UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Ingeniería en Sistemas de Información

ESTheR:
El Stack The Revenge

Porque las primeras partes no siempre son mejores

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

- 1C2017 -
Versión 1.0

Introducción

El trabajo práctico consiste en simular ciertos aspectos de un sistema multiprocesador, con la capacidad de interpretar la ejecución de scripts escritos en un lenguaje diseñado para el trabajo práctico. Este sistema planificará y ejecutará estos scripts (en adelante "Programas") controlando sus solicitudes de memoria, administrando los accesos a recursos tanto propios como compartidos.

Los scripts utilizados en el trabajo práctico estarán escritos en el lenguaje AnSISOP, el cual fue inventado y diseñado por la cátedra para fines didácticos. En el Anexo II encontrarán la especificación de este lenguaje. Se recomienda realizar una lectura detenida del mismo.

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 (API) que brindan los sistemas operativos.
- Entienda aspectos del diseño de un sistema operativo.
- Afirme diversos conceptos teóricos de la materia mediante la implementación práctica de algunos de ellos.
- Se familiarice con técnicas de programación de sistemas, como el empleo de makefiles, archivos de configuración y archivos de log.
- Conozca con grado de detalle la operatoria de Linux mediante la utilización de un lenguaje de programación de relativamente bajo nivel como C.

Características

- Modalidad: grupal (5 integrantes +- 0) y obligatorio
- Tiempo estimado para su desarrollo: 90 días
- Fecha de comienzo: 01/04/2017
- Fecha de entrega: 08/07/2017
- Fecha de primer recuperatorio: 15/07/2017
- Fecha de segundo recuperatorio: 29/07/2017
- Lugar de corrección: Laboratorio de Medrano

Evaluación del Trabajo Práctico

El trabajo práctico constará 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 realizar sus pruebas 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.

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 modernos, a fin de resaltar algún aspecto de diseño. En algunos casos los aspectos no fueron tomados de manera literal, por lo que invitamos a los alumnos a leer las notas y comentarios al respecto que haya en el enunciado, como así también a reflexionar y discutir con sus compañeros, ayudantes y docentes al respecto.

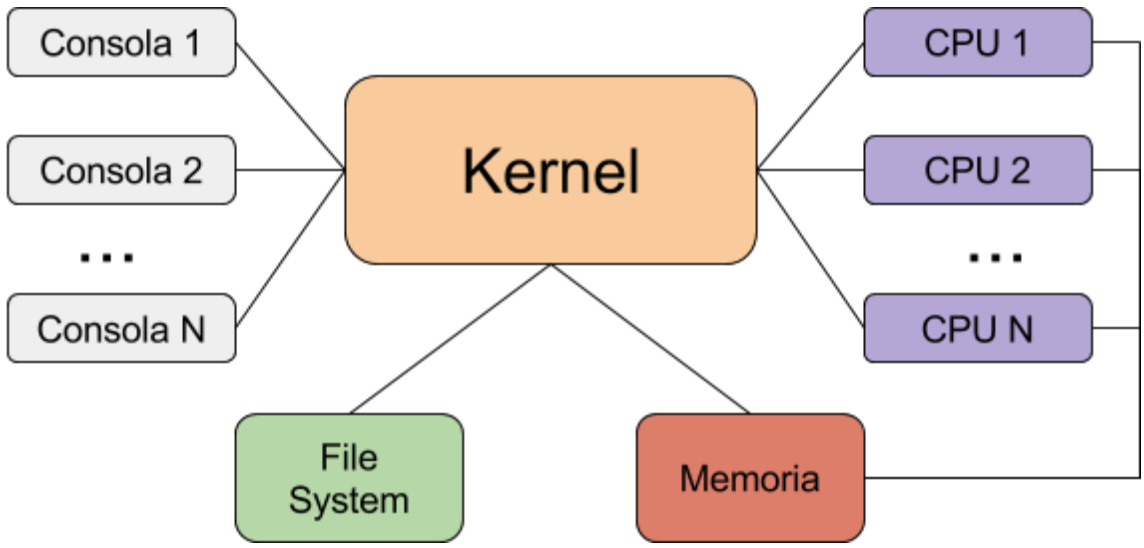
Arquitectura del sistema

El sistema simulará un sistema distribuido para el procesamiento de los Programas escritos en lenguaje de scripting AnSISOP. El mismo contará con un proceso central o Kernel, el cual se encargará de la planificación de dichos programas en la diferentes CPUs.

La información relacionada al contexto de ejecución de cada programa y las regiones dedicadas al sistema operativo para su gestión se almacenarán en un proceso denominado Memoria, que simula la memoria principal de una computadora.

Un proceso llamado Filesystem otorgará al sistema la capacidad de almacenar y gestionar datos de manera persistente.

Por último los procesos consola serán los encargados de enviar a ejecutar los programas AnSISOP al Kernel.



Proceso Consola

El proceso consola es el proceso encargado de enviar a ejecutar los Programas AnSISOP así como también funcionar como su interfaz tanto para recibir los mensajes que el programa necesite imprimir por pantalla, como los diferentes resultados de su ejecución. El proceso consola será capaz de ejecutar más de 1 programa de manera concurrente, por lo que los alumnos deberán idear un sistema para imprimir los mensajes enviados desde el código AnSISOP hacia la consola.

El proceso consola, al finalizar un Programa AnSISOP, deberá detallar los siguientes datos por pantalla:

- Fecha y hora de inicio de ejecución
- Fecha y hora de fin de ejecución
- Cantidad de impresiones por pantalla
- Tiempo total de ejecución (diferencia entre tiempo de inicio y tiempo de fin)

Arquitectura del proceso Consola

El proceso consola constará de una serie de hilos:

Interfaz de Usuario

Es el hilo principal y se encargá de la gestión de todo el proceso consola. Este hilo deberá poder ejecutar los siguientes comandos. Queda a criterio del grupo como implementarlos de manera tal que sea amigable para su uso:

- **Iniciar Programa:** Este comando iniciará un nuevo Programa AnSISOP, recibiendo por parámetro el path del script AnSISOP a ejecutar. Una vez iniciado el programa la consola quedará a la espera de nuevos comandos, pudiendo ser el iniciar nuevos Programas AnSISOP o algunas de las siguientes opciones. Quedará a decisión del grupo utilizar paths absolutos o relativos y deberán fundamentar su elección.
- **Finalizar Programa:** Como su nombre lo indica este comando finalizará un Programa AnSISOP, terminando el thread correspondiente al PID que se desee finalizar.
- **Desconectar Consola:** Este comando finalizará la conexión de todos los threads de la consola con el kernel, dando por muertos todos los programas de manera abortiva.
- **Limpiar Mensajes:** Este comando eliminará todos los mensajes de la pantalla.

Hilo Programa

Es el hilo encargado de iniciar y enviar a ejecutar Programas AnSISOP, visualizar los mensajes que éste imprima, prestando atención a que constantemente deberá tener un registro del PID del proceso que está ejecutando a fin de poder identificar los mensajes que imprime.

Por cada vez que se inicie un programa desde la Interfaz de la consola la misma deberá crear un hilo programa para atender las peticiones anteriormente descritas.

El PID (Process ID) será un identificador **único** en todo el sistema para cada programa y el mismo será proporcionado por el proceso Kernel.

Al iniciar, leerá su archivo de configuración, se conectará mediante ***sockets***[1] al Proceso Kernel y,

[1] Siempre que en el enunciado se lea la palabra socket, se refiere a los sockets STREAM tipo AF_INET

luego de un intercambio de mensajes inicial (*handshake*[2]), enviará el código del Programa AnSISOP al Kernel y obtendrá del Kernel el PID del proceso. A partir de ese momento, el hilo quedará a la espera de mensajes del Kernel correspondientes a las sentencias ***imprimir***[3], con los valores que deberá mostrar en pantalla.

La terminación del hilo Programa implica la finalización de la ejecución del Programa asociado a dicho hilo en el sistema.

Archivo de Configuración

Al iniciar, el Proceso Consola deberá leer los siguientes parámetros de un archivo de configuración, cuya ruta se indicará como argumento de la línea de comandos del programa.

Parámetro	Tipo de dato	Descripción
IP_KERNEL	[ip]	Dirección IP del proceso Kernel
PUERTO_KERNEL	[numérico]	Puerdo del proceso Kernel

Ejemplo de Archivo de Configuración

```
IP_KERNEL=127.0.0.1
PUERTO_KERNEL=5010
```

[2] Un handshake es un intercambio de mensajes inicial que permite que los procesos se identifiquen y reconozcan, evitando también comunicaciones entre procesos que no deben hacerlo.

[3] Ver especificación del lenguaje AnSISOP en el Anexo II

Proceso Kernel

El proceso Kernel es el proceso principal del sistema. Recibirá los programas, almacenará los scripts AnSISOP en la Memoria y planificará su ejecución en los distintos procesos CPUs del sistema según alguno de los algoritmos definidos. Además, será el encargado de resolver las llamadas a sistema *(syscalls)* realizadas por los programas.

Arquitectura del Proceso Kernel

El proceso Kernel al ser iniciado se conectará con el proceso Memoria y el proceso FS, obtendrá el tamaño de página de la memoria y quedará a la espera de conexiones por parte de las CPUs y/o Consolas.

Al contar con al menos un Proceso CPU, comenzará a planificar los diversos programas en función del algoritmo de planificación.

- En cualquier momento, instancias de los Procesos CPU y Procesos Consola pueden ingresar o desconectarse del sistema, debiendo el Kernel responder de forma favorable e informar del cambio en la configuración del sistema.
- La falta de Procesos CPU en el sistema es posible. En ese caso, los programas quedarán a la espera de una CPU para ser planificados.
- Es posible que mediante la consola del Kernel se solicite la finalización de un programa. En ese caso, se deberán realizar las acciones solicitadas con el PCB correspondiente, cargando el código de error en el campo Exit Code, e informando al Proceso Consola correspondiente.

Creación del PCB

Al recibir la conexión de una nueva Consola, el Kernel intercambiará unos mensajes iniciales con la misma (handshake), para luego recibir la totalidad del código fuente del script que se deberá ejecutar.

