eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Grado de Ingeniería Informática

PROYECTO SIMULADOR DE UN KERNEL

Autor: Gorka Dabó

Fecha: 20/06/2024

Índice

1. Introducción

   Este informe detalla el desarrollo y la implementación de un simulador de sistema operativo multi-hilo que gestiona múltiples procesos en un entorno simulado de múltiples CPUs, cada una con múltiples núcleos y múltiples hilos de ejecución.

   El proyecto está centrado sobre todo en la creación de un scheduler eficiente que utiliza un mecanismo de planificación de tipo round-robin para distribuir equitativamente las tareas entre los hilos disponibles, asegurando una gestión de procesos justa y efectiva.

   Además, se han implementado mecanismos para la sincronización y comunicación entre hilos, como condiciones y señales, que permiten a los procesos sincronizarse con los ciclos del reloj y los intervalos del scheduler.

2. Fundamentos teóricos

   En este apartado se detallan brevemente los fundamentos teóricos que componen el diseño y la implementación del simulador del sistema operativo descrito en este informe.

   a. Reloj: El reloj simula el paso del tiempo dentro del sistema, generando pulsos que activan la señalización necesaria para el avance de los ciclos de ejecución de los procesos.

   b. Temporizador: El temporizador trabaja con el reloj para gestionar intervalos específicos, determinando cuándo se deben activar eventos como la planificación de procesos, asegurando que se cumplan los tiempos de respuesta.

   c. PCB: Estructura de datos que guarda toda la información necesaria sobre los procesos, incluyendo el estado del proceso, el contador de programa (PC), y otros detalles de gestión.
      i. mm: Representa el manejo de memoria del proceso en el PCB, incluyendo direcciones de segmentos de código y datos, así como la tabla de páginas para la traducción de direcciones.

d. Máquina: Simula un sistema informático completo con múltiples CPUs, cada una con varios núcleos e hilos de ejecución, permitiendo la simulación de un entorno de procesamiento paralelo.

    i. CPU: Cada CPU en la máquina puede tener múltiples núcleos y ejecutar varios hilos de proceso.

    ii. Core: Cada core en una CPU puede ejecutar uno o más hilos de hardware simultáneamente.

    iii. Hilo Hardware: Unidad de ejecución más pequeña dentro del core que puede ejecutar un proceso asignado.

        1. PC: Registra la dirección de la siguiente instrucción del programa a ejecutar.

        2. IR: Almacena la instrucción actualmente en ejecución.

        3. PTBR: Puntero a la tabla de páginas del proceso actual en ejecución.

        4. MMU: Gestiona la traducción de direcciones de memoria virtual a física mediante el uso de tablas de páginas y TLB.

            a. TLB: Caché que almacena traducciones de direcciones recientes de memoria virtual a física para acelerar el acceso a memoria.

e. Memoria Virtual: En nuestro proyecto, la memoria virtual se representa mediante un fichero que simula el espacio de direcciones accesible por los procesos

f. Memoria Física: En nuestro simulador, la memoria física está dividida en dos secciones. La primera, destinada al espacio de usuario, almacena datos e instrucciones de los programas en ejecución. La segunda parte, asignada al espacio del kernel, se utiliza para almacenar las tablas de páginas que gestionan la traducción de direcciones en el sistema de memoria virtual.

3. Desarrollo del Kernel

En este apartado, explicaremos cómo se han implementado los distintos componentes del simulador del sistema operativo. Detallaremos la estructura del código y su función dentro del conjunto del sistema.

Además, trataremos las principales dificultades encontradas durante el proceso de desarrollo y cómo se han resuelto estos desafíos. Leer este apartado le permitirá comprender mejor las decisiones de diseño tomadas y los mecanismos internos que permiten al simulador funcionar.

a. main.c:
Este archivo es el que contiene la función principal del programa. Recibe los parámetros de los intervalos, el quantum, el número de cpus, cores e hilos para luego contar el número de archivos elf que se encuentran en la carpeta correspondiente.

Con esta información, el programa establece la configuración inicial necesaria para realizar la simulación del sistema operativo. Se inicializan las estructuras para la gestión de la memoria física y la planificación de procesos.

