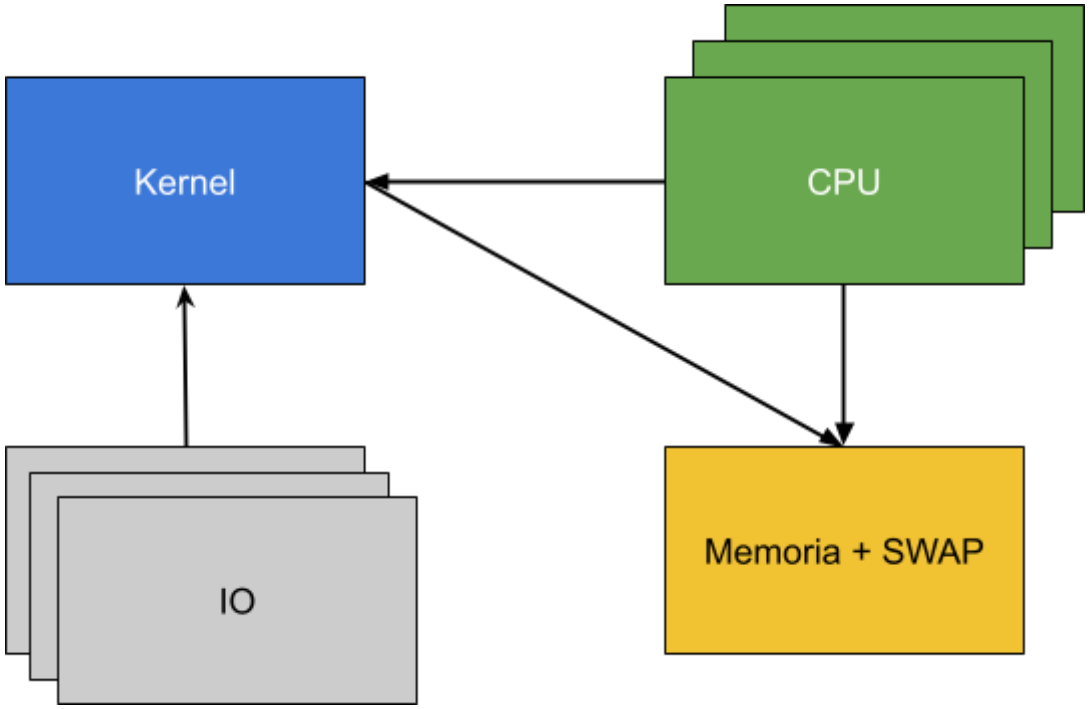
El objetivo del trabajo práctico consiste en desarrollar una solución que permita la simulación de un sistema distribuido, donde los grupos tendrán que planificar procesos, resolver peticiones al sistema y administrar de manera adecuada una memoria bajo los esquemas explicados en sus correspondientes 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 Kernel es necesario contar con la Memoria + SWAP iniciada.

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

Desarrollar únicamente temas de conectividad, serialización, sincronización o el módulo IO 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 no persistirlos en archivo, *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 módulo encargado de la gestión de los procesos que se creen dentro de este sistema simulado, planificando su ejecución en múltiples CPUs mediante diferentes algoritmos, llevando a cabo sus peticiones de entrada/salida hacia múltiples dispositivos (representados por el módulo I/O) e interactuando con el módulo Memoria para su utilización.

Lineamiento e Implementación

Para lograr su propósito, este módulo mantendrá conexiones efímeras vía API's con una o más instancias del módulo CPU y con el módulo Memoria.

Para la conexión con cada instancia del módulo CPU, se tendrán que consumir mínimamente 2 api's: la primera enviará los procesos a ejecutar y el Kernel esperará la respuesta en una API que él expondrá; y la segunda será utilizada para enviarle las notificaciones de interrupciones al momento de trabajar con algoritmos que requieran interrumpir a la CPU.