El Kernel creará la estructura PCB con **al menos** los siguientes campos[4]:

- Un identificador único (PID), el cual deberá informar a la consola.
- Program Counter (PC)
- Referencia a la tabla de Archivos del Proceso
- Posición del Stack (SP)
- El Exit Code (EC) del proceso

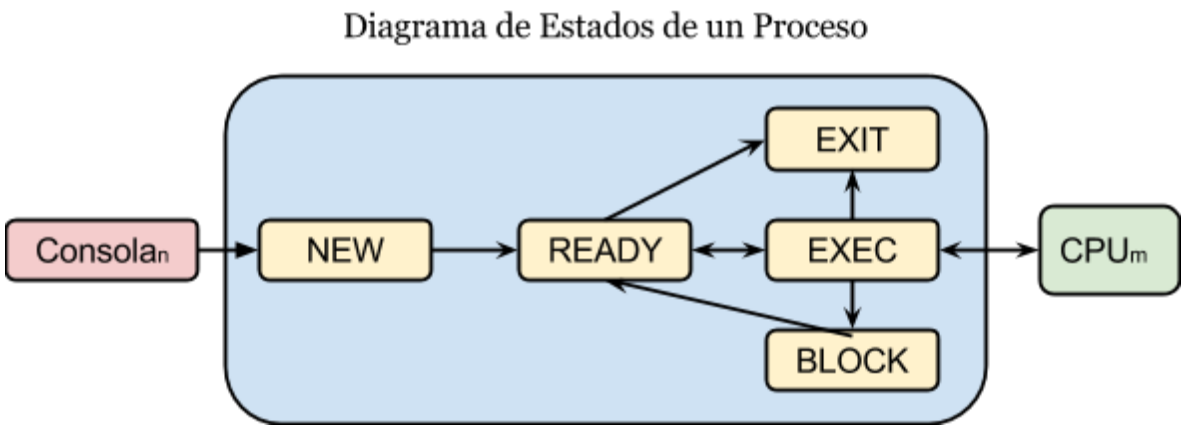
A partir de esta información, el Proceso Kernel deberá solicitarle al Proceso Memoria que le asigne lás páginas necesarias para almacenar el código del programa y el stack. Para simular el comportamiento de un sistema real, también le enviará el código completo del Programa.

Si no se pudiera obtener espacio suficiente para algunas de las estructuras necesarias del proceso, entonces se le rechazará el acceso al sistema, informándose oportunamente en el Proceso Consola correspondiente el motivo por el cual no pudo aceptarse el programa en el sistema.

Planificación de Procesos

El sistema planificará los procesos según el siguiente esquema:

[4] Para más información, referirse al Anexo I - Bloque de Control del Programa



Algunos de los aspectos principales del esquema a tener en cuenta son:

- Los procesos no se crearán en la memoria hasta que no pasen del estado NEW al estado READY. Esto quedará determinado por el grado de multiprogramación del sistema.
- Los PCB que lleguen a la cola de EXIT se quedarán en dicha cola para poder mantener una traza de ejecución del sistema.
- Es posible que en algún momento de la ejecución el sistema no cuente con CPUs conectadas, con lo cual no será posible tener procesos en el estado EXEC.
- Cualquier programa podrá ser finalizado desde la consola del Kernel

Algoritmos de Planificación

El sistema utilizará los algoritmos configurables FIFO y Round Robin. Para este último, debe ser configurable el valor del Quantum por archivo de configuración.

System Calls - Operaciones Privilegiadas

Existen operaciones en el sistema que no pueden ser resueltas únicamente por el proceso CPU, ya que estas requieren información privilegiada del sistema o acceso a elementos de la arquitectura sobre las cuales tan solo el Kernel tiene permisos. Dichas instrucciones son denominadas System Calls y deberán ser procesadas por el Kernel, a pedido expreso de una CPU.

Se registran dos módulos en el sistema que permiten operar System Calls: La Capa de Memoria, encargada de crear y reservar memoria para el sistema, y administrar las variables compartidas; y la Capa de FileSystem, encargada de proveer el acceso a archivos.

Capa de Memoria

En el código AnSISOP, los identificadores de las variables compartidas, para diferenciarlas de las variables normales, comenzarán con el caracter **signo de admiración (!)**, seguido del identificador alfanumérico, por ejemplo: !varGlobal o !precio. Las variables compartidas existentes en el sistema son definidas por archivo de configuración del Kernel y automáticamente inicializadas en cero.

Manejo de Semáforos

Los semáforos estarán definidos por archivo de configuración con un identificador alfanumérico y un valor inicial, por ejemplo: *SemSocket* o *SemDisco*. Se crearán al iniciar el proceso Kernel con los valores definidos en el archivo de config en SEM_INIT.

Reservar Memoria

Existe la posibilidad de que los procesos le soliciten al Kernel bloques de memoria dinámica. Para ello se creará, en caso que sea necesaria, páginas dedicadas exclusivamente a cumplir dicho propósito.

El Kernel será el encargado de administrar el ciclo de vida de estos bloques de memoria, denominados Heap, de forma tal que permita al proceso reservar y liberar bloques de forma aleatoria.

Gestión del Heap

Al atender la Syscall de reserva proveniente de un proceso, el Kernel deberá proveer un *espacio contiguo de memoria*, accesible para el proceso, sobre el cual el mismo pueda trabajar. Para ello, se reservará espacio en la Memoria a través de la creación de páginas dedicadas a almacenar memoria dinámica.

Con el fin de optimizar el espacio usado por los procesos dentro de las páginas, **se permitirá almacenar más de un bloque dentro de cada página.**

Esto significa que si trabajamos con páginas de 512 bytes, y un proceso hace una petición de 50 bytes, el Kernel creará una página en la memoria con el fin de almacenar dichos datos. Luego, si el proceso realiza una nueva petición, supongamos de 100 bytes, el Kernel utilizará la misma página ya creada para reservar dicho valor.

Para mantener la gestión de páginas utilizadas para Heap, el Kernel mantendrá una tabla, asociada al PCB que cuente con el PID del proceso, el número de página y el tamaño disponible dentro de la misma.

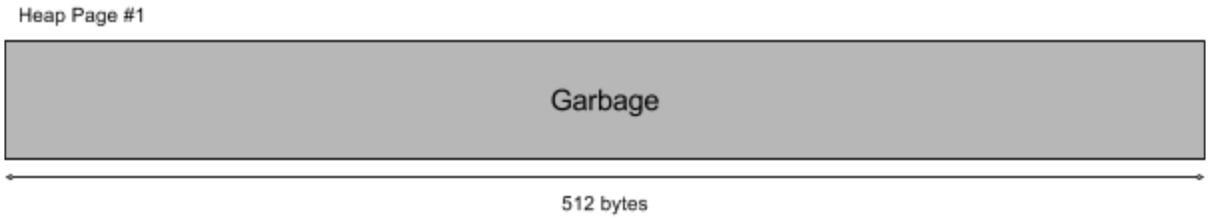
En el caso que el tamaño disponible en una página no sea suficiente para almacenar el valor requerido, se pedirá una nueva tabla a la Memoria.

Gestión de Página

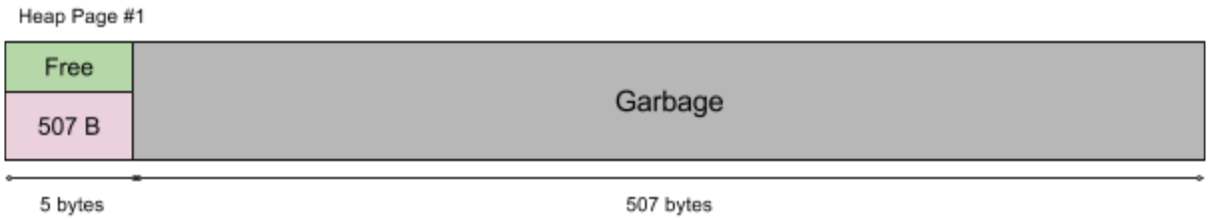
Cada página dedicada a la gestión de memoria dinámica podrá almacenar uno o más bloques de datos. Para poder determinar la consistencia de los bloques de datos almacenados en una página **se guardará metadata específica dentro de la misma,** la que nos permitirá determinar cuáles son las secciones de memoria sobre las cuales el proceso puede trabajar. Para ello, utilizaremos la siguiente estructura:

```
typedef struct HeapMetadata {

        uint32_t size,

        Bool isFree

}
```

Retomando el ejemplo anterior, supongamos que nuestro sistema cuenta con páginas de 512 bytes, y un proceso sin Heap previamente allocado reserva 50 bytes. Para atender dicho pedido, el Kernel deberá reservar una página, que en un principio cuenta tan solo con datos basura:

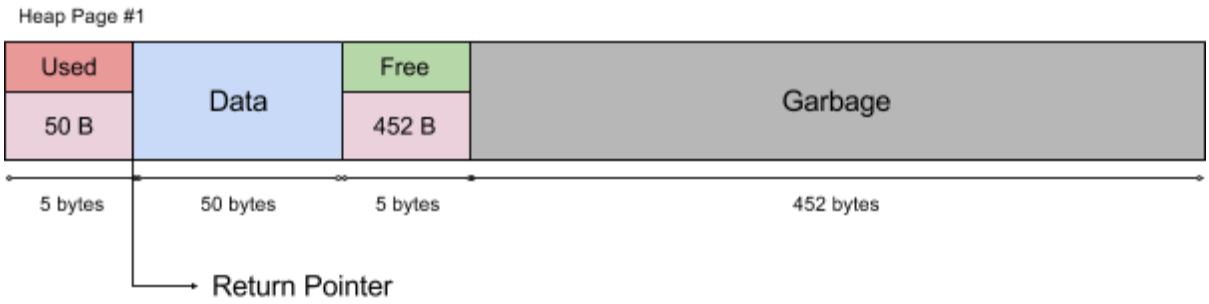


Luego, deberá registrar los bloques libres y ocupados, con su tamaño correspondiente. Al pedir una página, como todos los datos se encuentran disponibles para usar, se creará una estructura en la página que lo indique. Como dicha estructura ocupa 5 bytes (4 del tamaño del bloque y 1 que indica si está libre o no), tenemos 507 bytes libres para usar.



Se puede observar en la imagen que **la estructura indica cuál es el estado de los *size* bytes contiguos a la misma.** Por ende, siempre tiene que existir al menos una estructura de metadata en cada página de Heap, y siempre existirá al menos una que indique la cantidad de bytes libres (aunque estos sean 0).