La función main también se encarga de lanzar los hilos que manejan el reloj del sistema, el temporizador y la distribución del scheduler, asegurando que el sistema se ejecute de manera coordinada.

Una vez configurado, el sistema carga los programas desde los archivos ELF, distribuyéndolos entre los distintos núcleos e hilos según la capacidad disponible.

Finalmente, el main espera a que todos los hilos terminen su ejecución antes de liberar los recursos utilizados y terminar la ejecución del programa.

Al ser una función básica y poco compleja, no me ha traído ninguna dificultad más que la implementación de la función encargada de contar el número de archivos .elf ya que no conocía las funciones necesarias para abrir un directorio y leerlo, y tampoco sabía de la existencia de la estructura dirent* que permite la iteración por los archivos en un directorio.

a. commons.h:

Este es el archivo que contiene las definiciones utilizadas a través de todo el proyecto. Sirve para centralizar la declaración de constantes relacionadas con la memoria y la paginación, y declarar como externas las variables globales y estructuras fundamentales del sistema.

Contiene también las definiciones de macros que afectan el tamaño de la memoria y la forma en que se maneja la paginación, asegurando una configuración centralizada que facilita la gestión y el mantenimiento del código.

b. structs.h:

Este archivo es crucial en el proyecto, ya que define todas las estructuras de datos utilizadas en la simulación del sistema operativo. Contiene definiciones para la memoria física, configuración del sistema, manejo de la memoria de procesos (PCB), colas de procesos, y estructuras para la administración del estado del planificador y las unidades de manejo de memoria (MMU).

Además, incluye estructuras para la representación de la CPU, sus núcleos (cores), y los hilos de hardware, además de las tablas de páginas y directorios de páginas necesarios para la implementación de la memoria virtual.

Cada estructura está diseñada para contener los diversos componentes y estados necesarios para la ejecución y manejo de procesos.

c. clock_timer.c:

Las funciones de clock_function y timer_function que se encuentran en este fichero, son las encargadas de la gestión del tiempo en el simulador del sistema operativo. clock_function genera pulsos de reloj regulares que simulan el paso del tiempo en el sistema, mientras que timer_function espera estos pulsos y, al alcanzar un intervalo configurado, emite señales para activar el scheduler.

Para ello, el clock tiene un bucle infinito que emite pulsos constantemente mediante señales, y la función del timer, que también tiene un bucle infinito, tiene un contador que actualiza cada vez que recibe un pulso para compararlo con el intervalo configurado y emitir la señal al scheduler.

Estas funciones no han traído ninguna dificultad más que entender su funcionamiento y aprender a utilizar las señales correctamente.

d. machine.c:

Este archivo contiene la función iniciarMaquina, que inicializa la estructura completa de la máquina simulada, compuesta por CPUs, cores e hilos de hardware. Al ejecutar esta función, se asignan dinámicamente los recursos necesarios para cada componente del sistema, estableciendo la cantidad de CPUs, el número de cores por CPU y los hilos por core según los parámetros proporcionados. Además, se inicializa el estado inicial de cada hilo de hardware, y llama a la función encargada de inicializar la MMU de cada hilo.

Esta función no ha tenido ninguna dificultad ya que se trata de una sencilla función de inicialización.

e. physical_memory.c:

Este archivo gestiona la memoria física del sistema simulado. La función initPhysicalMemory se encarga de asignar e inicializar la memoria física que será utilizada tanto por el espacio de usuario como por el núcleo del sistema. Se reserva un bloque de memoria que se divide en dos partes: dos tercios para el espacio de usuario y un tercio para el núcleo, el cual almacena las tablas de páginas.

Por otro lado, se encarga de llamar a la función que inicializa el directorio de las tablas de páginas. En cuanto a las dificultades de este apartado, en un inicio no tenía claro cuál podía ser una división adecuada para la parte del usuario y la del kernel, pero la decisión de dividirla en tercios ha sido adecuada ya que nos brinda espacio suficiente para ambas partes para los objetivos del simulador.

f. process.c:

Este archivo es un pilar fundamental en la gestión de procesos dentro del simulador de sistema operativo. Define las funciones que manejan la carga de programas, la asignación de memoria y la administración de los procesos. A continuación se detallan sus funciones principales:

i. initializeUserPages(): Esta función inicializa el estado de las páginas de usuario, marcándolas como libres para su uso posterior. Esto es crucial para gestionar el espacio de memoria que será asignado a los programas cargados.