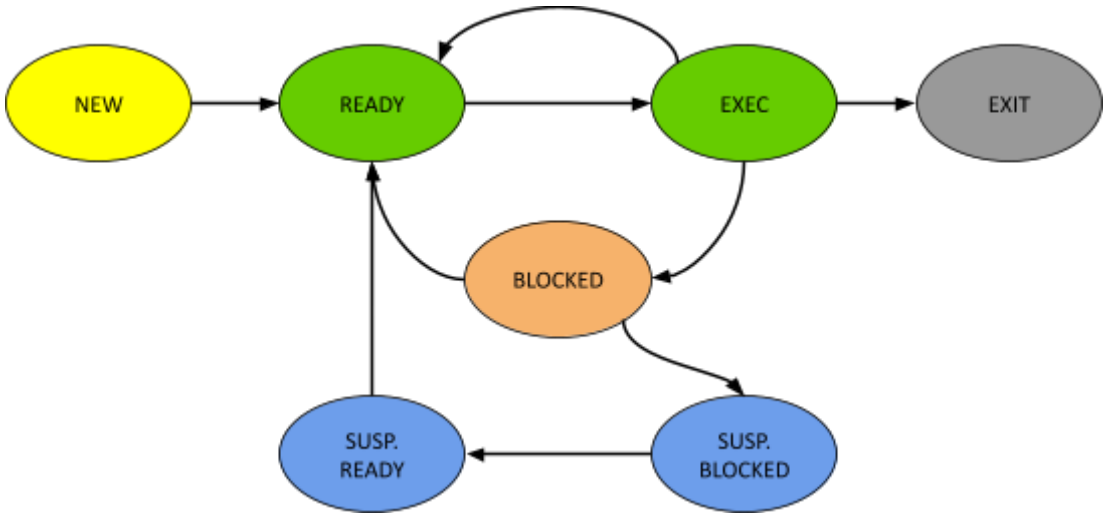
Para las conexiones con el módulo Memoria, el Kernel **deberá invocar distintas API's que la memoria exponga**, y dicho módulo deberá responder de forma **sincrónica** (es decir, en la respuesta a la solicitud)

Al iniciar el módulo, se creará un proceso inicial para que este lo planifique y para poder inicializarlo, se requerirá que este módulo reciba dos argumentos adicionales en la función main: el nombre del archivo de pseudocódigo que deberá ejecutar y el tamaño del proceso para ser inicializado en Memoria. Ejemplo de comando de ejecución:

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

Diagrama de estados

El kernel utilizará un diagrama de 7 estados para la planificación de los procesos.



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 que arranque en 0).
- **PC**: Program Counter, es un número entero que arranca en 0.
- **ME:** Métricas de Estado, un listado de estados contabilizando la cantidad de veces que el proceso estuvo en cada uno de ellos.
- **MT:** Métricas de tiempo por estado, un listado con los tiempos en milisegundos en los que permaneció el proceso en cada estado.

Planificador de Largo Plazo

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

Inicio

Al iniciar el proceso Kernel, el algoritmo de Largo Plazo debe estar frenado (estado STOP) y se deberá esperar un ingreso de un *Enter* por teclado para poder iniciar con la planificación.

Creación de procesos

Se tendrá una cola NEW que será administrada mediante un algoritmo definido por archivo de configuración. Estos algoritmos son: FIFO y Proceso mas chico primero (siendo mas chico el que menos memoria solicite).

Al llegar un nuevo proceso a esta cola y la misma esté vacía y no se tengan procesos en la cola de SUSP READY, se enviará un pedido a Memoria para inicializar el mismo. Si la respuesta es positiva, se pasará el proceso 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.

En cambio, si un proceso llega a esta cola y ya hay otros esperando se debe tener en cuenta el algoritmo definido y verificar si corresponde o no su ingreso, teniendo en cuenta que la cola de SUSP READY deberá estar vacía.

FIFO

Bajo el algoritmo FIFO, si hay un proceso en NEW que no puede pasar a ready por falta de espacio en la memoria, todos los procesos que lleguen a continuación, deberán esperar a que el primero ingrese.

Proceso más chico primero

En este algoritmo se va a priorizar a los procesos más chicos, por lo tanto, al llegar un nuevo proceso a NEW, independientemente de si hay procesos esperando, si el nuevo proceso fuera el de menor tamaño vamos a consultarle a la memoria si dispone del espacio requerido para iniciar el nuevo proceso y en caso afirmativo, se transiciona el proceso a READY. Caso contrario, se encolará para seguir esperando junto al resto de los procesos que no pueden ingresar por tamaño

Al liberarse espacio en la memoria, se deberá validar si se pueden ingresar procesos a READY ordenándolos por tamaño de manera ascendente.

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 o más de los que estén esperando, para ello primero se deberá verificar los procesos que estén en estado SUSP. READY y luego se verificará el estado NEW en caso de que la cola de SUSP. READY se encuentre vacía, solo se podrán pasar procesos de NEW a READY cuando la cola de SUSP. READY se encuentre vacía.

Luego de la finalización, se debe loguear las métricas de estado con el formato adecuado.

Planificador de Corto Plazo

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

- FIFO
- SJF sin Desalojo
- SJF con Desalojo (Shortest Remaining Time)

FIFO

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

SJF sin Desalojo

Se elegirá el proceso que tenga la rafaga más corta. Su funcionamiento será como se explica en las clases. Para la primera ráfaga de todos los procesos se utilizará una estimación inicial definida por archivo de configuración, la fórmula que se utilizará para calcular las siguientes ráfagas es:

$Est(n) = Estimado\ de\ la\ ráfaga\ anterior$
$R(n) = Lo\ que\ realmente\ ejecutó\ de\ la\ ráfaga\ anterior\ en\ la\ CPU$
$Est(n+1) = El\ estimado\ de\ la\ próxima\ ráfaga$

$$Est(n+1) = \alpha\, R(n) + (1-\alpha)\, Est(n) \;\; ; \;\; \alpha \in [0,1]$$

SJF con Desalojo

Funciona igual que el anterior con la variante que, al ingresar un proceso en la cola de Ready y no haber CPUs libres, se debe evaluar si dicho proceso tiene una rafaga más corta que los que se encuentran en ejecución. En caso de ser así, se debe informar al CPU que posea al Proceso con el tiempo restante más alto que debe desalojar al mismo para que pueda ser planificado el nuevo.

