Sistemas Operativos I – Trabajo Final – UNQ – diciembre de 2009

por Nahuel Garbezza

Emulación de funciones de un Sistema Operativo y de un Sistema de Archivos - "LittleOS"

La implementación del Sistema Operativo en Python incluye:

- Carga de programas desde archivos de texto plano (que más adelante se detalla su sintaxis).
- Gestión y planificación de procesos.
- Algoritmo de evitación de deadlocks.
- Manejo básico de Entrada/Salida (Pantalla, Entrada de datos, Disco).
- Implementación de un sistema de archivos basado en i-nodos.
- Consola para manejo del SO y también del sistema de archivos por separado.

Para poder iniciar las consolas del SO y el sistema de archivos, se lo puede hacer de dos maneras:

- Por medio de Eclipse IDE: abrir el archivo "mainOS.py" (o "mainFS.py" en caso del sistema de archivos) y ejecutarlo directamente.
- Por intérprete de comandos (sin entorno de programación): hay que modificar el encabezado de los archivos mencionados anteriormente de la siguiente forma:

```
import sys
sys.path.append("path completo hasta la carpeta src")
```

Con esto estaremos diciendo a Python la ruta para poder buscar los módulos y paquetes. Luego se ejecuta con : **`python mainOS.py`** / **`python mainFS.py`**

Nota: los archivos se encuentran en la carpeta "src/opsys"

Ejecución de programas

Los programas que el sistema operativo pueda ejecutar deberán estar almacenados en el sistema de archivos emulado. De todos modos, en cualquiera de las consolas (más adelante se explica su funcionamiento) se pueden cargar archivos externos (y guardarlos en nuestro sistema de archivos). En resumen, todos los programas estarán en el sistema de archivos, pero también está la posibilidad de agregar programas externamente.

La consola del Sistema Operativo

Aquí se detallan los comandos posibles que la consola puede entender, de todas formas con el comando "**`help`**" aparecerá en la consola una breve descripción de los comandos:

- **`start`** : Inicia todos los servicios, como los de planificación, la CPU, el sistema de archivos, los dispositivos de E/S, etc. En este punto el sistema está listo para ejecutar programas.
- **`addprog <nombreArchivo>;<nombre>`** : Este comando sirve para ingresar al

sistema de archivos un programa a partir de un archivo en el disco real. En el campo <nombreArchivo> debe estar la ruta completa incluido el archivo deseado, de otro modo el comando fallará. En el campo <nombre> se debe indicar qué nombre tendrá el archivo en nuestro sistema de archivos. Luego de su carga, el archivo se puede mover / eliminar (ya quedó en el sistema de archivos simulado). Este comando sólo se puede ejecutar si el sistema está detenido.

- **load <nombreArchivo>** : Comando para comenzar la ejecución de un programa. El archivo que reciba tiene que estar ya en el sistema de archivos, en la carpeta "//programs". No hay que especificar ninguna ruta, solo el nombre del archivo.
- **shutdown** : Comando que cierra y finaliza todos los servicios iniciados con **start**. Tener en cuenta que esto también finalizará todos los procesos que se estén ejecutando en ese momento.

La consola del sistema de archivos

Esta consola se utiliza para agilizar las operaciones sobre los archivos, o sea sin tener que correr el sistema operativo. Es requisito para la consola que el sistema operativo no esté corriendo, pues el disco es un recurso, y para que un recurso no genere conflictos en el sistema operativo corriendo, se debe hacer una solicitud. Pero para que haya una solicitud, tiene que haber un proceso, y así... En resumen, no se debería estar accediendo al "disco" por dos lados (procesos, con solicitudes; y por consola, directamente) esto generaría conflictos.