ii. getFreePhysicalMemoryInUserSpace(int pagesNeeded): Busca y reserva bloques de memoria física necesarios para cargar los procesos. Esta función es fundamental para la simulación del manejo de memoria en un sistema operativo, buscando secuencias contiguas de páginas libres que satisfagan la solicitud de memoria del proceso.

iii. getNextPID(): Genera y retorna un identificador único para cada proceso, asegurando que cada proceso tenga un identificador distinto.

iv. loader(int elfFileCount): Esta función se encarga de la carga de los archivos ELF (Executable and Linkable Format). Para cada archivo, abre el archivo, lee las direcciones de inicio de los segmentos de texto y datos, y carga el contenido del archivo en la memoria asignada. Gestiona también la creación de las tablas de páginas necesarias para el manejo de la memoria virtual del proceso.

Este conjunto de funciones permite simular aspectos de la gestión de procesos y memoria en un sistema operativo, proporcionando una base importante para entender cómo los sistemas operativos manejan múltiples procesos y su memoria en un entorno controlado.

En cuanto a las dificultades encontradas en este apartado, han sido varias y muy distintas. Para empezar, en distintas funciones en los que hago operaciones con punteros, no era consciente de la existencia de la aritmética de punteros, por lo que pensaba que para un puntero de por ejemplo uint32_t * tamaño, para hacer que avanzara 4 bytes, habría que sumarle ese puntero multiplicado por cuatro, pero al ser un puntero, al sumarle un valor ya se tiene en cuenta el tamaño del tipo al que apunta. Es decir, sumar uno a un puntero de tipo uint32_t * automáticamente incrementa la dirección en 4 bytes, que es el tamaño de un uint32_t. No obstante, esto solo pasa con punteros y no con variables normales.

Por otro lado, la función que reserva bloques de memoria en el espacio de usuario me fue difícil de implementar porque no se me ocurría una forma sencilla y óptima para representar que paginas estaban reservadas y cuáles no, y porque no entendía del todo el concepto de página ya que inicialmente creía que de alguna manera, habría que hacer malloc() para reservar las páginas sobre la memoria ya reservada.

El loader también me trajo sus problemas. Por un lado, no sabía como hacer para cargar todos los archivos .elf, y al final me decanté por meterlos todos en una carpeta que estuviera dentro de la carpeta donde se ejecuta el simulador.

Tampoco tenía claro si el loader tenía que ser ejecutado por varios hilos para que se fueran cargando los hilos ‘dinámicamente’ a medida de que hubiera hilos libres, pero acabé dándome cuenta de que no tenía mucho sentido ya que el objetivo de la función es cargar todos los programas en memoria y generar los PCBs correspondientes para luego colarlos en la cola de procesos.

Por otro lado, también tuve que familiarizarme con las funciones encargadas de leer el fichero para cargar su contenido en buffers. En cuanto a la lógica implementada para leer los archivos .elf, primero almacena las direcciones virtuales iniciales de los segmentos de texto e

instrucciones para luego contar el número de instrucciones y datos totales. Una vez hecho esto, reiniciar el puntero del archivo, y tras saltar las primeras dos filas, calculo el número total de páginas necesarias para almacenar las instrucciones y los datos.

Con esta información, utilizo la función encargada de reservar el espacio para las páginas, y con la dirección física inicial, almaceno las instrucciones y los datos en memoria.

En cuanto a los PCB, les asigno un PID unico, apunto los punteros de data y code a los valores leídos desde el fichero, y añado el pcb a la cola de PCBs, no sin antes inicializar la tabla de páginas correspondiente al proceso. Esta última función, la trataremos más adelante.

g. page_table.c:

El archivo page_table.c contiene las funciones necesarias para la gestión de tablas de páginas en un simulador de sistema operativo. Las funciones incluidas en este archivo son cruciales para el mapeo de direcciones virtuales a direcciones físicas, un componente esencial de la gestión de memoria en sistemas operativos.