Ante la reserva de 50 bytes de un proceso, el Kernel toma una página en donde hayan 55 bytes libres (50 para el bloque y 5 para la estructura), y **busca un lugar libre en la misma.** Si no existe alguna página con lugar suficiente, se pide una página nueva *(ver Fragmentación)*. El estado de la página anterior será:



Se observa en la imagen que **el valor a retornar por la syscall es la primer posición del bloque de datos reservado disponible.** Es importante notar que es responsabilidad del Kernel mantener la integridad de las estructuras, pero puede suceder que, ante una mala utilización del espacio de memoria, el proceso *"pise"* una estructura, guardando más datos de lo que reservó.

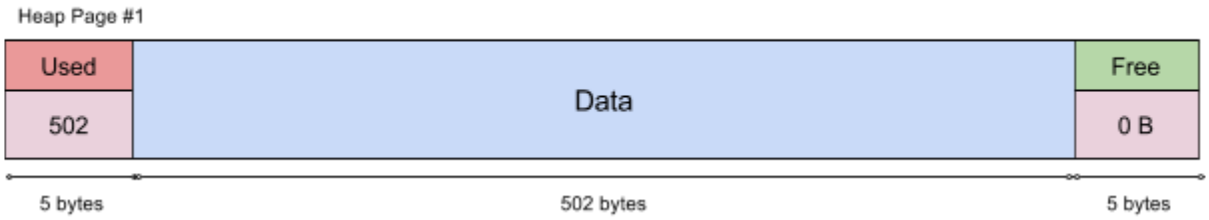
Tamaño máximo de Alloc

Un dato que se desprende de nuestro análisis recae en la *cantidad máxima de datos que podemos pedir.* Como no contamos con un esquema de segmentación, no podemos asegurar que las páginas que nos otorga la memoria se encuentren en posiciones contiguas. Por ende, tan solo podremos pedir memoria de un tamaño que se encuentre comprendido dentro de una página.

Como necesitamos, además de los datos, guardar la metadata de los bloques pedidos, **el tamaño máximo reservable por petición será de:**

Tmax = PAGE_SIZE - 10 Bytes

Quedando entonces, para nuestro ejemplo de página con 512 bytes, la siguiente estructura guardada en memoria:



En el caso que un proceso no pueda reservar memoria, deberá finalizar abruptamente.

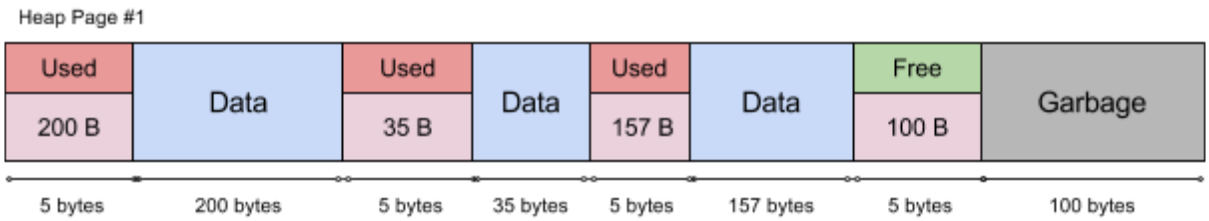
Liberación de Memoria

Un proceso puede solicitar liberar la memoria que tiene guardada en una posición de memoria. Ante ello, el Kernel debe buscar la estructura correspondiente al puntero indicado y marcarla como libre.

Si un proceso tiene una página con todos los bloques marcados como libres, esta debe ser liberada.

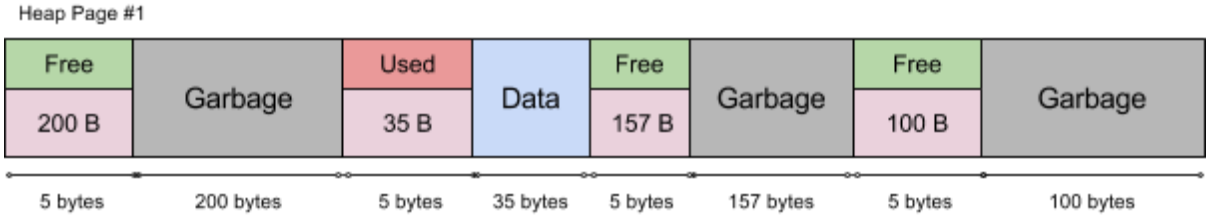
Fragmentación

Al reservar bloques en una página, un proceso genera estructuras intermedias que ocupan espacio dentro de la misma:



Supongamos que ahora el proceso necesita 250 bytes más, **el kernel deberá pedir una nueva página**, ya que no existen datos suficientes en la misma para almacenar dicho valor. Esta situación se denomina *Fragmentación Interna*, y no existe forma de solucionarlo bajo el modelo de paginación que utilizamos.

Ahora, supongamos que liberamos las posiciones 4 y 249 -*recuerden que las posiciones son en base 0*-, obtenemos el siguiente esquema:



Si el proceso ahora requiere un bloque de 250 bytes, **el Kernel tendrá que pedir una página nueva para guardarlo.** Esta situación se denomina *Fragmentación Externa* y **puede ser atacada en algunos casos.** Es responsabilidad del grupo realizar un análisis integral de cuáles son los casos salvables y proponer un algoritmo para resolverlo.

Memory Leaks

Al finalizar un proceso, el Kernel deberá informar si un proceso liberó todas las estructuras en las páginas de Heap (y todas las páginas), o no.

Consideraciones Generales

En caso de que algún programa desee realizar alguna operación incorrecta, como por ejemplo acceder a una variable compartida, realizar una operación de `wait` o `signal` a un semáforo inexistente, Intentar leer, escribir o cerrar un archivo que no fue previamente abierto, se considerará un error del programa y el mismo deberá finalizarse.

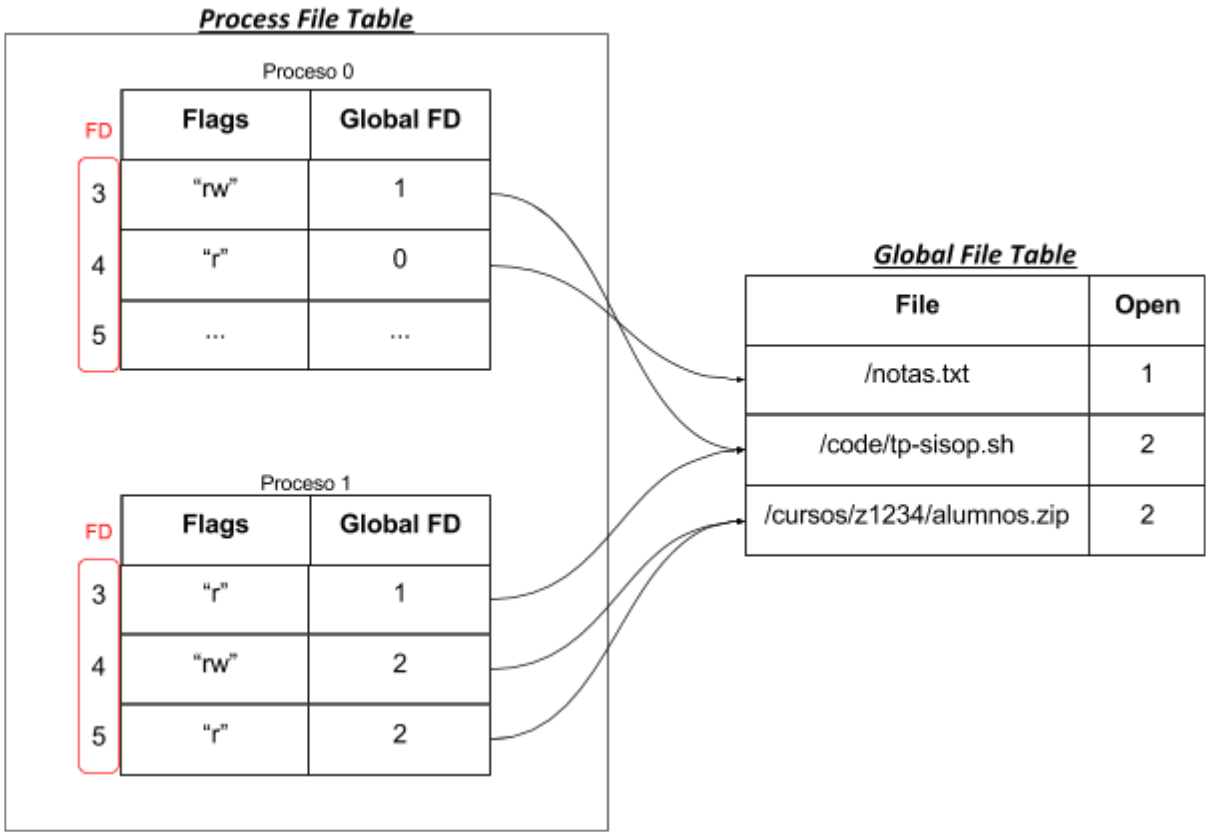
Capa de File System

En el código AnSISOP es posible que se requiera la intervención del Kernel para poder obtener los datos de un archivo almacenado en el FS. Para ello se definen 4 posibles instrucciones que operarán sobre File Descriptors.

File Descriptors

Un Descriptor de Archivo *(o File Descriptor, FD)* es un número entero positivo que identifica a un archivo o recurso abierto en el contexto de un proceso. El objetivo es poder generar una abstracción sobre un recurso que cumpla con la *interfaz de archivo,* y poder operar sobre el.

Para administrar los FD existentes en el sistema, el Kernel utilizará dos estructuras administrativas: Una **Tabla global de archivos,** que tendrá *una entrada para cada uno de los archivos abiertos en todo el sistema,* y una **Tabla de archivos para cada proceso,** que contendrá una entrada para cada uno de los archivos abiertos para ese proceso, referenciando una entrada a la tabla global. El índice de esta última tabla que se corresponda con el archivo abierto, **será el File Descriptor de ese archivo para ese proceso:**



Es interesante analizar dos particularidades que pueden observarse en la tabla ejemplo:

- Por convención, reservaremos los File Descriptors 0, 1 y 2. En los sistemas UNIX, dichos FD representan la entrada y salida estándar de un proceso, junto con los errores del mismo. En este Trabajo Práctico utilizaremos únicamente la salida estándar.
- Un mismo proceso puede abrir más de una vez a un mismo archivo, con diferentes o iguales permisos. Esta situación puede darse, por ejemplo, cuando varios Threads de un proceso se encuentran operando sobre el mismo archivo.
- La cantidad de procesos que se encuentran abriendo un mismo archivo será guardada en la Tabla Global de Archivos. Cuando un archivo no se encuentre abierto por ningún proceso (es decir, tenga el valor 0) en la tabla global, será eliminado de la misma.