Al igual que la consola del SO, ésta también acepta el comando **help** para la sintaxis de los comandos:

- **mkdir <nombre>**: Crea un directorio nuevo en el directorio actual.
- **rmdir <nombre>**: Borra un directorio que debe estar vacío.
- **cd <directorio>**: Ingresa al directorio especificado, como valores especiales existen "." (retorna el mismo directorio), ".." (va hasta el directorio padre), y "/" (que desde cualquier punto va hasta el directorio raíz).
- **rendir <directorio>;<nuevonombre>**: cambia de nombre un directorio.
- **newfile <nombre>**: crea un nuevo archivo en el directorio actual.
- **open <nombre>**: carga un archivo en memoria.
- **close <nombre>**: elimina un archivo de memoria. Descarta cambios.
- **save <nombre>**: guarda datos de un archivo abierto.
- **write <nombre>;<datos>**: escribe datos a un archivo abierto.
- **renfile <archivo>:<nuevonombre>**: cambia de nombre un archivo.
- **del <archivo>**: borra un archivo del sistema de archivos. No tiene que estar abierto.
- **ls**: muestra en un listado todas los directorios y archivos del directorio actual.
- **showdisk**: comando especial que imprime en pantalla todos los bloques del disco, para ver cómo están organizados.
- **addprog <ruta completa al archivo>;<nombre dentro del sist>**: carga un archivo real dentro del sistema de archivos, en la carpeta "programs".
- **exit**: cierra el sistema de archivos y sale de la consola.

En la consola, los espacios serán ignorados, por una cuestión de simplicidad, excepto para la operación **write** (así se pueden ingresar espacios en archivos).

Ambas consolas se diseñaron utilizando la librería *cmd,* que viene inlcuida con Python.

Acerca de los programas y sus sintaxis

Para que un programa pueda ser ejecutado por el SO tiene que cumplir con las siguientes condiciones:

- Ser de texto plano
- Estar en el sistema de archivos simulado del SO (esto se puede hacer con la instrucción **addprog**, descripta anteriormente)
- Tener estrictamente la siguiente sintaxis:

  Name : <nombrePrograma>
  Priority : <prioridad>
  <inst 1>
  <inst 2>
  ...
  <inst n>
  donde:
  <nombrePrograma> sólo debe contener letras (mayusculas o minúsculas), sin números ni espacios
  <prioridad> es un numero de 1 a 10
  <inst> es una instrucción, con sintaxis particular, que iremos analizando de a una.
  Se pueden colocar comentarios, con la secuencia "**--** ", y valen sólo para la linea donde esta ubicada.

Sintaxis de las instrucciones

Una instrucción se compone de un nombre, que la identifica y cero o más parámetros. Ejemplo:

add #12 6

donde la primera parte es el nombre de la función, y las otras dos son parámetros, que pueden ser:

- **Registros :** se denotan con "**Rn**" donde **n** es un número de 0 a 7.
- **Celdas de memoria:** son relativas al proceso (esto evita que dos procesos compartan memoria, lo cual resulta riesgoso), se denotan con el carácter **#** seguido del número de dirección. El número obviamente no puede ser cualquiera, los procesos tienen un espacio direccionable (que por defecto es 20 celdas)
- **Números:** solo se escribe su valor
- **Strings:** se escriben entre comillas dobles y no pueden contener caracteres especiales, pero sí espacios, puntos y la barra inclinada (/).

Los parámetros fueron modelados cada uno como una clase distinta, pues de esta forma se pueden tener instrucciones más dinámicas (todas las combinaciones de parámetros para la instrucción **mov**), e incluso polimórficas (por ejemplo el **add** para números es la suma y para los strings es la concatenación).

Semántica de las instrucciones

Instrucciones propias de CPU : Son aquellas que representan cómputo real que sólo puede ser ejecutado por la CPU. Las instrucciones que entiende el CPU son:

- **mov <destino> <origen>** : instrucción de copia de datos. El campo <destino> puede ser un registro o una dirección de memoria. <origen> puede ser cualquier parámetro.

- **add <1sumandoYdestino> <2sumando>**: suma para números, concatenación para Strings. El resultado se guarda en el primer parámetro, que debe ser un registro o celda de memoria. Obviamente los dos operandos tienen que ser del mismo tipo, sino falla.
- **sub <minuendoYdestino> <sustraendo>**: igual que **add**, solo que hace la resta para números, y no tiene implementación para Strings .
- **skip**: instrucción que no hace nada.

Instrucciones de control: Son aquellas que pasan por CPU pero que sirven para manejar dispositivos de E/S. Estas instrucciones son:

- **request <cant> <recurso> ...** : sirve para solicitar recursos que se van a utilizar luego. Para utilizar un recurso debe solicitarse primero. Ejemplo: si queremos pedir la pantalla y el disco, escribimos: **request 1 "disk" 1 "display"**
  El orden puede ser cualquiera y los nombres de los dispositivos se ponen como Strings.
- **free** : instrucción que libera todos los recursos solicitados anteriormente. No recibe parámetros.

