SISTEMAS OPERATIVOS
Informe

Rizzi, Román - Correa, David

Introducción

Durante lo largo de la cursada, estuvimos implementando una demostración de un Sistema Operativo. En las primeras clases, el modelo que teníamos era bastante simple, pero con el correr de las clases, se le fueron sumando complejidades, que terminaron haciendo de nuestro simple modelo, una representación casi real de lo que es un Sistema Operativo hoy en día y como opera con los elementos de la Computadora. Varias cosas de los sistemas que se utilizan hoy por hoy, fueron simplificadas dado a la complejidad que presentan sus implementaciones.

Tecnologías usadas

Como lenguaje *default* de la materia, utilizamos Python (2.7), y para poder lograr cosas específicas del modelo, utilizamos librerías externas, tales como *JSonPickle* y *cmd*. Para no tener que estar trabajando siempre en la misma computadora, decidimos versionar nuestro código; si bien había varias alternativas posibles de hacer esto, decidimos utilizar *GitHub*. Ninguno de los dos tenía mucha experiencia en como usarlo y poder aprovechar sus ventajas, pero quisimos utilizarlo de igual forma, para poder ir aprendiendo. En un principio, solo utilizamos un solo Branch, pero a medida que la complejidad del modelo fue incrementando, tuvimos que empezar a trabajar en un nuevo Branch en donde podamos ir poniendo las cosas que íbamos haciendo, incluso no estando terminadas. Como IDE para programar, utilizamos *PyCharm*, que nos facilitó mucho la conexión con el repositorio, dado al complemento que tiene para repositorios GitHub.

Decisiones de Diseño

- **Drive Allocation**

Este paquete contiene diversas clases y tiene como objetivo principal, persistir la información de manera no volátil. Es más interesante y se le podrá sacar más jugo si adentramos en ellas una a una.

- HDD: Es el encargado de almacenar toda la información persistida, esta compuesto de un *DriveSaver* que se encarga de grabar sobre el, sectores de disco (representados mediante un Dictionary) donde se almacenan los *Disk Blocks* (Almacenan las instrucciones del programa), posee una representación de la estructura del *File System* serializada a *Json* (En base a esto puede serializar y deserializar la representación actual bajo demanda). Por último, posee un área dedicada a acciones de *Swap* utilizadas por la política de manejo de memoria llamada Paginación.

- File System: Representación lógica de la información en disco, provee una capa de abstracción sobre este. Se buscó implementar de forma tal que estuviera totalmente desacoplado del disco en sí (Si vemos el codigo, en ningun momento se hace mención al disco). Posee una estructura de árbol, teniendo archivos en cada nodo. Este es navegable.

- Disk Block: Representa la información almacenada en los sectores, los cuales contienen las instrucciones de un programa fraccionadas dentro de estos.

- Drive Saver: Componente que tiene conocimiento de disco, encargado de la división de instrucciones en *Disk Blocks*, es el nexo entre el *HDD* y el *File System*, ya que sabe grabar a disco los programas. Al grabar a disco, retorna al *File System*, un objeto navigator que será explicado a continuación.

- Drive Navigator: Este objeto puede ser visto como un token o coupon, solo almacena la ubicación de la información en disco. El *HDD* puede devolver los *Disk Blocks* correspondientes a cada programa solo con recibir uno de estos como argumento. Este componente es recibido por el *Drive Saver* (como mencione antes, es el nexo entre los dos componentes).

En conclusión, el objetivo a la hora de implementar este módulo fue que sea lo más desacoplado posible, siendo así más extensible ante los cambios que pudiera sufrir en el transcurso del desarrollo con la incorporación de nuevas funcionalidades y características deseables.

- **Memory**

Tenemos la memoria finita representada como un Array, en donde se pueden colocar y obtener elementos segùn como fuese necesario. Tambièn posee un método de compactación útil para Asignación Continua. Se encuentran implementadas las dos polìticas aprendidas en clase: *Continuous Assignment* y *Paging*. Un detalle a mencionar, es que ambas políticas, una vez realizado su trabajo, devuelven una instancia de la clase *PolicyResult* que contiene la información necesaria para que el *MemoryManager* pueda escribir en memoria.

- Continuos Assignment: Se pueden encontrar las distintas políticas que utiliza a la hora de definir en qué lugar de la memoria se va a ubicar el siguiente bloque: *First Fit, Best Fit o Worst Fit*. Hablando sobre implementación, decidimos representar esta política como una *Double Linked List*, dado que nos simplificó mucho el trabajo de como poder mover los bloques cuando había que realizar compactación. Además cuando un bloque nuevo se creaba, era sencillo acomodar toda la estructura.