Nota: Todos los archivos referenciados en este sistema se encontrarán en el FileSystem SADICA.

Un archivo **no puede ser borrado del FileSystem si existe su entrada en la Global File Table.** El mismo debe finalizar abruptamente si no puede realizar esta operación.

Impresión por Pantalla

El File Descriptor 1 será reservado para la Consola y no requiere ser listado en las tablas de archivos abiertos. Sin embargo, el grupo puede optar por modificar las tablas para incorporar las consolas correspondientes a cada programa si así lo desea.

Al escribir en el File Descriptor 1, se considerará que el programa quiere escribir por consola, y el Kernel será el encargado de detectar dicho comportamiento y enviar dicho texto a imprimir por pantalla.

Permisos de Archivos

El sistema permitirá la utilización de varias *flags o modos de acceso* al abrir un archivo. Cada flag representa un privilegio que el proceso le solicita al sistema operativo para operar sobre un archivo. A la hora de operar sobre un archivo, el Kernel deberá validar los privilegios del proceso en cuestión. Si un proceso no cuenta con los permisos necesarios, la ejecución del proceso terminará de forma irregular, se liberarán sus recursos y se fijará su Exit Code según corresponda.

Las flags serán provistas al abrir un archivo como un String, cada letra indicando una flag:

- Create (c): Crea el archivo al abrirlo si este no existe.
- Read (r): Permite al proceso leer los contenidos del archivo.
- Write (w): Permite al proceso escribir contenidos en el archivo.

Abrir Archivo

El kernel recibirá un path de un archivo y los permisos de apertura, generará un File Descriptor y se lo devolverá a la CPU para que esta continúe con la ejecución.

Leer

Para leer el contenido de un archivo el Kernel obtendrá el File Descriptor enviado por la CPU, traducirá dicho File Descriptor a un Path y realizará la petición al FS con los valores del Path, el offset y el tamaño de lo que desea leer para luego devolverlo a la CPU.

Escribir

Para leer el contenido de un archivo el Kernel obtendrá el File Descriptor enviado por la CPU, traducirá dicho File Descriptor a un Path y realizará la petición al FS con los valores del Path, el offset, el tamaño y el contenido a grabar en el archivo.

Una vez que reciba una respuesta del File System el Kernel enviará la confirmación a la CPU en caso de éxito o finalizará el programa en caso de Fallo.

Cerrar Archivo

Al cerrar un archivo el Kernel tendrá que destruir la relación entre File Descriptor y Path. Es importante que el sistema elimine de la tabla global de archivos aquellos que ya no se encuentran referenciados por ninguna tabla de procesos.

Terminación irregular de procesos

Existe la posibilidad de que un proceso finalice su ejecución de forma irregular. Para poder administrar la finalización de los procesos, el Kernel mantendrá en cada PCB un Exit Code que identificará el motivo por el cuál finalizó dicho proceso.

El Exit Code es un valor entero signado, que en caso de ser positivo, representará una finalización normal de un proceso. Por default, el valor de finalización de un programa que ejecutó todas sus instrucciones es 0.

Por otro lado, en caso de una finalización irregular, el Exit Code de un proceso será un entero negativo. El valor a guardar en el EC dependerá del motivo por el cual un programa finalizó inesperadamente:

Exit Code	Motivo
0	El programa finalizó correctamente.
-1	No se pudieron reservar recursos para ejecutar el programa.
-2	El programa intentó acceder a un archivo que no existe.
-3	El programa intentó leer un archivo sin permisos.
-4	El programa intentó escribir un archivo sin permisos.
-5	Excepción de memoria.
-6	Finalizado a través de desconexión de consola.
-7	Finalizado a través del comando Finalizar Programa de la consola.
-8	Se intentó reservar más memoria que el tamaño de una página
-9	No se pueden asignar más páginas al proceso
-20	Error sin definición

Existen errores no listados en esta tabla que pueden ocurrir dentro del sistema. Es menester la ampliación de la tabla con los errores que el grupo considere necesarios.

Cabe aclarar que *no es válido consultar por el Exit Code de un proceso antes de que el mismo finalice.*

Consola del Kernel

El Kernel contará con una consola que deberá permitir realizar las siguientes operaciones:

1. Obtener el listado de procesos del sistema, permitiendo mostrar todos o solo los que estén en algún estado (cola).
2. Obtener para un proceso dado:
    a. La cantidad de rafagas ejecutadas.
    b. La cantidad de operaciones privilegiadas que ejecutó.
    c. Obtener la tabla de archivos abiertos por el proceso.
    d. La cantidad de páginas de Heap utilizadas
        i. Cantidad de acciones alocar realizadas en cantidad de operaciones y en bytes
        ii. Cantidad de acciones liberar realizadas en cantidad de operaciones y en bytes
    e. Cantidad de syscalls ejecutadas
3. Obtener la tabla global de archivos.
4. Modificar el grado de multiprogramación del sistema.
5. Finalizar un proceso. En caso de que el proceso esté en ejecución, se deberá esperar a que la CPU devuelva el PCB actualizado para luego finalizarlo.
6. Detener la planificación a fin de que no se produzcan cambios de estado de los procesos.

Por la existencia de la consola, el Proceso Kernel no deberá mostrar mensajes de log por pantalla, solamente deberán ser guardados en su respectivo archivo de log.

Archivo de Configuración

Al iniciar, el Proceso Kernel deberá leer los siguientes parámetros de un archivo de configuración, cuya ruta se indicará como argumento de la línea de comandos del programa.

Parámetro	Tipo de dato	Descripción
PUERTO_PROG	[numérico]	Puerto TCP utilizado para recibir las conexiones de los Programas
PUERTO_CPU	[numérico]	Puerto TCP utilizado para recibir las conexiones de los CPUs
IP_MEMORIA	[ip]	Dirección IP del proceso Memoria
PUERTO_MEMORIA	[numérico]	Puerto TCP utilizado para conectarse al proceso memoria
IP_FS	[ip]	Dirección IP del proceso FS
PUERTO_FS	[numérico]	Puerto TCP utilizado para conectarse al proceso FS
QUANTUM	[numérico]	Valor del Quantum (en instrucciones a ejecutar) del algoritmo Round Robin.
QUANTUM_SLEEP	[numérico]	Valor de retardo en milisegundos que el CPU deberá esperar luego de ejecutar cada sentencia. Este valor puede modificarse en tiempo de ejecución[5]
ALGORITMO	[cadena]	El valor de este campo indicará el algoritmo de planificación del Kernel, los mismos pueden ser: ● FIFO ● RR
GRADO_MULTIPROG	[numérico]	Grado de Multiprogramación del sistema. Indica la cantidad máxima de procesos que pueden ser aceptados en el sistema simultáneamente al inicio de la ejecución. Este valor puede cambiar en tiempo de ejecución pero este cambio **no debe** reflejarse en el archivo de configuración.
SEM_IDS	[array: alfanumerico]	Identificador de cada semáforo del sistema. Cada posición del array representa un semáforo
SEM_INIT	[array: numérico]	Valor inicial de cada semáforo definido en SEM_IDS, según su posición
SHARED_VARS	[array: alfanumerico]	Identificador de cada variable compartida

[5] Para tener un ejemplo de cómo detectar cambios en los archivos, ver https://github.com/sisoputnfrba/so-inotify-example

STACK_SIZE	[numérico]	Tamaño en páginas del Stack

Ejemplo de Archivo de Configuración

```
PUERTO_PROG=5000
PUERTO_CPU=5001
IP_MEMORIA=127.0.0.1
PUERTO_MEMORIA=5002
IP_FS=127.0.0.1
PUERTO_FS=5003
QUANTUM=3
QUANTUM_SLEEP=500
ALGORITMO=RR
GRADO_MULTIPROG=1
SEM_IDS=[SEM1, SEM2, SEM3]
SEM_INIT=[0, 0, 5]
SHARED_VARS=[!Global, !UnaVar, !tiempo3]
STACK_SIZE=2
```

Las variables compartidas !**Global**, !**Unavar**, y !**tiempo3** se inicializan en cero.

Proceso File System

El proceso File System será el encargado de gestionar las peticiones que realicen los procesos AnSISOP sobre los archivos a través del Kernel. Para poder llevar adelante la gestión de las peticiones utilizará un sistema de archivos basado en SADICA.

Arquitectura

El proceso File System será un proceso de tipo servidor, es decir, que estará a la espera de conexiones de otros procesos, que deberá validar por medio de un Handshake que solamente esos procesos sean de tipo Kernel.

El proceso File System deberá atender las peticiones de manera secuencial, es decir, no podrá existir paralelismo a la hora de atender las peticiones del kernel.

Operaciones del File System

El Proceso File System, maneja una interfaz reducida, que no puede ser ampliada.

Validar Archivo

Parámetros: [Path]

Cuando el Proceso Kernel reciba la operación de abrir un archivo deberá validar que el archivo exista.

Crear Archivo

Parámetros: [Path]

Cuando el Proceso Kernel reciba la operación de abrir un archivo deberá, en caso que el archivo no exista y este sea abierto en modo de creación (“c”), llamar a esta operación que creará el archivo dentro del path solicitado. Por default todo archivo creado se le debe asignar al menos 1 bloque de datos.

Borrar

Parámetros: [Path]

Borrará el archivo en el path indicado, eliminando su archivo de metadata y marcando los bloques como libres dentro del bitmap.

Obtener Datos

Parámetros: [Path, Offset, Size]

Ante un pedido de datos el File System devolverá del path enviado por parámetro, la cantidad de bytes definidos por el Size a partir del offset solicitado. Requiere que el archivo se encuentre abierto en modo lectura (“r”).