Ejecución

Una vez seleccionado el proceso a ejecutar, se lo transicionará al estado EXEC y se enviará a uno de los módulos CPU (conectados y libres) el PID y el PC a ejecutar a través del endpoint de dispatch, quedando a la espera de recibir dicho PID y PC después de la ejecución junto con un motivo por el cual fue devuelto.

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

Al recibir el PID del proceso en ejecución, en caso de que el motivo de devolución implique replanificar, se seleccionará el siguiente proceso 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 las instrucciones INIT_PROC y EXIT.

INIT_PROC, esta syscall recibirá 2 parámetros de la CPU, el primero será el nombre del archivo de pseudocódigo que deberá ejecutar el proceso y el segundo parámetro es el tamaño del proceso en Memoria. El Kernel creará un nuevo PCB y lo dejará en estado NEW, esta syscall no implica cambio de estado, por lo que el proceso que llamó a esta syscall, inmediatamente volverá a ejecutar en la CPU.

EXIT, esta syscall no recibirá parámetros y se encargará de finalizar el proceso que la invocó, siguiendo lo descrito anteriormente para *Finalización de procesos*.

Memoria

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

Esta syscall bloqueará al proceso 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, se desbloquea normalmente pasando a READY.

Entrada Salida

El módulo Kernel administra los distintos dispositivos de IO que se conecten a él. Al momento de que un módulo de IO se conecte al Kernel (mediante una API), este último deberá identificarlo por medio de su nombre, en este punto el Kernel deberá conocer todos los módulos de IO conectados (recibiendo su IP y PUERTO), qué procesos están ejecutando IO en cada módulo y todos los procesos que están esperando una IO determinada.

Para la utilización de estos módulos existirá la syscall "**IO**" que recibirá dos parámetros: el nombre de la entrada/salida a usar y la cantidad de milisegundos que deberá usarla. Primero se deberá validar que la IO solicitada exista en el sistema, si no existe ninguna IO en el sistema con el nombre solicitado, el proceso se deberá enviar a EXIT. En caso de que si exista al menos una instancia de IO, aun si la misma se encuentre ocupada, el kernel deberá pasar el proceso al estado BLOCKED y agregarlo a la cola de bloqueados por la IO solicitada, en caso de que el kernel tenga una instancia de esta IO libre, deberá enviarle al módulo correspondiente el PID y el tiempo por el cual debe bloquearse. Al momento que se conecte una nueva IO o se reciba el desbloqueo por medio de una de ellas, se deberá verificar si hay proceso encolados para dicha IO y enviarlo a la misma.

Al momento de recibir un mensaje de una IO se deberá verificar que el mismo sea una confirmación de fin de IO, en caso afirmativo, se deberá validar si hay más procesos esperando realizar dicha IO. En caso de que el mensaje corresponda a una desconexión de la IO, el proceso que estaba ejecutando en dicha IO, se deberá pasar al estado EXIT. Si hay procesos encolados y no quedan más instancias disponibles, también se deberá pasar a EXIT.[1]

Planificador de Mediano Plazo

Al ingresar un proceso al estado BLOCKED se deberá iniciar un timer el cual se encargará de esperar un tiempo determinado por archivo de configuración, al terminar ese tiempo si el proceso continúa en estado BLOCKED, él mismo deberá transicionar al estado SUSP. BLOCKED. En este momento se debe informar al módulo memoria que debe ser movido de memoria principal a swap. Cabe aclarar que en este momento vamos a tener más memoria libre en el sistema por lo que se debe verificar si uno o más nuevos procesos pueden entrar (tanto de la cola NEW como de SUSP. READY).

Cuando el módulo IO informe que el proceso en SUSP. BLOCKED finalizó su IO correspondiente, el mismo deberá cambiar al estado SUSP. READY y deberá quedar a la espera de su oportunidad de pasar al estado READY.[2]

Una vez que el proceso llegue a SUSP. READY tendrá el mismo comportamiento, es decir, utilizará el mismo algoritmo que la cola NEW teniendo más prioridad que esta última. De esta manera, ningún proceso que esté esperando en la cola de NEW podrá ingresar al sistema si hay al menos un proceso en SUSP. READY.

---

[1] Para notificar desde el módulo IO al módulo Kernel se deberá implementar el Handler de la señal de finalización del proceso. Para esto, recomendamos investigar Signal Notify, con código de ejemplo "signal.Notify(sigs, syscall.SIGTERM, syscall.SIGINT)"

[2] Recuerden que las API's tienen una vida útil (timeout) por lo que debe existir una API del módulo IO para recibir el pedido y una API del módulo Kernel para recibir la respuesta del mismo.