Aclaración: se pueden realizar tantos request-free como se desee.

Instrucciones propias de E/S: Comandos que sólo pueden ejecutar ciertos dispositivos. Los dispositivos que este SO va a manejar son:

- Pantalla ("display")
- Teclado ("keyboard")
- Disco ("disk") - En realidad, acceso al sistema de archivos.

Las instrucciones para cada dispositivo son:

Pantalla :

- **show <parámetros>** : Imprime en pantalla el/los parámetros. Los parámetros pueden ser cualquier cosa menos registros.

Teclado:

- **input <mensaje> <celdaMemoria>** : Lee strings por medio de una ventana. Dicha información la guarda en memoria, en la dirección indicada en el parámetro. También se debe indicar el mensaje que aparecerá en la ventana, indicado como string.
- **intinput <mensaje> <celdaMemoria>** : Igual que input, pero convierte la entrada a entero.

Disco:

Las mismas operaciones que en la consola, con el agregado de:

- **readall <archivo> <celdaMemoria>**: Lee todo el contenido de un archivo y lo coloca en la celda de memoria especificada.
- **writeline <archivo> <contenidos>** : genera una linea en un archivo y copia el contenido pasado como parámetro. <contenidos> puede ser string, numero o celda de memoria.
- **Pwd:** imprime en pantalla el directorio actual.

Todas las instrucciones tienen un tiempo asignado (en segundos) que representa su ejecución y depende de su nombre. Esto es realizado para poder observar mejor la simulación. El tiempo de cada instrucción es configurable (más adelante se explica la configuración).

Un ejemplo de programa puede ser el siguiente:

```
Name : prueba1
Priority: 3
-- este es un comentario y se ignora completamente
mov R0 "Hola "
mov #4 " como andas..."
request 1 "keyboard" 1 "display"
input "Ingrese su nombre..." #7
add R0 #7
mov #3 R0
add #3 #4
show #3
free
request 1 "disk"
cd "ejemplos"
newfile "prueba"
write "prueba" #3
save "prueba"
close "prueba"
free
```

El ejemplo anterior muestra algunas de las posibilidades que se pueden lograr con las instrucciones, y con los recursos.

Cualquier fallo en alguna instrucción significará la finalización de ese proceso, sin afectar en absoluto a los otros procesos.

Asignación / liberación de los recursos a los procesos

Es sabido que en sistemas con multiprogramación la asignación de recursos puede generar problemas, los deadlocks. Este sistema implementa una versión del Algoritmo del Banquero para evitación de deadlock. Es decir, para cada pedido verifica con un algoritmo (el de seguridad) si el mismo se puede satisfacer y además satisfacer futuros pedidos de los otros procesos. Si el sistema queda en un estado inseguro, ese pedido no puede satisfacerse y por ende seguirá intentando.

Acerca de los procesos

Un proceso no es más que un programa en ejecución. El sistema necesita representarlo de alguna forma, y la estructura más importante el el PCB, que en este caso está compuesto de:

- Contador de programa
- Identificación del proceso
- Información de directorios y archivos abiertos
- Información de la planificación
- Prioridad
- Estado / Registros
- Parámetros de memoria (registro base, límite)

Acerca de las planificaciones

El sistema posee dos tipos de planificaciones: la de largo plazo y la de corto plazo.

La planificación de largo plazo se encarga de mantener un equilibrio entre los usos de cada proceso (procesos con mucho uso de CPU, contra procesos con mucho uso de E/S). Toma procesos recién creados y los coloca en la cola de listos, donde trabaja el planificador a corto plazo. Éste decide que proceso obtiene la CPU, basándose en diferentes políticas. Las políticas implementadas en este sistema son:

- FCFS
- SJF (versión expropiativa)
- SJF (versión no expropiativa)
- Prioridad (versión expropiativa)
- Prioridad (versión no expropiativa)
- Round Robin

Acerca de los dispositivos de E/S

Cada tipo de dispositivo tiene un manejador, y las instancias que le han sido asignadas. La CPU se comunica con los manejadores y les envía procesos para ejecutarse. Las instancias son hilos de ejecución que revisan cada cierto tiempo el estado del manejador. Si existe un proceso, se asigna una instancia de recurso y ésta se encarga de correr las instrucciones del proceso. Pero la implementación de las instrucciones está en un set de instrucciones (propio de cada tipo de recurso).