- Paging: Se encuentran implementados varios objetos para poder representar esta política. Creemos que fue una buena decisión dividir las tareas entre objetos para poder hacer el desarrollo más fácil y pueda ser testeado de manera sencilla. Contamos con la representación de *Pages*, *Frames*, existe una Tabla que lleva registro de los frames con sus correspondientes páginas asignadas, un Frame Manager que se encarga de elegir qué frame va a recibir a la siguiente pàgina, y un objeto al que denominamos *PageCreator*, que ayudará a la hora de crear las páginas para un proceso indicado.

- **Model**

Dentro de nuestro trabajo, tenemos ubicadas, dentro del paquete Model, las clases que forman parte del *Core* de todo el proyecto. Algunas de estas clases son: Kernel, CPU, Console, Clock, entre otras. Como el propósito de estas clases es bastante obvio, sólo daremos algunos detalles que tuvimos en cuenta para implementarlas, tales como la utilización de Threads en el Clock para lograr una sincronización adecuada entre la ejecución de procesos y el manejo de interrupciones, la utilización de un patrón Builder para generar interrupciones, dado un determinado PCB. Cada interrupción sabe manejarse por si misma y solo necesita que el handler le indique que lo haga.

- **Process**

Todas las clases relacionadas con la creación de PCBs, programas, entre otras cosas, están en el paquete Process. Los PCBs incluyen prioridad para que el S.T.S. (*Short Term Scheduler*) pueda poder elegir segùn la política de Prioridad. Decidimos crear un clase llamada *BlockHolder y un PageHolder* cuyo único propósito sea mantener la información del PCB con respecto a memoria y su política: que bloque tiene asignado o sus páginas. Pensamos que era mejor tener esta información en un objeto aparte que la tenga el PCB en sí. También dentro de este paquete incluimos la clase Program, dado que una vez ejecutado, se le crearía un proceso correspondiente, por lo que pensamos que ponerlo dentro del paquete Process era una buena idea.

- **Scheduling**

Utilizamos el patrón de diseño *Strategy* para esta parte del proyecto, dado que necesitábamos poder elegir que tipo de política queríamos que nuestro Scheduler siguiera: *FIFO, Priority Queue o Round Robin*. Cada política posee su implementación correspondiente, permitiendo elegir al Usuario como quisiera que su Scheduler funcione para su Sistema Operativo. Además, contamos con el L.T.S. (*Long Term Scheduler*) para poder mantener en espera a aquellos programas que agoten los recursos del mismo.

- **Tests**

Para poder corroborar que lo todo lo implementado funcione de la manera correcta y sin problemas, contamos con una gran cantidad de Tests, los cuales verifican cada parte de nuestro modelo por separado. Dentro de la carpeta Tests, se puede observar que se tiene una estructura de carpetas similar a la del proyecto, para poder mantener el orden y encontrar y/o añadir tests según como fuese necesario y según lo que se está testeando. Estos nos ahorraron bastantes dolores de cabeza durante el proceso de desarrollo ya que se hicieron en conjunto a las clases del modelo.

Problemas afrontados

Posiblemente, el problema con el que nos topamos seguido a lo largo del desarrollo, fue que algunos temas, en primera instancia, no los entendimos al cien por ciento, por lo que a la hora de programar, nos costaba más tiempo y esfuerzo poder lograr lo que queríamos.

El hecho de usar librerías externas también puede entrar en esta categoría, a pesar de que después pudimos terminar haciéndola andar como la necesitamos.

Otro punto a mencionar fue que, al querer implementar todo lo que se nos iba pidiendo, nos olvidábamos en algunos casos, de retocar lo ya implementado, para que pueda unirse a lo nuevo. Esto nos produjo en ciertos casos, que nos topemos con algunos problemas, que pudieron ser solucionados posteriormente.

Partes favoritas

- **David**: Creo que lo más me gustó del proyecto en sí, más allá del tamaño que tuvo y cómo las ideas que tenían los profesores hacían que el proyecto sea cada vez más interesante, fue la posibilidad de poder trabajar con Python por primera vez. Nunca lo habìa usado antes, y me gustó mucho poder mezclar el Paradigma Funcional con el Paradigma de los Objetos de una manera sencilla, permitiendo implementar ciertos aspectos del modelo, con tan solo una o como mucho dos líneas de còdigo.

- **Román**: Sumado a lo que dijo David, el cambio constante se nos presenta como un desafío, cambiando bastante la forma de trabajo a la que estamos acostumbrados habitualmente. En cuanto al trabajo, la parte de memoria lógica y File System fueron mis favoritas porque abstraen las ideas de sus contrapartes físicas, siendo más fáciles para razonar por el usuario.

Posibles Features

Decidimos agregar una pequeña sección para contar sobre las funcionalidades extras que esperamos poder implementar para la defensa, creemos que le dan un valor agregado al trabajo e intentaremos presentarlo el martes de ser posible.

- Consola: Le permite al usuario usar el modelo desde una línea de comando, pudiendo recorrer entre las carpetas y ejecutar programas.
- Logging: Se le permite al usuario poder ver que tareas se van realizando cuando ejecuta una acción y llevar un registro en caso de algo no salga como debe.