Consideraciones teóricas del esquema

Los esquemas de planificación aquí planteados son arbitrarios a modo de evaluación académica y no necesariamente reflejan lo que un planificador de largo, mediano o corto plazo implementaría en un sistema real para optimizar sus objetivos.

Logs mínimos y obligatorios

Syscall recibida: "`## (<PID>) - Solicitó syscall: <NOMBRE_SYSCALL>`"

Creación de Proceso: "`## (<PID>) Se crea el proceso - Estado: NEW`"

Cambio de Estado: "`## (<PID>) Pasa del estado <ESTADO_ANTERIOR> al estado <ESTADO_ACTUAL>`"

Motivo de Bloqueo: "`## (<PID>) - Bloqueado por IO: <DISPOSITIVO_IO>`"

Fin de IO: "`## (<PID>) finalizó IO y pasa a READY`"

Desalojo de SJF/SRT: "`## (<PID>) - Desalojado por algoritmo SJF/SRT`"

Fin de Proceso: "`## (<PID>) - Finaliza el proceso`"

`Métricas de Estado`: "`## (<PID>) - Métricas de estado: NEW (NEW_COUNT) (NEW_TIME), READY (READY_COUNT) (READY_TIME), ...`"

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
scheduler_algorithm	String	FIFO/SJF/SRT
ready_ingress_algorithm	String	FIFO/PMCP
alpha	Numérico	Valor del alfa para utilizar en la estimación SJF
initial_estimate	Numérico	Estimación inicial del primer proceso del Kernel
suspension_time	Numérico	Tiempo en milisegundos que se debe esperar antes de mover un proceso al estado SUSP. BLOCKED.
log_level	String	Nivel de detalle máximo a mostrar. Compatible con slog.SetLogLoggerLevel()
port_kernel	Numérico	Puerto en el cual se levantará el kernel
ip_kernel	String	IP del Kernel

Ejemplo de Archivo de Configuración

```
{
    "ip_memory": "127.0.0.1",
    "port_memory": 8002,
    "ip_kernel": "127.0.0.1",
    "port_kernel": 8001,
    "scheduler_algorithm": "FIFO",
    "ready_ingress_algorithm": "PMCP",
    "alpha": 0.5,
    "initial_estimate": 10000,
    "suspension_time": 4500,
    "log_level": "DEBUG"
}
```

Módulo: IO

El módulo IO, en nuestro contexto de TP, va a ser el encargado de simular los diferentes dispositivos de IO que puedan existir en el Sistema.

Lineamiento e Implementación

El módulo **IO** es el encargado de simular las operaciones de Entrada/Salida, para ello al iniciar deberá leer como parámetro de entrada el nombre el cual identificará a la interfaz de IO.

```
➜ ~ ./bin/io [nombre]
```

Una vez leído el nombre, se conectará al Kernel y en el handshake inicial le enviará su nombre, ip y puerto para que él lo pueda identificar y quedará esperando las peticiones del mismo.

Al momento de recibir una petición del Kernel, el módulo deberá iniciar un usleep por el tiempo indicado en la request.

Al finalizar deberá informar al Kernel que finalizó la solicitud de I/O y quedará a la espera de la siguiente petición.

Finalización del Módulo IO

El Módulo IO, deberá notificar al Kernel de su finalización, para esto se deberá implementar el manejo de las señales SIGINT y SIGTERM, para enviar la notificación y finalizar de manera controlada.

Para esto, recomendamos investigar Signal Notify, con código de ejemplo “signal.Notify(sigs, syscall.SIGTERM, syscall.SIGINT)

Logs mínimos y obligatorios

Inicio de IO: “`## PID: <PID> - Inicio de IO - Tiempo: <TIEMPO_IO>`”.

Finalización de IO: “`## PID: <PID> - Fin de IO`”.

Archivo de Configuración

Campo	Tipo	Descripción
port_io	Numérico	Puerto en el cual se levantará el IO
ip_io	String	IP del módulo IO
ip_kernel	String	IP a la cual se conectará con el módulo Kernel
port_kernel	Numérico	Puerto al cual se conectará con el módulo Kernel

log_level	String	Nivel de detalle máximo a mostrar. Compatible con slog.SetLogLoggerLevel()

Ejemplo de Archivo de Configuración

```
{
    "ip_kernel": "127.0.0.1",
    "port_kernel": 8001,
    "port_io": 8003,
    "ip_io": "127.0.0.1",
    "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

Al momento de ejecutar un CPU se debe indicar por argumento cuál es su "identificador" para poder identificarlo entre los demás.