Guardar Datos

Parámetros: [Path, Offset, Size, Buffer]

Ante un pedido de escritura el File System almacenará en el path enviado por parámetro, los bytes del buffer, definidos por el valor del Size y a partir del offset solicitado. Requiere que el archivo se encuentre abierto en modo escritura (“w”).

En caso de que se soliciten datos o se intenten guardar datos en un archivo inexistente el File System deberá retornar un error de Archivo no encontrado.

Sistema de Archivos De Índole Completamente Académica (SADICA)

El SADICA es un filesystem creado con propósitos académicos para que el alumno se interiorice y comprenda el funcionamiento básico de la gestión de archivos en un sistema operativo.

La estructura básica de SADICA se basa en el propio File System de linux, es decir, en una estructura de árbol de directorios para representar la información administrativa y los datos de los archivos. El árbol de directorios tomará su punto de partida del archivo de configuración.

Metadata

FileSystem

Este archivo tendrá la información correspondiente a la cantidad de bloques y al tamaño de los mismos dentro del File System.

Dentro de cada archivo se encontrarán los siguiente campos:

- Tamanio_Bloques: Indica el tamaño en bytes de cada bloque
- Cantidad_Bloques: Indica la cantidad de bloques del File System
- Magic_Number: Un String fijo con el valor "SADICA"

```
Ej:
      TAMANIO_BLOQUES=64
      CANTIDAD_BLOQUES=5192
      MAGIC_NUMBER=SADICA
```

Dicho archivo deberá encontrarse en la ruta [Punto_Montaje]/Metadata/Metadata.bin

Bitmap

Este será un archivo de tipo binario donde solamente existirá un bitmap[6], el cual representará el estado de los bloques dentro del FS, siendo un 1 que el bloque está ocupado y un 0 que el bloque está libre.

La ruta del archivo de bitmap es:

[Punto_Montaje]/Metadata/Bitmap.bin

File Metadata

Los archivos dentro del FS se encontrarán en un path compuesto de la siguiente manera:

[Punto_Montaje]/Archivos/[PathDelArchivo]

Donde el path del archivo incluye también el nombre y la extensión del archivo.

```
Ej: /mnt/FS_SADICA/Archivos/passwords/alumnosSIGA.bin
```

Dentro de cada archivo se encontrarán los siguiente campos:

- Tamaño: indica el tamaño real del archivo en bytes
- Bloques: es un array de números que contienen en orden los archivos

[6] Se recomienda investigar sobre el manejo de los bitarray de las commons library.

```
Ej:
      TAMANIO=250
      BLOQUES=[40,21,82,3]
```

Datos

Los datos estarán repartidos en archivos de texto nombrados con un número, el cual representará el número de bloque. (Por ej 1.bin, 2.bin, 3.bin),

Dichos archivos se encontraran dentro de la ruta:

[Punto_Montaje]/Bloques/[nroBloque].bin

```
Ej: /mnt/FS_SADICA/Bloques/1.bin
```

Archivo de Configuración

Al iniciar, el Proceso File System deberá leer los siguientes parámetros de un archivo de configuración, el cual podrá ser parametrizable como primer argumento del programa.

Parámetro	Tipo de dato	Descripción
PUERTO	[numérico]	Puerto TCP utilizado para recibir las conexiones del Kernel.
PUNTO_MONTAJE	[alfanumerico]	Valor del punto de montaje inicial del FS, a partir de este punto se creará el árbol para administrar el FS

Ejemplo de Archivo de Configuración

```
      PUERTO=5003
      PUNTO_MONTAJE=/mnt/SADICA_FS/
```

Proceso Memoria

El Proceso Memoria[7] es el responsable en el sistema de brindar a los Programas espacio en memoria para que estos realicen sus operaciones. Para ello simulará un mecanismo de tabla de páginas invertida y la utilización de una Memoria Caché. Es importante tener en cuenta que la memoria deberá tener una serie de estructuras administrativas las cuales deberán estar administradas dentro del mismo espacio de direcciones que los datos de los usuarios.

Arquitectura de la Memoria

Al iniciar, solicitará **un bloque de memoria contigua**[8] de tamaño configurable por archivo de configuración, para simular la memoria principal del sistema. Luego creará las estructuras administrativas necesarias para poder gestionar dicho espacio, permitiendo a cada Programa en funcionamiento utilizar un conjunto de páginas de tamaño fijo.

A la Memoria se conectarán, mediante sockets, el proceso Kernel y los diversos CPUs. Por cada conexión, la Memoria creará un hilo dedicado a atenderlo, que quedará a la espera de solicitudes de operaciones. La Memoria deberá validar cada pedido recibido, y responder en consecuencia.

- Ante cualquier solicitud de acceso a la memoria principal, el Proceso Memoria deberá esperar una cantidad de tiempo configurable (en milisegundos), simulando el tiempo de acceso a memoria. En caso de que la solicitud sea resuelta por la Memoria Caché, no se deberá esperar.
- Es importante destacar la naturaleza multi-hilo del proceso, por lo que será parte del desarrollo del trabajo atacar los problemas de concurrencia que surjan.
- A modo de simular el funcionamiento de un Sistema Operativo real, las estructuras administrativas de la Memoria deberán estar contenidas dentro del espacio del bloque de memoria contigua.
- En caso de que un proceso solicite memoria y no se le pueda asignar por falta de espacio, se deberá informar al proceso que lo solicite (CPU o Kernel) de que no hay más espacio disponible.

Diagrama de páginas de la memoria:

Las estructuras Administrativas deben guardarse en los primeros N bloques de la memoria, mientras que los bloques de los procesos se deberán ir asignando a medida que sean necesarios, pudiendo quedar de forma continua o no como se ve en el siguiente diagrama:

Adm	Adm	Adm	Adm	Adm	Adm	Adm	Adm	Adm	Adm	Adm	Adm
Adm	Adm	Adm	Adm	Adm	Adm	Adm	Adm	1	1	1	1
1	1	2	2	2	2	2	3	3	3	3	3
3	3	3	3	3	3	3	3	2	2	2	1

[7] *A pesar de que la implementación de la memoria virtual en un sistema real es una combinación entre el sistema operativo y el CPU (MMU), se implementa en este caso como un solo proceso aparte, para facilitar su comprensión e integración con el sistema.*

[8] Utilizando alguna función de la familia de `malloc`

1	1	5	5	5	5	5	5	5	5	5	3
4	4	4	4	4	3	3	1	1	2	2	2

Para hacer que las estructuras administrativas ocupen la menor cantidad posible de espacio en el sistema se utilizará una tabla de páginas invertida, y se recomienda la siguiente estructura de datos pudiendo alterarse la misma según las necesidades de cada grupo siempre buscando reducir el tamaño de la estructura a su mínimo tamaño:

#Frame	PID	#Pag
0	-1	0
1	-1	1
...	...	...
21	1	0
22	1	1
23	1	2
24	1	3

Frame Lookup

Potencialmente la tabla de páginas podría llegar a tener una cantidad muy alta de entradas, haciendo de la búsqueda secuencial un recurso muy lento. Para subsanar dicho inconveniente, se suele utilizar una *Función de Hashing* para encontrar rápidamente un subset de la tabla que contenga el valor a buscar.

Una función de hashing es una operación que convierte valores de entrada determinados en un valor único de salida. Dichas funciones tienen como propiedad ser **referencialmente transparentes,** es decir que *siempre que se llame a la función de hash con los mismos valores, se obtendrá el mismo resultado.*

Además, la función de hash **no es necesariamente inyectiva.** Esto significa que varios elementos del dominio pueden generar un mismo elemento del codominio. Dichos resultados se denominan colisiones, y se debe analizar qué sucederá con la búsqueda cuando estas ocurran.

¿Para qué nos sirve esto en el TP? Supongamos que obtenemos una función de hash que comprende la siguiente firma:

f(PID, N_PAGINA) = INDICE_EN_LA_TABLA

Podemos pensar en una función que genere los siguientes resultados:

f(0, 10) = 100

f(1, 15) = 150

f(0, 15) = 100

Por una cuestión de simplicidad, asociamos el número de frame con el índice en el que se encuentra la entrada en la página. En el caso de una función de hashing inyectiva, el valor que genera la misma es el índice en la tabla en el que se encuentra la página.

Pero, como vemos, nuestra función genera colisiones, ya que existe más de una combinación distinta de elementos del dominio que generan el valor 100. Por ende, no podemos determinar cuál es el frame exacto en el que se encuentra la página, pero una buena función de hashing **únicamente obtiene colisiones para frames contiguos.** Por ende, al obtener un valor podemos comenzar a evaluar secuencialmente los frames hasta llegar al valor que buscábamos.

Esto disminuye drásticamente el tiempo de búsqueda de las tablas.

Quedará a criterio del equipo **idear, validar con sus ayudantes e implementar una función de hashing que permita hacer búsquedas en nuestro modelo de paginación invertida.**

Memoria Caché

El proceso Memoria utilizará una estructura en que emulará una única caché del sistema, la cual permitirá tener un acceso más rápido a los datos de la memoria. A modo de simplificar el desarrollo de la Caché, la misma será una estructura de tamaño configurable por archivo de configuración, así como también la cantidad máxima de entradas que un proceso podrá tener en la misma (pudiendo no alcanzarse el máximo establecido por proceso, hasta incluso no tener ninguna entrada por encontrarse la caché totalmente llena). La caché deberá eliminarlas páginas cacheadas de un proceso cuando este sea finalizado.

La estructura de las entradas de la caché deberá ser la siguiente

- Identificador del Proceso
- Número de Página
- Contenido de la Página

Las entradas en la memoria caché se deberán reemplazar siguiendo el algoritmo LRU. El alcance del reemplazo es global, es decir, un proceso puede quitarle páginas a otro en caso de ser necesario.

Operaciones de la Memoria

El Proceso Memoria, simulando aspectos de un controlador de memoria real, maneja una interfaz reducida, que no puede ser ampliada.

Inicializar programa

Parámetros: [Identificador del Programa] [Páginas requeridas]

Cuando el Proceso Kernel comunique el inicio de un nuevo Programa, se crearán las estructuras necesarias para administrarlo correctamente. En una misma página no se podrán tener datos referentes a 2 segmentos diferentes (por ej. Código y Stack, o Stack y Heap).