Concurrencia entre los componentes del sistema operativo

Cuando se ejecuta **`start`** en el sistema operativo, lo que ocurre es que se lanzan en paralelo varios hilos de ejecución (thread) para cada componente. Los dispositivos que corren en threads aparte son:

- El sistema operativo (la clase OS)
- La CPU
- El planificador a corto plazo
- El planificador a largo plazo
- Las instancias de recursos (si tenemos 2 pantallas y un disco, tenemos 3 threads)

Los tiempos de espera entre cada loop de cada componente son configurables. Los threads que se utilizan son los de la librería *threading*.

Acerca del sistema de archivos y su implementación

El manejo de archivos por parte del sistema operativo se realiza sobre el sistema de archivos, una estructura de organización de los datos utilizando objetos persistentes (framework *pickle*). No obstante se deja una entrada para archivos reales, sino sería imposible construir un “disco virtual” que es el que este sistema de archivos gestiona. Los datos en el “disco” están organizados en i-nodos, y las operaciones que se proveen son:

- Creación de archivos y directorios
- Eliminación de archivos y directorios
- Navegación por los diferentes directorios
- Abrir / Cerrar archivos (Cargar en memoria / Sacar de memoria)
- Lectura / Escritura de archivos
- Renombre de archivos y directorios

Cómo está representada la información (bajo nivel)

La manera en que los datos se guardan en el “disco” es en forma de bloques, los hay de 4 tipos:

- Bloque de directorio
- Bloque de archivo
- Bloque de índices
- Bloque de datos

Los bloques de directorio y de archivo contienen en su interior referencias a bloques de índices, representados en dos niveles de indirección. En el primero, existe un bloque de índices cuyos índices apuntan a directorios o archivos (en el caso de directorios) o a bloques de datos (en el caso de archivos). En el segundo nivel, el bloque principal apunta a otros bloques de índices, obteniendo así un aumento de capacidad exponencial. No obstante, habrá un límite de capacidad, determinado por la máxima cantidad de índices que pueda haber. Ejemplo:

Total de entradas en un directorio (2 niveles de indirección):

total = tamaño de bloque + tamaño de bloque ^ 2

Con un tamaño de bloque = 10 tenemos 110 entradas posibles para un directorio, descontando los casos especiales (“.” y “..”) nos quedan 108.

Tamaño máximo de un archivo (2 niveles de indirección):
Como cada índice apunta a un bloque de datos, el tamaño también va a estar determinado por el tamaño de dicho bloque. Entonces:

total = (tamaño de bloque + tamaño de bloque ^ 2) * tamaño de bloque de datos

Con bloque de índices = 10 y bloque de datos = 25 caracteres, tenemos 110 bloques posibles, multiplicado por 25 = 2750 caracteres

El disco está implementado con un diccionario, para permitir asignación dinámica, o sea sólo está representado lo que se necesita, sin espacios vacíos (importante ventaja sobre un array o matriz o cualquier estructura de tamaño fijo). El segundo bloque de indirección tanto para archivos como para directorios sólo se usa si el primero está completo, por lo que se gana en eficiencia en este punto.

Además, se mantiene una lista de tamaño configurable que contiene índices a bloques disponibles, lo que hace más eficientes las operaciones que impliquen crear nuevos bloques (creación de archivos/directorios, guardar archivos). La lista cuando agota sus bloques disponibles, recorre el disco y genera nuevos bloques disponibles. Cuando se eliminan datos, los bloques eliminados pueden ser reasignados (esto puede verse con el comando **`showdisk`**).

La memoria

Está implementada al igual que el disco como un diccionario, con las ventajas mencionadas anteriormente. Permite el direccionamiento relativo, para ello debe recibir el PCB para calcular la dirección real como la suma entre la dirección lógica y el registro base del proceso. Si el resultado es mayor que el registro límite, se genera un error.

Se implementó usando el patrón Singleton, por el hecho de que debía ser accedida desde muchos lugares, y sin ser singleton, hubiese generado mucho acoplamiento entre los componentes, y complejidad en los constructores.