```
➜ ~ ./bin/cpu [identificador]
```

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

Para cada CPU se deberá tener un archivo de log independiente que deberá contar con el identificador pasado por parámetro para poder identificar a qué CPU corresponde el archivo de log que se está leyendo.

Las CPUs deberán conectarse al Kernel enviándoles en el handshake su IP, su PUERTO y su identificador para que él pueda luego identificar y enviar los datos correctamente.

Al momento de recibir un **PID y un PC** de parte del Kernel la CPU deberá solicitarle a la Memoria la siguiente instrucción 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.

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

```
NOOP
WRITE 0 EJEMPLO_DE_ENUNCIADO
READ 0 20
GOTO 0
IO IMPRESORA 25000
INIT_PROC proceso1 256
DUMP_MEMORY
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:

- **NOOP**: Representa la instrucción No Operation, es decir, va a consumir solamente el tiempo del ciclo de instrucción.
- **WRITE** (Dirección, Datos)**:** Escribe los datos del parámetro datos en la dirección física obtenida a partir de la dirección lógica que se encuentra en el parámetro Dirección, el campo datos será siempre una cadena de caracteres **sin espacios**.
- **READ** (Dirección, Tamaño): Lee el valor de memoria correspondiente a la dirección física obtenida a partir de la dirección lógica que se encuentra en el parámetro Dirección, de un tamaño determinado por el parámetro Tamaño y lo imprime por pantalla y en el **Log Obligatorio** correspondiente.
- **GOTO** (Valor): Actualiza el PC o Program Counter del proceso al valor pasado por parámetro.

Las siguientes instrucciones se considerarán *Syscalls*, ya que las mismas no pueden ser resueltas por la CPU y depende de la acción del Kernel para su realización, a diferencia de la vida real donde la llamada es a una única instrucción, para simplificar la comprensión de los scripts, vamos a utilizar un nombre diferente para cada *Syscall*.

- **IO** (Dispositivo, Tiempo)
- **INIT_PROC** (Archivo de instrucciones, Tamaño)
- **DUMP_MEMORY**
- **EXIT**

Es importante tener en cuenta que al finalizar el ciclo de instrucción el Program Counter (PC) deberá ser actualizado sumándole 1 en caso de que este no haya sido modificado por la instrucción GOTO.

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 PID y el Program Counter (PC) actualizado 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 es de Paginación Multi-nivel, por lo tanto, las direcciones lógicas se compondrán de la siguiente manera:

```
[entrada_nivel_1 | entrada_nivel_2 | ... | entrada_nivel_X |
                                        desplazamiento]
```

Estas traducciones, a diferencia de lo que se hace en los ejercicios prácticos que se ven en clases y se toman en los parciales, no se realizarán en binario, ya que es más cómodo utilizar números enteros en sistema decimal para ver los valores, y tomando en cuenta que todas las tablas de todos los niveles tendrán la misma cantidad de entradas, la operatoria sería más parecida a la siguiente.

Teniendo una cantidad de niveles **N** y un identificador **X** de cada nivel podemos utilizar las siguientes fórmulas:

nro_página = floor(dirección_lógica / tamaño_página)
entrada_nivel_X = floor(nro_página / cant_entradas_tabla ^ (**N** - **X**)) % cant_entradas_tabla
desplazamiento = dirección_lógica % tamaño_página

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.

La TLB contará con la siguiente estructura base: `[ página | marco ]`, pudiendo agregar campos 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 o 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.

Al momento de desalojar un proceso, todas las páginas que se encuentran en la TLB se deberán eliminar.

Caché de páginas

Dado que esta CPU es una simulación de una CPU real, en lugar de tener un tamaño definido en MB y diferentes niveles de Cache (ya que en los procesadores reales tenemos Cache L1, L2, L3, etc), para nuestro trabajo práctico vamos a tener una versión simplificada, donde la caché de páginas va a estar definida por una cantidad de páginas que se podrán almacenar en esta caché. La estructura básica para esta cache deberá ser `[ página | contenido ]`, pudiendo agregar campos extra para facilitar la implementación de los algoritmos.

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

Al momento de acceder a una página, la CPU deberá validar si la Caché de páginas se encuentra habilitada (tiene al menos 1 frame) y en caso de que así sea, no debe acceder directamente a las páginas en memoria, si no que deberá hacer todas las operaciones en su Caché, mientras que si la misma está deshabilitada, ahí sí deberá realizar todas las operaciones de lectura y escritura directamente en la memoria principal.

Al momento de cargar una página en la Caché, puede darse el caso de que la caché se encuentre llena y por lo tanto se deberá seleccionar una víctima a ser reemplazada, al momento de reemplazar una página, si la misma fue modificada mientras estuvo cargada en caché, los cambios deberán enviarse a la memoria principal para que esta última tenga la versión actualizada de los datos.

Al momento de desalojar un proceso, las páginas que se encuentran modificadas en caché, deberán actualizarse en la memoria principal. Para esto, primero consultará sus direcciones físicas, luego enviará a escribir su contenido, y finalmente se eliminarán todas las entradas de la caché.