Solicitar bytes de una página

Parámetros: [Identificador del Programa], [#página], [offset] y [tamaño]

Ante un pedido de lectura de página de alguno de los procesos CPU, se realizará la traducción a marco (frame) y se devolverá el contenido correspondiente consultando primeramente a la Memoria Caché. En caso de que esta no contenga la info, se informará un Cache Miss, se deberá cargar la página en Caché y se devolverá la información solicitada.

Almacenar bytes en una página

Parámetros: [Identificador del Programa], [#página], [offset], [tamaño] y [buffer]

Ante un pedido de escritura de página de alguno de los procesadores, se realizará la traducción a marco (frame), y se actualizará su contenido. En caso de que la página se encuentre en Memoria Caché, se deberá actualizar también el frame alojado en la misma.

Asignar Páginas a Proceso

Parámetros: [Identificador del Programa] [Páginas requeridas]

Ante un pedido de asignación de páginas por parte del kernel, el proceso memoria deberá asignarle tantas páginas como se soliciten al proceso ampliando así su tamaño. En caso de que no se pueda asignar más páginas se deberá informar de la imposibilidad de asignar por falta de espacio.

Finalizar programa

Parámetros: [Identificador del Programa]

Cuando el Proceso Kernel informe el fin de un Programa, se deberán eliminar las entradas en estructuras usadas para administrar la memoria.

Otras operaciones

Para facilitar el desarrollo del trabajo práctico se definen también una serie de funciones que exceden a la administración de memoria.

Handshake

[Tipo: Kernel/CPU]

Consola de la Memoria

El Proceso Memoria, al iniciar, quedará a la espera de comandos enviados por teclado permitiendo al menos las siguientes funcionalidades. El diseño de la misma y la sintaxis de los comandos queda a criterio del equipo.

- ***retardo***: Este comando permitirá modificar la cantidad de milisegundos que debe esperar el Proceso Memoria antes de responder una solicitud. Este parámetro será de ayuda para evaluar el funcionamiento del sistema.
- ***dump***: Este comando generará un reporte en pantalla y en un archivo en disco del estado actual de:
  - **Caché:** Este comando hará un dump completo de la memoria Caché.
  - **Estructuras de memoria**: Tabla de Páginas y Listado de procesos Activos
  - **Contenido de memoria**: Datos almacenados en la memoria de todos los procesos o de un proceso en particular.
- **flush**
  - **Caché**: Este comando deberá limpiar completamente el contenido de la Caché.
- **size**
  - **Memory**: Indicará el tamaño de la memoria en cantidad de frames, la cantidad de frames ocupados y la cantidad de frames libres
  - **PID**: indicará el tamaño total de un proceso

Archivo de Configuración

Al iniciar, el Proceso Memoria deberá leer los siguientes parámetros de un archivo de configuración, el cual podrá ser parametrizable como primer argumento del programa.

Parámetro	Tipo de dato	Descripción

PUERTO	[numérico]	Puerto TCP utilizado para recibir las conexiones del Kernel y los CPUs
MARCOS	[numérico]	Cantidad de marcos disponibles en el sistema.
MARCO_SIZE	[numérico]	Valor en bytes del tamaño de marco del sistema.
ENTRADAS_CACHE	[numérico]	Cantidad disponible de entradas en la cache. 0=Deshabilitada
CACHE_X_PROC	[numérico]	Cantidad máxima de entradas de la cache asignables a cada Programa
RETARDO_MEMORIA	[numérico]	Cantidad de milisegundos que el sistema debe esperar antes de responder una solicitud que aceda a la memoria principal.

Ejemplo de Archivo de Configuración

```
PUERTO=5003
MARCOS=500
MARCO_SIZE=256
ENTRADAS_CACHE=15
CACHE_X_PROC=3
REEMPLAZO_CACHE=LRU
RETARDO_MEMORIA=100
```

Proceso CPU

Este proceso es el encargado de interpretar y ejecutar las operaciones escritas en código AnSISOP de un Programa.

Estará en permanente contacto con el Proceso Memoria, tanto para obtener información del Programa en ejecución, como para actualizar las estructuras requeridas luego de ejecutar una operación.

Al iniciar, se conectará al Proceso Kernel y quedará a la espera de que este le envíe el PCB de un Programa para ejecutarlo.

Incrementará el valor del registro Program Counter del PCB y utilizará el índice de código para solicitar a la Memoria la próxima sentencia a ejecutar. Al recibirla, la parseará, ejecutará las operaciones requeridas, actualizará los valores del Programa en la Memoria, actualizará el Program Counter en el PCB y notificará al Kernel que concluyó su ejecución.[9]

Ejemplo:

> Luego de recibir de la Memoria la instrucción "a = b + 3", el parser la interpretará y ejecutará las siguientes **Primitivas**:
>
> 1) `obtener_direccion('b')`: utilizando el índice del stack, devolverá la posición de la variable b. Por ejemplo: *Pag: 5, Offset: 8, Size: 4.*
> 2) `obtener_valor(Pag: 5, Offset: 8, Size: 4)`: pedirá a la Memoria el valor de la **variable b**, en este caso los 4 bytes a partir del offset 8 de la página 5. Supongamos b = 9
> 3) `obtener_direccion('a')`: como 1), pero para saber dónde guardar el resultado. Por ejemplo, Pag: 4, Offset: 0, Size: 4.
> 4) `almacenar(Pag: 4, Offset: 0, Size: 4, 11)`: almacenará en la Memoria el resultado de la suma (9 + 3 = 11).

El ingreso a funciones o procedimientos requiere que se asienten datos como el punto de retorno o las variables locales en el índice de ***Stack*** - Ver detalle en el Anexo I: PCB - Índice del Stack

Hot plug

En cualquier momento de la ejecución del sistema pueden conectarse nuevas instancias del proceso CPU. Será responsabilidad del Kernel aceptar esas nuevas conexiones, y tener en cuenta a las nuevas CPUs a la hora de planificar.

Mediante la señal SIGUSR1, se le podrá notificar a un CPU que deberá desconectarse una vez concluida la ejecución del Programa actual, dejando de dar servicio al sistema.

Fin de la ejecución

Al ejecutar la última sentencia, el CPU deberá notificar al Kernel que el proceso finalizó para que este se ocupe de solicitar la eliminación de las estructuras utilizadas por el sistema.

[9] En este punto el alumno ya puede notar que si el proceso fuera desalojado y su PCB enviado a otro CPU, este tendría ahí y en la Memoria toda la información necesaria para continuar su normal ejecución

Excepciones

Existe la posibilidad que el Proceso CPU reciba un mensaje de excepción como resultado de una solicitud a la Memoria o al Kernel. En este caso la CPU deberá mostrar por pantalla un mensaje aclarando cuál fue el inconveniente que sucedió y como consecuencia del error se deberá finalizar el programa AnSISOP que se encuentre en ejecución.

Anexo I - Bloque de Control del Programa (PCB)

Al igual que en un sistema operativo convencional, todo Programa estará identificado por una estructura denominada PCB (Process Control Block - Bloque de Control del Programa) la cual contendrá el identificador único del proceso y el ***program counter (PC)*** del programa. Con esta información, cada proceso CPU puede retomar la ejecución desde el punto que se encontraba la última vez que el proceso fue expropiado por el Kernel.

Junto con los campos comunes del PCB, se almacenarán algunas estructuras auxiliares para el proceso (como los tres índices) con el fin de simplificar la comprensión del trabajo práctico y su posterior desarrollo:

Estructura	Información
Identificador Único	Número identificador del proceso único en el sistema.
Program Counter	Número de la próxima instrucción del Programa que se debe ejecutar.
Páginas de código	Cantidad de páginas utilizadas por el código del Programa AnSISOP, empezando por la página cero.
Índice de código	Estructura auxiliar que contiene el offset del inicio y del fin de cada sentencia del Programa.
Índice de etiquetas	Estructura auxiliar utilizada para conocer las líneas de código correspondientes al inicio de los procedimientos y a las etiquetas.
Índice del Stack	Estructura auxiliar encargada de ordenar los valores almacenados en el Stack.
Exit Code	Número entero que identifica el motivo de finalización del proceso.

Índice de Código

AnSISOP es un lenguaje interpretado, por lo que las líneas de código pueden tener distintas longitudes, además de poder existir líneas en blanco o comentarios.

Dado que la Memoria atiende únicamente solicitudes relacionadas con posiciones de memoria, el índice de código es una estructura que almacena el desplazamiento respecto al inicio del código del programa y la longitud de cada línea ejecutable.

De esta manera es posible saber la ubicación de cada línea *útil* en el código para poder solicitarla a la Memoria.

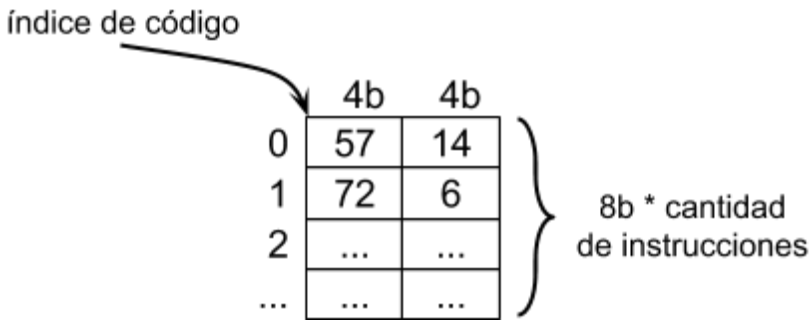
Es vital recordar que el índice **no es de líneas sino de instrucciones**: el código para generar este índice recorrerá el código de un programa ignorando las primeras líneas vacías o con comentarios hasta encontrar la primer instrucción válida, registrará su inicio y su longitud y, luego, buscará la siguiente ignorando saltos y comentarios. Repetirá el procedimiento hasta el fin del archivo.

Por ejemplo, para un script como el siguiente:

```
#!/usr/bin/ansisop
begin
```

```
# primero declaro las variables
variables a, b
a = 20

print a

end
```

La instrucción 0 (`variables a, b`) comienza en el byte 57, con una longitud de 14 bytes, mientras que la instrucción 1 (`a = 20`) comienza en el byte 72 y mide 6 bytes. Se recomienda utilizar un array de dos valores enteros como el descrito a continuación:



Al momento de ejecutar una instrucción, la CPU solicitará a la Memoria la entrada del Índice correspondiente a la instrucción a ejecutar, determinada por el Program Counter. Teniendo el byte de inicio y longitud de la instrucción, realizará la petición a la Memoria en el correspondiente valor de `Pagina, Offset y Longitud`.