Finalización de procesos

Una condición de error en un proceso (mal la semántica de una instrucción, como sumar un número con un string, o borrar un archivo que no existe) no debería significar la caída del sistema. Es por ello que lo que se intentó realizar es manejo de excepciones, y en caso de ocurrir algún error relacionado con un proceso, “matarlo” (eliminarlo del sistema sin afectar a los otros procesos).

La responsabilidad de “matar” un proceso fue delegada al singleton denominado ProcessKiller, que es singleton porque existen muchas condiciones de error por varios lugares, y el único momento en el que actúa es en caso de un error.

El proceso "loader"

El sistema operativo sólo corre programas almacenados en su sistema de archivos emulado. Ahora bien, cómo acceder a él mientras otros procesos corren, y también lo utilizan? Pues no se puede. Mientras el sistema operativo esté corriendo, sólo los procesos pueden acceder y hacer uso del sistema de archivos. Entonces, necesitamos alguien que cargue los programas. El responsable es un proceso especial (llamado "loader") que está definido dentro del sistema operativo, y contiene las primitivas para cargar programas. Su funcionamiento es el siguiente: Comienza a correr, accede al disco buscando el archivo que contiene el programa deseado, carga en memoria su contenido (no crea un archivo), y finaliza su actividad. El sistema operativo verifica que esté en la memoria los contenidos volcados por el loader, y entonces ya tiene lo necesario para lanzar el "verdadero" proceso. El "loader" tiene las siguientes características: es el más prioritario (prioridad 1), su PID es siempre 1, ingresa directamente a la "cola de listos", y sus instrucciones son de tiempo 0, para garantizar su rapidez.

Acerca del parseo

Al principio se dijo que el sistema corría programas partiendo desde archivos de texto plano. También se habló de las instrucciones y de los procesos. Ahora bien, cómo, cuándo, dónde se realiza este pasaje de texto a objetos? El encargado de realizar esto es el parser. Para definir una gramática que se ajuste a las necesidades del proyecto, se recurrió al framework *pyparsing* (http://pyparsing.wikispaces.com/) de Python, que simplifica la tarea. El framework está incluido en el proyecto, por lo que no es necesario ninguna instalación externa.

