1:​ ​Describir cómo funciona el MMU y qué datos necesitamos para correr un proceso

MMU

El MMU es un dispositivo del hardware encargado de calcular la posición relativa de las instrucciones de un programa en la RAM y garantizar la integridad de los procedimientos.

Para ello, el sistema operativo mediante el dispatcher le avisa de la posición en memoria en la cual comienza el programa, la posición base, y tambien cual es su límite. La cpu entonces le avisa cual es la próxima instrucción a correr, y el MMU hace la cuenta: Posición Base + PR para ir a buscar la instrucción que necesite. Si la instrucción se sale de su límite, levanta una excepción.

PCB

Para correr un proceso, necesitamos poder representarlo primero. Sus atributos y variables los vamos a encapsularlas en un objeto que llamamos PCB, Process Control Block.

El primer atributo que tiene, es un id identificador el cual asigna la pcb table que lleva un conteo de todos los PCB creados.

Necesitamos también conocer cuál es la próxima instrucción a leer, lo cual guardamos en el PC Counter como atributo.

Cómo estamos manejando memoria relativa; para que el MMU opere necesitamos también que cada PCB sepa cual es la dirección base sobre la cual opera.

Finalmente, debemos poder saber cual es el estado del PCB a modo de poder poder manejar con facilidad los comportamientos que depende de tal.

Por el momento, los estados que conocemos son: New, Terminated, Ready, Waiting, Running.

2:​ ​Entender las clases IoDeviceController, PrinterIODevice y poder explicar como funcionan

PrinterIODevice

Emula un dispositivo de entrada/salida.

Es una subclase de AbstractIODevice la cual se instancia con el nombre “Printer” y el tiempo 3.

Esta, guarda en una variable interna ese nombre, ese tiempo, y setea un flag interno en falso, el cual se utiliza para saber si está siendo utilizado el device o no. Lo que nos interesa realmente de esta clase son sus 2 métodos: Ejecutar:​ ​Recibe una instrucción de I/O, si ya se estaba utilizando el device, levanta una excepción avisando. Caso contrario, cambia su estado a ocupado, setea una variable interna de ticks count en 0 y guarda la operación como la actual.

IoDeviceController

Emula un driver de I/O. Guarda como atributo un device al cual está asociado, su cola de waiting, y el PCB que está corriendo en ese momento en el Device.

runOperation:​Recibe un Pcb junto con la instrucción de I/O con la que llega desde el Handler. Genera un pair con estos dos y los envía a la cola de waiting. Finalmente, se envia a sí mismo el mensaje de cargar un programa en el device si es posible. (Lo cual es un poco sospechoso, no es mejor uso de recursos enviarlo directamente a correr si la lista está vacía?)

\_\_ load\_From\_Waiting\_Queue\_If\_Apply:​ Se fija si el dispositivo está ocioso y si tiene algún programa en la lista de esperando. Si lo hace, hace un pop del primer procedimiento esperando en la lista, y le pasa la instrucción y pcb al device para que comience a trabajar con el.

3: ​Explicar cómo se llegan a ejecutar IoInInterruptionHandler.execute() y IoOutInterruptionHandler.execute()

Cuando a la CPU le llega para que procese una instrucción de I/O, genera una interrupción de I/O in y se la envía al vector de interrupciones, el cual evalúa esta interrupción y le avisa al sistema operativo por medio del handler.

I/O out en cambio, también llega a través del vector de interrupciones, pero lo envía en dispositivo de I/O una vez terminó de procesar la instrucción que tenía a cargo.

4:

4.1​:​ Qué está haciendo el CPU mientras se ejecuta una operación de I/O??

Lee, interpreta y ejecuta instrucciones. La cpu tiene una responsabilidad muy acotada y no sabe hacer mucho más que eso; siempre y cuando tenga una dirección la cual patear al MMU y le lleguen instrucciones, es lo que va a seguir haciendo. Tanto el CPU como el I/O trabajan en paralelo, son dos unidades de procesamiento distintas.

4.2:​ ​Si la ejecución de una operación de I/O (en un device) tarda 3 "ticks", cuantos ticks necesitamos para ejecutar el siguiente batch?? Cómo podemos mejorarlo?? (tener en cuenta que en el emulador consumimos 1 tick para mandar a ejecutar la operación a I/O)

prg1 = Program("prg1.exe", [ASM.CPU(2), ASM.IO(), ASM.CPU(3), ASM.IO(), ASM.CPU(2)]) prg2 = Program("prg2.exe", [ASM.CPU(4), ASM.IO(), ASM.CPU(1)]) prg3 = Program("prg3.exe", [ASM.CPU(3)])

Bueno, si no tenemos multiprogramación y no podemos conseguir que la cpu y el I/O device trabajen en paralelo (osea, cuando llegue una instrucción de IO, la cpu siga corriendo otro programa mientras el programa que se fue a hacer I/O hace su función dentro del segundo dispositivo). Necesitariamos: Asm.cpu totales=15 Asm.io totales= 3 Tenemos 18 ticks con eso + los 3x3 ticks de la cpu esperando por el io, serian 27 ticks totales. Con multiprogramación nos ahorramos 9 ticks