Índice de Etiquetas

El Índice de Etiquetas consta de la serialización de un diccionario que asocia el identificador de cada función y etiqueta del programa con la primer instrucción ejecutable de la misma (es decir, el valor que el Program Counter deberá tomar cuando el programa deba pasar a ejecutar esa función o saltar a esa etiqueta).

Índice del Stack

Esta estructura, también parte del PCB, es una manera auxiliar de simplificar la operatoria del segmento de Stack de un proceso.

Cada vez que un Programa AnSISOP ingrese a una función, se deberá agregar un elemento a este índice con los siguientes atributos:

Nombre	Tipo	Descripción
Argumentos (args)	Lista de Argumento	Posiciones de memoria donde se almacenan las copias de los argumentos de la función.
Variables (vars)	Lista de Variable	Identificadores y posiciones de memoria donde se

		almacenan las variables locales de la función
Dirección de Retorno (retPos)	Numérico	Posición del índice de código donde se debe retornar al finalizar la ejecución de la función.
Posición de la variable de retorno (retVar)	Posición de memoria (Pag#, Offset, Size)	Posición de memoria donde se debe almacenar el resultado de la función provisto por la sentencia RETURN

Ejemplo de Índice de Stack

Para ejemplificar el Índice del Stack, utilizaremos el siguiente código AnSISOP:

```
#!/usr/bin/ansisop
begin
variables a,g
    a = 1
    g <- doble a
    prints g
end

function doble
variables f
    f = $0 + $0
    return f
end
```

Como se puede observar en el código, existen dos funciones: la principal y la función doble.
Al iniciar el programa, se crea la el siguiente registro en la estructura de stack, dado que la función principal no tiene argumentos, ni debe retornar por lo que las columnas retPos y retVar están vacías.

Pos	args	vars	retPos	retVar
0				

Al momento de ejecutar la sentencia variables a, g se agregan en la columna vars las entradas correspondientes a cada variable:

Pos	args	vars	retPos	retVar
0		**ID** | **Pag** | **Off** | **Size** **a** | 0 | 0 | 4		

		g | 0 | 4 | 4		

El programa ejecuta y, en un punto, llega a la línea de código útil número 3: `g <- doble a`
En este punto generamos una nueva entrada en nuestro Stack donde:

- En la columna retPos ponemos el valor 3 ya que es el punto de nuestro índice de código donde deberemos retornar una vez que finalice nuestra función doble.
- En la columna args pondremos la posición de memoria de la copia de la variable a que es el argumento que recibe esta función
- En la columna retVar pondremos la posición de memoria donde se guardará el retorno de nuestra función, en este caso es la dirección de la variable g

Pos	**args**	**vars**	**retPos**	**retVar**
0		**ID** | **Pag** | **Off** | **Size** **a** | 0 | 0 | 4 **g** | 0 | 4 | 4		
1	**ID** | **Pág** | **Off** | **Size** a | 0 | 8 | 4		3	**Pág** | **Off** | **Size** 0 | 4 | 4

Al avanzar en la ejecución de nuestro programa encontraremos la sentencia `variables f`, en este punto deberemos agregar a nuestra columna vars una entrada para nuestra variable:

Pos	**args**	**vars**	**retPos**	**retVar**
0		**ID** | **Pag** | **Off** | **Size** **a** | 0 | 0 | 4 **g** | 0 | 4 | 4		
1			3	

	Pág | **Off** | **Size** 0 | 8 | 4	**ID** | **Pág** | **Off** | **Size** **f** | 0 | 12 | 4		**Pág** | **Off** | **Size** 0 | 4 | 4

Luego de llegar al final de la función `doble`, marcada por la instrucción `end`, deberemos eliminar la entrada del stack quedando el mismo de la siguiente forma:

Pos	**args**	**vars**	**retPos**	**retVar**
0		**ID** | **Pag** | **Off** | **Size** **a** | 0 | 0 | 4 **g** | 0 | 4 | 4		

Es de suma importancia tener en cuenta que la cantidad de entradas del índice de stack puede variar muchas veces durante la ejecución de un proceso, aca se utilizó un ejemplo fácil para mostrar cómo funciona.

Otro aspecto no menor a tener en cuenta es que el índice de args no se referencia por el nombre de la variable, sino que como se verá en el Anexo II – Especificación del Lenguaje se referencian mediante un valor numérico comprendido entre **[0-9]** y precedido por el signo **$**

Anexo II – Especificación del Lenguaje

AnSISOP es un lenguaje de programación interpretado de propósito general y bastante bajo nivel. Su sintaxis es simple y, en general, no es muy recomendable para utilizar en tareas productivas. En cambio, su objetivo es principalmente académico: ayudar a entender los conceptos y mecanismos que un sistema operativo debe tener en cuenta para manejar la ejecución de los programas.

El lenguaje se divide en dos capas: la sintaxis de alto nivel, utilizada para escribir los scripts, y las operaciones Primitivas que la CPU deberá ejecutar para llevar a cabo las instrucciones de los primeros. El alumno deberá desarrollar el intérprete del programa y las diversas primitivas que serán invocadas.

Para la evaluación del trabajo práctico no se proveerán programas con errores de sintaxis ni de semántica.

Sintaxis

- El lenguaje es case-sensitive; es decir, `hola` y `Hola` son diferentes
- El código principal del programa estará comprendido entre las palabras reservadas *begin* y *end*. *begin* solo indica por donde comenzará a ejecutar el programa.
- Las sentencias finalizan con un salto de línea. Los saltos adicionales son ignorados.
- Toda línea comenzada por un caracter numeral (#) es un comentario y debe ser ignorado.
- Todo programa deberá terminar con una línea en blanco.
- Un ":" seguido de una palabra es una etiqueta que será utilizada para permitir saltos dentro del código con *jump, jz* y *jnz*, explicados más adelante.
- Todas las variables dentro de una función son locales.
- Una función puede llamar a otra función.

Variables

Las variables locales se declaran luego de la sentencia *variables*. Son solamente de tipo entero con signo y su identificador es un caracter alfabético [a-zA-Z]. Su valor no debe ser inicializado. Se las indica en el código solo con su nombre.

Las variables dadas como parámetros de funciones se nombran con un único dígito [0-9] y se accederá a ellas en el código anteponiendo el signo **$** (`$0` refiere al primer parámetro, y así).

Las variables compartidas[10] se declaran e inicializan en la configuración del Kernel, se nombran como cadenas sin restricción de nombre, y se las indica en el código como **!identificador**.

Asignación

Con el nombre de la variable a la izquierda de un signo igual, se le podrá asignar a una variable como valor:

- Un número entero u otra variable (local, parámetro o compartida; no semaforos)
- El resultado de una operación aritmética la cual podrá ser suma o resta

[10] Llamamos "*variables compartidas*" a aquellas manejadas por el Kernel, de las que todos los CPUs tiene acceso. No son "*variables globales*" ya que "*global*" refiere al scope/contexto de cada programa. No existen en las variables globales.

Salto condicional

Las instrucciones de salto condicional *saltar-si-no-es-cero* (jnz - jump on not-zero) y *saltar-si-es-cero* (jz - jump on zero) recibirán como parámetro una variable que evaluarán y una etiqueta a la que deberán saltar en caso de que se cumpla la condición.

Estas instrucciones, por definición del enunciado, tendrán como origen y destino el procedimiento actual. En otras palabras, el código de una función o procedimiento no podrá saltar dentro del código de otro.

Este código de ejemplo incrementa la variable i de uno a diez e imprime dichos valores en pantalla.

```
begin
variables     i,b
      i = 1
      :inicio_for
      i = i + 1
      prints i
      b = i - 10
      jnz b inicio_for
      #fuera del for
end
```

Observe que si la variable *i* no es igual a 10 entonces b = i - 10 no es cero, entonces la instrucción de salto condicional iría a la etiqueta *inicio_for* hasta llegar a la 10ma iteración, donde no saltará más.

Impresión en pantalla

Para poder imprimir por pantalla dentro de un programa AnSISOP se deberá llamar a la función prints, la cual recibirá una variable como parámetro, la cual se mostrará por pantalla según corresponda:

La información deberá ser mostrada en la terminal del programa y registrada en el log del sistema.

Ejemplo:

```
a = 45
prints "El valor de A es:"
prints a
```

Resultado:

El valor de A es:
45

Funciones

La definición de las funciones estará dada por la palabra reservada *function* seguida del nombre de la misma. Todas las funciones retornan un valor y no existe en el lenguaje el concepto de procedimiento.

Código de ejemplo

Este código de ejemplo imprime variables.

```
function prueba
      variables a,b
```

```
      a = 2
      b = 16
      print b
      print a
      a = a + b
      return a
end

begin
variables a, b
      a = 20
      print a
      b <- prueba
      print b
      print a
end
```

Lo que se ve por pantalla sería (Nótese la localidad/scope de las variables):

20
16
2
18
20

Anexo III - Primitivas de AnSISOP

Para evitar la complejidad que presenta realizar un analizador de sintaxis y dado que estas tareas no son inherentes al contenido de la materia, se le facilita al alumno un Parser que se encargará de interpretar cada línea de código y ejecutar las Primitivas correspondientes, cuyo código será desarrollado por el alumno. Tanto su implementación directa o parcial es aconsejable.

Tanto el código del Parser como la documentación de sus primitivas podrá obtenerse desde: https://github.com/sisoputnfrba/ansisop-parser

Nota: Al momento de la publicación de esta versión, la API de las primitivas se encuentra en estado de desarrollo, y podrá encontrarse en la branch *NewAPI* del Parser. Al finalizar su desarrollo, se actualizará la branch Master. Por el momento, pueden encontrar la API tentativa en:

https://github.com/sisoputnfrba/ansisop-parser/blob/NewAPI/parser/parser/parser.h

Descripción de las entregas

Para permitir una mejor distribución de las tareas y orientar al alumno en el proceso de desarrollo de su trabajo, se definieron una serie de puntos de control y fechas que el alumno podrá utilizar para comparar su grado de avance respecto del esperado.