De esta manera, al momento de acceder a una página primero se deberá verificar la caché de páginas, luego la TLB y como última instancia la tabla de páginas en memoria.

Logs mínimos y obligatorios

Fetch Instrucción: “`## PID: <PID> - FETCH - Program Counter: <PROGRAM_COUNTER>`”.

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

Instrucción Ejecutada: “`## PID: <PID> - Ejecutando: <INSTRUCCION> - <PARAMETROS>`”.

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

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

TLB Hit: “`PID: <PID> - TLB HIT - Pagina: <NUMERO_PAGINA>`”

TLB Miss: “`PID: <PID> - TLB MISS - Pagina: <NUMERO_PAGINA>`”

Página encontrada en Caché: “`PID: <PID> - Cache Hit - Pagina: <NUMERO_PAGINA>`”

Página faltante en Caché: “`PID: <PID> - Cache Miss - Pagina: <NUMERO_PAGINA>`”

Página ingresada en Caché: “`PID: <PID> - Cache Add - Pagina: <NUMERO_PAGINA>`”

Página Actualizada de Caché a Memoria: “`PID: <PID> - Memory Update - Página: <NUMERO_PAGINA> - Frame: <FRAME_EN_MEMORIA_PRINCIPAL>`”

Archivo de Configuración

Campo	Tipo	Descripción
port_cpu	Numérico	Puerto en el cual se levantará el CPU
ip_cpu	String	IP de la 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
ip_kernel	String	IP a la cual se deberá conectar con el Kernel
port_kernel	Numérico	Puerto en el cual se conectará al Kernel
tlb_entries	Numérico	Cantidad de entradas de la TLB
tlb_replacement	String	Algoritmo de reemplazo para las entradas de la TLB. FIFO o LRU.
cache_entries	Numérico	Cantidad de entradas de la Caché de páginas
cache_replacement	String	Algoritmo de reemplazo para las entradas de la Caché de páginas. CLOCK o CLOCK-M.
cache_delay	Numérico	Tiempo en milisegundos que se deberá esperar antes de responder a las solicitudes.
log_level	String	Nivel de detalle máximo a mostrar. Compatible con slog.SetLogLoggerLevel()

Ejemplo de Archivo de Configuración

```
{
    "port_cpu": 8004,
    "ip_cpu": "127.0.0.1",
    "ip_memory": "127.0.0.1",
    "port_memory": 8002,
    "ip_kernel": "127.0.0.1",
    "port_kernel": 8002,
    "tlb_entries": 15,
    "tlb_replacement": "TLB",
    "cache_entries": 10,
    "cache_replacement": "CLOCK",
    "cache_delay": 10,
    "log_level": "DEBUG"
}
```

Módulo: Memoria + SWAP

Este módulo será el encargado de responder los pedidos realizados por la CPU (de manera síncrona) para leer y/o escribir en una tabla de páginas, en una porción del espacio de usuario de la memoria o para administrar el pseudocódigo de los procesos que se ejecutarán en el sistema.

A su vez, será el encargado de manejar un espacio de SWAP que se encontrará abstraído en un único archivo para todos los procesos, de forma tal que se le permite al mismo manejar un mayor nivel de direccionamiento.

Lineamiento e Implementación

Al iniciar levantará un servidor *multihilo* para esperar las peticiones por parte del módulo Kernel y de las múltiples CPU.