La configuración

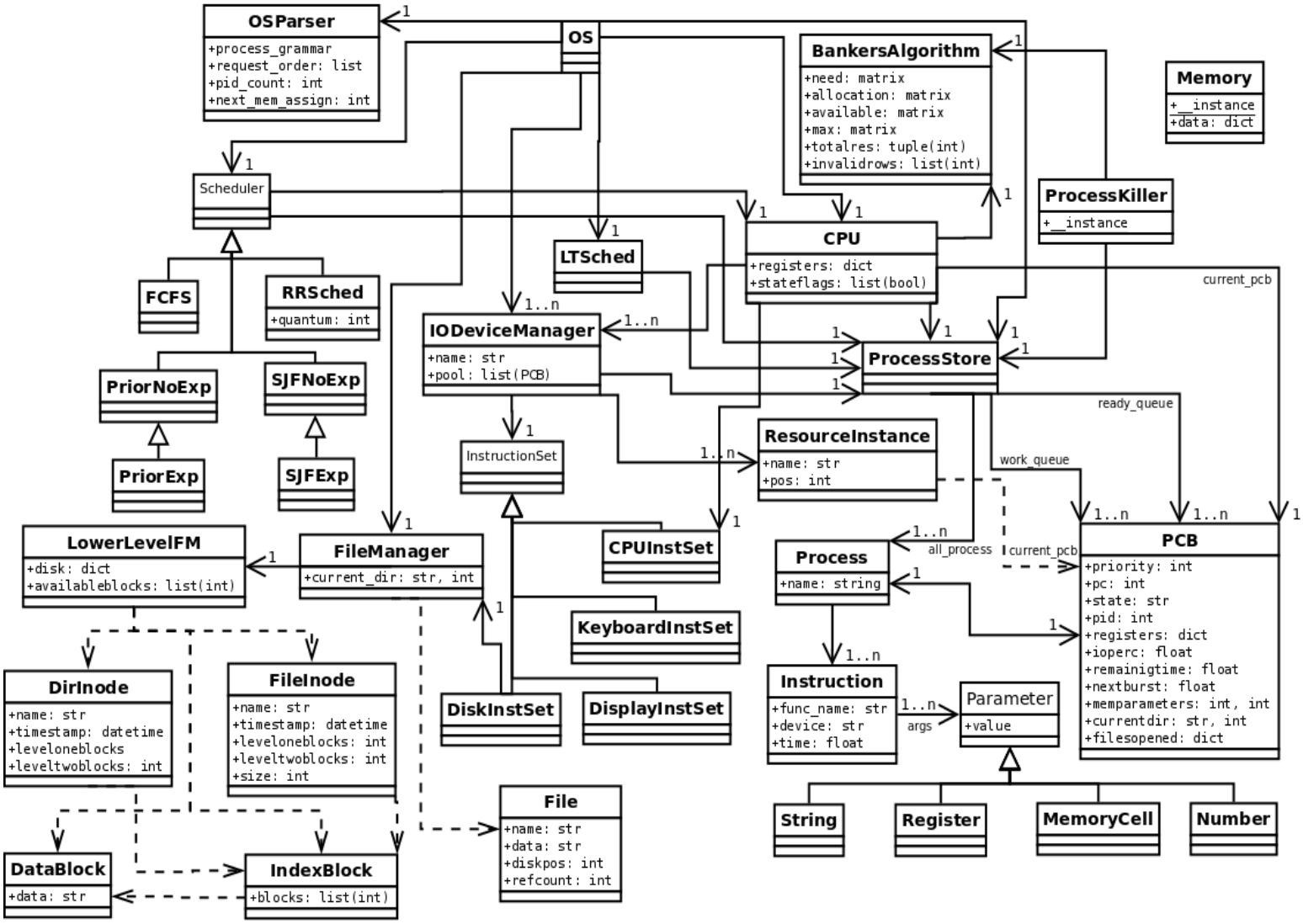
Inicialmente se pensó en tener un archivo de configuración, en el cual se pudiera colocar por ejemplo los planificadores, los recursos, etc. Ello implicaba tener que parsear, definiendo una gramática, y lo que es peor de todo, agregarle responsabilidades al parser que no debería tener (hasta incluso, definir una clase Config que sea singleton y que contenga la configuración). Pero luego, por una cuestión de simplicidad y como una opción para solucionar estos inconvenientes, se incluyeron los parámetros de configuración en un archivo Python, en forma de constantes. Dicho archivo se llama “osconfig.py” y está en la carpeta principal del proyecto. De esta forma, también es más sencillo agregar nuevos parámetros de configuración.

Logging

Los eventos más importantes que se generan en el sistema son guardados a un archivo .log. El responsable de hacer esto es el Logger (otro Singleton). Lo más interesante del logger es que utiliza locks (librería *multiprocessing*) para asegurar de que la escritura al archivo sea realizada sólo por un proceso a la vez. La ruta al archivo log es configurable. Si no se finaliza el sistema operativo con **`shutdown`**, no se guardará nada de lo que se registró.

Diagrama de clases

El diagrama está simplificado por cuestiones de claridad, sólo se incluyeron las variables de instancia más relevantes. La línea punteada indica una relación que no tiene que ver con conocimiento directo, por ejemplo: el FileManager trabaja con Files pero estos no forman parte de su estructura interna, o un DirectoryInode tiene un número que referencia a un IndexBlock, es decir, existe una relación.



Trabajos futuros

- Hacer un set de instrucciones más completo para el CPU
- Incorporar más dispositivos de E/S
- Modelar usuarios / permisos para el sistema de archivos
- Incorporar administración y algoritmos de memoria
- Interfaz gráfica

Apéndice / Extras

Patrones de diseño utilizados: No fueron muchos, template method en la clase Scheduler, singleton en ProcessKiller, Memory y Logger, y una especie de strategy de IODeviceManager con InstructionSet. También se usó double-dispatch para evaluar los parámetros de las instrucciones.

Lista de archivos ya cargados en el sistema de archivos y listos para ejecutar (que también están en la carpeta "examples"):

ej1.pr
super.pr
family.pr
readyourself.pr

Existe un backup del disco (archivo serializado) en la carpeta "diskbackup".