PREGUNTAS TEÓRICAS

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

MMU

Es un dispositivo del hardware encargado de calcular la posición física de las instrucciones de un programa en la ram.
Cuando el CPU realiza un fetch para acceder a una dirección en memoria, envía la dirección virtual (pc) al MMU y este le suma la baseDir de ese programa para que la CPU pueda acceder al lugar correspondiente en la memoria ram.

PCB

Para correr un proceso, necesitamos tener guardados sus atributos, los cuales vamos a encapsularlas en un objeto que llamamos PCB (Process Control Block).
Sus atributos son:
pid: es un id identificador el cual asigna la pcb table que lleva un conteo de todos los PCB creados, los cuales son únicos.
baseDir: la cual la proporciona el loader, la baseDir es en qué dirección comienza el proceso actual.
pc: Dice cual es la próxima dirección a leer
state: Guarda el estado del proceso para poder manejar su comportamiento (New, Ready, Running, Waiting, Terminated)

2) Entender las clases IoDeviceController, PrinterIODevice y poder explicar cómo funcionan.

La clase **IoDeviceController** se encarga de controlar las operaciones de entrada/salida (I/O) en un dispositivo específico.

Se inicia:
El dispositivo (device) que controlará.
Una cola de espera (_waiting_queue) para almacenar las solicitudes de operaciones de I/O pendientes.
Una variable _currentPCB que representa el PCB (Control Block Process) del proceso actualmente en ejecución en el dispositivo I/O.

En el método **runOperation(self, pcb, instruction):**
Este método se utiliza para ejecutar una operación de I/O para un proceso específico (pcb), con una instrucción dada (instruction).
Primero, se crea un diccionario (pair) que contiene el PCB y la instrucción.
El pair se agrega al final de la cola de espera (_waiting_queue).
Después, se intenta enviar la instrucción al dispositivo hardware si este se encuentra en estado "idle" (inactivo).

getFinishedPCB(self):
Este método se utiliza para obtener el PCB del proceso que ha terminado su operación de I/O en el dispositivo.
Devuelve el PCB del proceso que ha finalizado su operación de I/O y luego se prepara para recibir un nuevo proceso.

__load_from_waiting_queue_if_apply(self):
si hay algo en la cola de espera y si el dispositivo está inactivo.
Se coloca en una variable el primer elemento de la cola de espera. (y se elimina de la cola)
Se crean dos variables, una con el pcb y otra con la instrucción
Luego, se obtiene el PCB y la instrucción de la solicitud y se ejecuta la instrucción en el dispositivo hardware y se coloca el pcb que está siendo ejecutado ahora.
La clase IoDeviceController administra las operaciones de I/O que están en espera, asegurándose de que se ejecuten cuando el dispositivo esté disponible. Garantiza que la multiprogramación funcione correctamente y que los procesos no queden inactivos esperando operaciones de I/O.

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

Cuando la CPU decodifica una instrucción de tipo 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 al cual se le transfieren los datos del proceso (se hacen cambios de estados de procesos, etc) y ya el control no le pertenece al programa, sino al dispositivo de I/O.

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

4.1) **¿Qué está haciendo el CPU mientras se ejecuta una operación de I/O?**

Sigue leyendo, interpretando y ejecutando instrucción siempre y cuando mientras cada vez que haga un fetch al MMU se traiga consigo el pc de la siguiente instrucción a leer en memoria, pueden trabajar en paralelo en diferentes cosas el CPU y el I/O ya que son dos cosas diferentes, si no hay más instrucciones a procesar el CPU se quedará esperando el I/O, ósea haciendo nada(no es lo mas optimo).

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)])

Para ejecutar completamente vamos a necesitar 3 “ticks” + 1 “tick” para enviar la operacion osea un total de 4 por instrucción de I/O
Para calcular cuantos ticks necesitamos para pasar al siguiente batch (conjunto de programas) necesitamos saber la duración del batch actual.
-prg1.exe = 2 (CPU) + 4 (I/O) + 3(CPU) + 4 (I/O) + 2 (CPU) = 15 "ticks".
-prg2.exe = 4 (CPU) + 4 (I/O) + 1 (CPU) = 9 "ticks".
-prg3.exe = 3 (CPU) = 3 “ticks”.
La suma de todo da =
15 (prg1) + 9 (prg2) + 3 (prg3) = 27 ticks son necesarios para ejecutar el siguiente batch

Se podría mejorar con multiprogramación(se tiene varios programas en memoria al mismo tiempo, de manera que la CPU puede cambiar de un programa a otro mientras espera por el de entrada/salida (I/O), lo que permite que múltiples programas se ejecuten concurrentemente.), así mientras se ejecuta la instruccions de I/O, la cpu sigue con el próximo programa