Memoria de Sistema

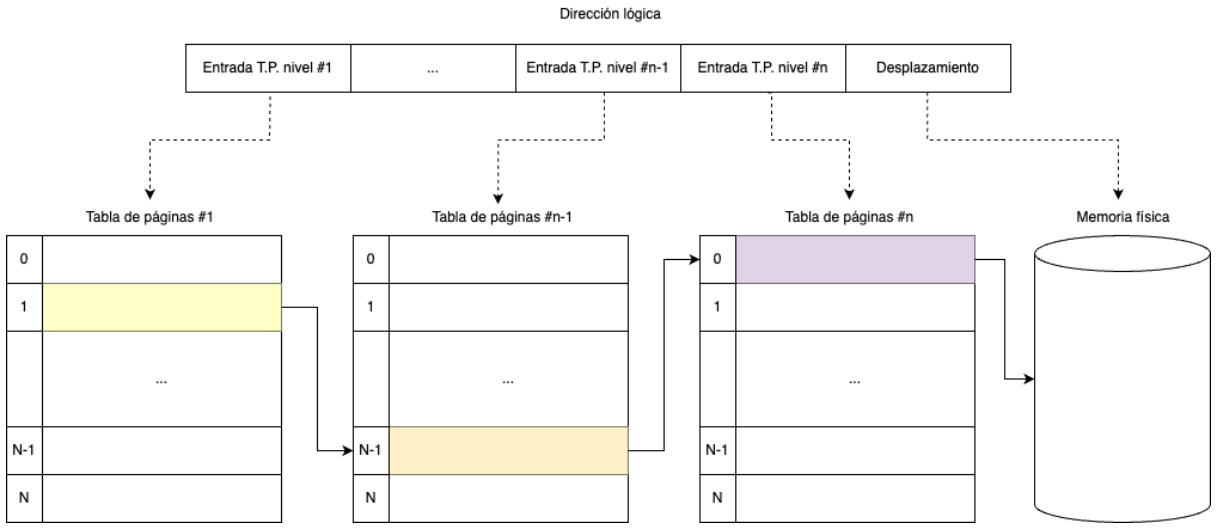
Archivos de pseudocódigo

Por cada PID 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

La asignación de memoria de este trabajo práctico utilizará un esquema de **paginación jerárquica multinivel**, donde las tablas de cada nivel contarán con la misma cantidad de entradas, definida por archivo de configuración. Así mismo, la cantidad de niveles de paginación también será definida por archivo de configuración.



Consideraciones teóricas del esquema

Como se aclara al comienzo de este documento, los conceptos reflejados son, en general, versiones simplificadas o alteradas de los componentes reales de hardware y de sistemas operativos vistos en las clases. Particularmente para este esquema planteado, en sistemas reales normalmente se ve combinado con Memoria Virtual, donde se manifiestan sus principales ventajas. Por otro lado, la cantidad de niveles de paginación normalmente no es libremente configurable sino que depende de la especificación de cada Procesador e implementación del Sistema Operativo.
Este trabajo simplifica el esquema al no combinarlo con Memoria Virtual para enfocarse solamente en sus estructuras y principales diferencias o desventajas con respecto a paginación de un nivel.
Invitamos al alumnado a investigar sobre este esquema en sistemas reales, tanto en el ámbito académico como en procesadores y sistemas operativos modernos, así como reflexionar y discutir acerca de sus beneficios para el aprovechamiento de la memoria.

Estructuras

La memoria contará principalmente con 4 estructuras:

- Un espacio contiguo de memoria (representado por un **array de bytes**[3]). Este representará el espacio de usuario de la misma, donde los procesos podrán leer y/o escribir.
- Las tablas de páginas, que representarán el espacio de Kernel.
- Un archivos SWAP denominado `swapfile.bin`.
- Listado de métricas por proceso

Métricas por proceso

El módulo memoria llevará un listado de métricas de cada proceso a fin de poder dar seguimiento a la cantidad de operaciones realizadas por cada proceso en la memoria, por lo tanto el Listado de métricas por proceso tendrá los siguientes valores:

- Cantidad de accesos a Tablas de Páginas
- Cantidad de Instrucciones solicitadas
- Cantidad de bajadas a SWAP
- Cantidad de subidas a Memoria Principal o al espacio contiguo de memoria
- Cantidad de Lecturas de memoria
- Cantidad de Escrituras de memoria

Estas métricas serán informadas al finalizar/destruir un proceso de la memoria principal. La forma de informarlas será por medio de los logs obligatorios con el formato indicado en dicha sección.

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

Comunicación con Kernel y CPU

Inicialización del proceso

En caso de que se cuente con el tamaño necesario, el módulo memoria deberá crear las estructuras administrativas necesarias y enviar como *respuesta* un mensaje de 'OK'

Suspensión de proceso

Al ser suspendido un proceso, se debe liberar su espacio en memoria, escribiendo en SWAP solamente la información necesaria. Se debe tener en cuenta que, para la realización de este trabajo práctico, no se requiere el swapeo de tablas de páginas.

Des-suspensión de proceso

En caso de que se cuente con el tamaño necesario, el módulo memoria deberá leer el contenido del archivo SWAP correspondiente al proceso a des-suspender, escribirlo en los marcos asignados, liberar el espacio en SWAP, actualizar las estructuras administrativas necesarias y, por último, enviar como respuesta un mensaje de 'OK'.

Finalización de proceso

Al ser finalizado un proceso, se debe liberar su espacio de memoria y marcar como libres sus entradas en SWAP.

A su vez, se deberá generar el log obligatorio con las métricas del proceso antes definidas.

Acceso a tabla de páginas

El módulo deberá responder con el número de marco correspondiente. En este evento se deberá tener en cuenta la cantidad de niveles de tablas de páginas accedido, debiendo considerar un acceso (con su respectivo conteo de métricas y retardo de acceso) por cada nivel de tabla de páginas accedido.

Acceso a espacio de usuario

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. En caso satisfactorio se responderá un mensaje de 'OK'.

Leer Página completa

Se deberá devolver el contenido correspondiente de la página a partir del byte enviado como dirección física dentro de la Memoria de Usuario, que deberá coincidir con la posición del byte 0 de la página.

Actualizar página completa

Se escribirá la página completa a partir del byte 0 que igual será enviado como *dirección física*, esta operación se realizará dentro de la Memoria de Usuario y se responderá como OK.