Se propone un esquema de trabajo iterativo, en donde el equipo deberá mostrar un sistema funcionando para cada checkpoint. Ante la posibilidad de que no se llegue a cubrir todos los elementos del checkpoint, se recomienda a los equipos que se enfoquen en tener funcionalidades activas y testeables. El poder ir depurando errores en cada checkpoint resulta de gran ayuda para llegar a la entrega final con un producto pulido.

El enfoque indicado en los siguientes checkpoints se basa en la interconexión temprana de todos los módulos. **Recomendamos fuertemente empezar a probar las funcionalidades con los procesos conectados entre sí** *(incluso lo proponemos desde el primer checkpoint)*. Creemos que de esta forma, surgen rápidamente los problemas de interfaces, y esto se puede ver reflejado rápidamente en un sistema que funcione.

Los checkpoints catalogados como *"Obligatorios"* requieren que un ayudante evalúe explícitamente el progreso del equipo. En cambio, los checkpoints *"no obligatorios"* son a modo de guía, y podrán ser evaluados por un ayudante en caso que el grupo lo desee.

Checkpoint 1 - Obligatorio

Fecha: 15/04/2017

Objetivos:

Crear los proyectos para los diversos procesos del trabajo práctico y sincronizarlos al repositorio de Git.

Desarrollar un servidor multicliente en el Kernel y clientes simples en el resto de los procesos. Cada proceso deberá leer su configuración inicial desde su archivo de configuración, en la que contará con los parámetros de conexión.

Al iniciar, cada proceso informará por pantalla su configuración, hará un Handshake con los procesos con quienes deba conectarse y luego quedarán a la espera de recibir datos.

Desde la consola podrá enviarse un mensaje con tamaño máximo fijo al Kernel, el cual deberá imprimir por pantalla el texto ingresado en la misma, y replicarlo a la Memoria, CPU y FileSystem. Estos últimos también deberán mostrar el contenido del mensaje por pantalla.

Distribución recomendada:

- ★ Todos los integrantes trabajando equitativamente.

Lectura recomendada:

- ★ http://faqoperativos.com.ar/arrancar
- ★ Beej Guide to Network Programming - link
- ★ Linux POSIX Threads - link
- ★ SisopUTNFRBA Commons Libraries - link

Checkpoint 2

Fecha: 06/05/2017

Objetivos:

El sistema anterior ahora permite enviar archivos a través de un comando enviado por el proceso Consola, de tal forma que sean almacenados en el proceso Memoria y sean accesibles desde el Kernel y las CPU. Dichos archivos serán de tamaño variable.

El kernel a este checkpoint tan solo permitirá la ejecución de un proceso muy simple que se creará en el momento de enviar el código desde la consola. Se creará un pequeño PCB con un ID de proceso y un contador de páginas para dicho PCB. Al enviar los datos a la Memoria, y obtener el OK de la misma, se aumentará el contador de páginas.

Si bien el Kernel debe permitir la conexión de múltiples consolas, debe validar que no exista más de una consola enviando datos en simultáneo. Para ello, se fijará un nivel de multiprogramación en 1 en el archivo de configuración.

Al inicializar el proceso Memoria a este checkpoint, reservará un gran bloque de datos de tamaño configurable que será visto como un solo Frame, obteniendo tan solo una gran página. Esto le permitirá al Kernel leer y escribir bytes en dicho frame, utilizando la API definida por el enunciado. Este proceso deberá permitir, a través de su consola, hacer un dump del único frame utilizando el comando *dump*.

Por otro lado, deberá poder conectarse un proceso CPU a través de un handshake al Kernel, y se deberá incorporar al mismo la biblioteca del parser. Se visualiza ya la implementación de algunas instrucciones básicas de ANSISOP.

Distribución recomendada:

- ★ Consola: 1 Integrantes.
- ★ Kernel: 1,5 Integrantes.
- ★ Memoria: 1,25 Integrante.
- ★ FileSystem: 0 Integrantes
- ★ CPU: 1,25 Integrante

Lectura recomendada:

Sistemas Operativos, Silberschatz, Galvin - Capítulo 4: Hilos
Sistemas Operativos, Silberschatz, Galvin - Capítulo 5: Planificación del Procesador

Checkpoint 3

Fecha: 20/05/2017

Objetivos:

El proceso Consola ahora reconoce todos los comandos propuestos en el enunciado. El objetivo de este checkpoint es tener una aplicación que ejecute programas simples.

A la hora de recibir un nuevo proceso, el Kernel almacenará las páginas de dicho proceso en la Memoria y guardará los valores de las mismas en el PCB, inicializando el Program Counter y cargando los datos del los índices provistos por el parser. Sumado a esto, el PCB deberá ahora contener datos del Exit Code del programa, que podrá ser 0 o un valor negativo dependiendo si la consola finalizó su conexión antes de tiempo. Además, el Kernel permitirá obtener el listado de procesos del sistema, finalizar un proceso, consultar su estado y detener la planificación.

Además, existirá una regla fija ante la syscall write que permite enviar a imprimir texto a la consola siempre y cuando el FD sea 1. Caso contrario, el sistema deberá fallar.

La Memoria ya se encuentra dividida en Frames de tamaño configurable. Se creará la estructura de paginación inversa, que asociará un Frame, una Página y un PID. Por el momento no se requiere que la estructura de páginas se encuentre en memoria principal. Deberá atender, además, las peticiones de memoria de nuestra CPU, por lo que ya se visualiza

una arquitectura multihilo. Para evitar problemas de sincronización en una etapa temprana, es válido atender los pedidos de forma secuencial.

El proceso CPU ya puede conectarse con la Memoria y realizar pedidos de lectoescritura. Además, el CPU permite ahora ejecutar una ráfaga de ciclos de instrucción al recibir un PCB del Kernel. Para ello, ahora puede manejar la estructura del Stack. Si bien no se requiere que se implementen todas las instrucciones, es menester la implementación de la mayoría de las operaciones no privilegiadas.

Distribución recomendada:

- ★ Memoria: 2 integrantes
- ★ Kernel: 2 integrantes
- ★ CPU: 1 integrantes

Lectura recomendada:

Sistemas Operativos, Silberschatz, Galvin - Capítulo 4: Hilos
Sistemas Operativos, Silberschatz, Galvin - Capítulo 5: Planificación del Procesador

Checkpoint 4 - Obligatorio (En laboratorio)

Fecha: 03/06/2017

Objetivos:

El objetivo de este checkpoint es soportar la utilización de varias operaciones privilegiadas sobre memoria.

La consola comienza a llevar la información estadística requerida en el enunciado.

El Kernel ya puede trabajar con múltiples programas, permitiendo indicar su nivel de multiprogramación por archivo de configuración. Debido a esto, comienzan a vislumbrarse los estados de ejecución de un programa. Se requiere, cuánto mínimo, una cola de New que mantenga el grado de multiprogramación, una cola de Ready, una cola de Exec que mande a ejecutar un programa a una única CPU, y una cola de Exit que permite mantener la información de finalización de un programa. El sistema de planificación será FIFO.

Se crea en el Kernel la Capa de Memoria, inicializando las variables compartidas y soportando las operaciones con semáforos de ANSISOP, junto con las estructuras administrativas necesarias para que puedan utilizarse. Se sincroniza correctamente el acceso a las mismas.

Además, se soportan ahora las syscall sobre las operaciones de sincronización de ANSISOP y sus variables compartidas.

La memoria ya contiene un hilo por conexión y permite acceso en simultáneo a las estructuras del sistema.

El proceso Filesystem deberá poder reconocer estructuras de árboles de directorios y archivos. Podrá responder una solicitud simple del Kernel como la existencia de un archivo o su tamaño.

Distribución recomendada:

- ★ Kernel y consola: 3 integrantes
- ★ Memoria: 1 integrante
- ★ CPU: 1 integrante

Checkpoint 5

Fecha: 10/06/2017

Objetivos:

El objetivo de este checkpoint es tener un programa que permite ejecutar varios programas en forma paralela, incorporando las estructuras del FileSystem y una caché que deberá aumentar ampliamente su performance.

El Kernel ya permite ejecutar varios programas en paralelo. Se debe prestar especial atención a la hora de sincronizar la aplicación para soportar las transiciones de múltiples programas.

Se amplía la capa de Memoria del Kernel, encargada de administrar las syscalls que requieren acceso a la creación de memoria dinámica del programa (malloc).

Se crea la capa de FileSystem en el Kernel, encargada de manejar la tabla de archivos por proceso y tabla de archivos globales. Se implementan las syscalls de la CPU que permiten abrir y cerrar archivos, junto con los comandos de la consola del kernel que permiten manejar dichas estructuras.

Se crea la estructura de caché para la memoria.

Las CPUs ahora corren en forma paralela. Además, se implementan las operaciones que requieren acceso al heap de los programas.

Distribución recomendada:

- ★ Kernel: 2 integrantes
- ★ CPU: 2 integrantes
- ★ Memoria: 0.5 integrantes
- ★ Filesystem: 0.5 integrantes

Entrega Final

Fecha: 08/07/2017

Objetivos:

Finalización del trabajo práctico, se itera por última vez para obtener una versión de TP con todos los módulos funcionando.

Se incorporan al Kernel los algoritmos de planificación a la hora de elegir qué procesos ejecutar. Además, se implementan permisos sobre los archivos que se encuentran abiertos, validando todos los casos de errores posibles. El Kernel ya mantiene información estadística de los programas a la hora de ejecutarlos.

La CPU ya puede ejecutar instrucciones que requieren acceso a archivos, completando todas las instrucciones del sistema. Se permite la desconexión de los procesos CPU en tiempo de ejecución, y se maneja de forma acorde el error.

Se implementan las operaciones que el enunciado propone desde la consola de la Memoria para manejar la caché.

El FileSystem ya permite obtener los datos de un archivo y escribir sobre el mismo a través de un mensaje del Kernel. Esto permite al Kernel y la CPU implementar las Syscalls que leen y escriben sobre archivos. Además, se permite borrar archivos desde la syscall de ANSISOP.

Distribución recomendada:

- ★ Kernel: 2 integrantes
- ★ CPU: 2.5 integrantes
- ★ Memoria y Filesystem: 0.5 integrantes

Lectura recomendada:

- ★ Test del trabajo práctico provistos por la cátedra.