Memory Dump

Al momento de recibir la operación de memory dump el módulo memoria deberá crear 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 mismo.

El archivo debe llamarse "<PID>-<TIMESTAMP>.dmp" dentro del path definido por archivo de configuración.

Manejo de SWAP

Como se mencionó anteriormente el submódulo de SWAP va a ser el encargado dentro de la memoria de manejar las peticiones de lectura y escritura en SWAP.

Vamos a contar con un archivo denominado *swapfile.bin*, cuya ubicación se va a encontrar definida por archivo de configuración. Este archivo se utilizará para la suspensión de los procesos y contendrá la contenido de las páginas del proceso que fue suspendido, es responsabilidad del grupo definir la o las estructuras necesarias para poder saber donde se encuentran las páginas de cada proceso en SWAP ya que las mismas en algún momento deberán volver a cargarse en memoria principal para reanudar la ejecución del proceso.

Es importante aclarar que al momento de suspender un proceso, bajo el contexto simplificado de este trabajo, no es necesario que se guarden en el archivo SWAP sus tablas de páginas ni ninguna otra estructura administrativa relacionada al proceso suspendido.

Logs mínimos y obligatorios

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

Destrucción de Proceso: "`## PID: <PID> - Proceso Destruido - Métricas - Acc.T.Pag: <ATP>; Inst.Sol.: <Inst.Sol.>; SWAP: <SWAP>; Mem.Prin.: <Mem.Prin.>; Lec.Mem.: <Lec.Mem.>; Esc.Mem.: <Esc.Mem.>`"

Obtener instrucción: "`## PID: <PID> - Obtener instrucción: <PC> - Instrucción: <INSTRUCCIÓN> <...ARGS>`"

Escritura / lectura en espacio de usuario[4]**:** "`## PID: <PID> - <Escritura/Lectura> - Dir. Física: <DIRECCIÓN_FÍSICA> - Tamaño: <TAMAÑO>`"

Memory Dump: "`## PID: <PID> - Memory Dump solicitado`"

[4] En el caso de leer o escribir una página entera el tamaño es el tamaño de página y la dirección es el byte 0 de la página.

Archivo de Configuración

Campo	Tipo	Descripción
port_memory	Numérico	Puerto al cual se deberá conectar con la memoria
ip_memory	String	IP de la Memoria
memory_size	Numérico	Tamaño expresado en bytes del espacio de usuario de la memoria.
page_size	Numérico	Tamaño de las páginas en bytes.
entries_per_page	Numérico	Cantidad de entradas de cada tabla de páginas.
number_of_levels	Numérico	Cantidad de niveles de tablas de páginas
memory_delay	Numérico	Tiempo en milisegundos que debera esperarse antes de responder una petición de memoria
swapfile_path	String	Path donde se encuentra el archivo de swapfile.bin
swap_delay	Numérico	Tiempo en milisegundos que deberá esperarse antes de responder una petición de SWAP
log_level	String	Nivel de detalle máximo a mostrar. Compatible con slog.SetLogLoggerLevel()
dump_path	String	Path donde se almacenarán los archivos de DUMP
scripts_path	String	Path donde se almacenarán los scripts

Ejemplo de Archivo de Configuración

```
{
    "port_memory": 8002,
    "ip_memory": "127.0.0.1",
    "memory_size": 4096,
    "page_size": 64,
    "entries_per_page": 4,
    "number_of_levels": 5,
    "memory_delay": 1500,
    "swapfile_path": "/home/utnso/swapfile.bin",
    "swap_delay": 15000,
    "log_level": "DEBUG",
    "dump_path": "/home/utnso/dump_files/",
    "scripts_path": "/home/utnso/scripts/"
}
```

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: 19/04/2025

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: Ejecución Básica

Fecha: 25/05/2025

Objetivos:

- **Módulo Kernel:**
  - Planificación de corto y largo plazo con FIFO
  - Administración de IO
- **Módulo CPU:**
  - Interpretación de instrucciones y ejecución de instrucciones
- **Módulo Memoria:**
  - Devolver lista de instrucciones
  - Devolver un valor fijo de espacio libre (mock)
- **Módulo IO: Completo**

Carga de trabajo estimada:

- **Módulo Kernel:** 40%
- **Módulo CPU:** 30%
- **Módulo Memoria:** 15%
- **Módulo IO**: 15%

Check de Control Obligatorio 3: CPU Completa y Memoria

Fecha: 21/06/2025

Objetivos:

- **Módulo Kernel:**
  - Planificación de corto y largo plazo completa.
- **Módulo CPU: Completo**
- **Módulo Memoria:**
  - Esquema de Memoria principal completo
  - Administración de espacio libre en memoria principal

Carga de trabajo estimada:

- **Módulo Kernel:** 30%
- **Módulo CPU:** 30%
- **Módulo Memoria:** 40%

Entregas Finales

Fechas: 12/07/2025, 19/07/2025, 02/08/2025

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.