initPageTable(PageTable *pt, int numEntries, uint32_t *base): Esta función inicializa una tabla de páginas. Se asigna un bloque de memoria para las entradas de la tabla y se configuran todas las entradas como no válidas. Esto prepara la tabla para ser llenada con direcciones físicas válidas conforme se asignen páginas.

initPageDirectory(int numTables, uint32_t *kernelBase): Prepara el directorio de páginas del sistema, que contiene múltiples tablas de páginas. Se inicializa cada tabla con una base específica, empezando desde la parte reservada para el kernel y reservando la memoria para las tablas consecutivamente teniendo en cuenta el número de tablas necesarias para el proceso en cuestión.

initPageTableForProcess(size_t totalEntries, uint32_t* programSegment): Esta función es utilizada para completar las tablas de páginas para un nuevo proceso. Busca espacio libre en las tablas de páginas del directorio utilizando el bit de validez y configura las entradas necesarias para mapear el segmento de programa del proceso.

Es esencial para la carga y ejecución de programas dentro del entorno del simulador, permitiendo la separación y protección de la memoria entre diferentes procesos.

En cuanto a las dificultades encontradas a la hora de implementar las tablas de páginas, han sido varias. Para empezar, no entendía correctamente que era lo que almacenaban las tablas de páginas ya que creía que almacenaban tanto la dirección virtual como la física hasta que entendí que solamente almacenan la dirección física y que la entrada correspondiente a la dirección virtual a traducir, se calcula a partir de una máscara derivada del tamaño de las páginas.

Por otro lado, creía que en vez de almacenar la dirección física inicial de las páginas que han sido reservadas para el proceso, se almacenaban las direcciones físicas correspondientes a todas las direcciones virtuales de un proceso, lo cual además de incorrecto, es inefectivo.
En cuanto a la implementación de las funciones, las primeras dos funciones de inicialización no me trajeron muchos problemas, pero la función encargada de completar la tabla de páginas, me trajo más complicaciones. El bucle que busca el número de tablas libres necesarias para almacenar todas las entradas no me trajo grandes complicaciones debido a que su implementación es muy parecida a la realizada en la función de getFreePhysicalMemoryInUserSpace() en process.c, pero hice un fallo importante que me costó bastante arreglarlo a pesar de ser realmente sencillo.

El problema era que a la hora de marcar los espacios utilizados en la tabla de páginas, al principio no consideraba el estado de ocupación de la tabla completa (el bit de occupied). Esto provocó que todas las entradas de diferentes procesos, que cabían dentro de una única tabla de páginas, terminaran siendo almacenadas en la misma tabla.

La razón del problema era que únicamente estaba utilizando el bit de validez para cada entrada, sin marcar la tabla de páginas como ocupada una vez que se asignaba a un proceso. La solución fue introducir un registro booleano llamado occupied para las tablas de páginas en el directorio de páginas, asegurando que cada vez que una tabla fuera utilizada, se marcase como tal.

h. scheduler.c:

El archivo scheduler.c contiene el núcleo del despachador de procesos del simulador, gestionando la asignación de procesos a hilos de hardware disponibles en un sistema emulado con múltiples CPUs y núcleos.

La función de void* scheduler_dispatcher() es un bucle infinito que espera señales de reloj (ticks) para despachar procesos. Utiliza una variable pulseCount para contar los intervalos de tiempo y decidir cuándo ejecutar el programa correspondiente.

Para asegurar un uso equitativo de los recursos, se utiliza una estrategia de rotación para seleccionar hilos. Esta estrategia me permite que cada hilo tenga la oportunidad de ejecutar un proceso sin favorecer a ninguno en particular.

Dentro del bucle, se verifica la disponibilidad de los hilos de hardware. Si se encuentra un hilo libre, se obtiene un proceso de la cola (dequeuePCB()), se inicializa su gestión de memoria y se lanza su ejecución en un hilo separado. Esto lo he manejado a través de la creación de una estructura ExecutionArgs, que se pasa a pthread_create() para iniciar la función threadedExecution().

Además, si todos los hilos están ocupados, se imprime un mensaje indicando que no hay hilos disponibles.

Al no ser una política muy complicada, a la hora de implementarla no he tenido grandes complicaciones. A pesar de todo, la parte de pasar los argumentos a la función que inicia la ejecución del programa si que me trajo problemas ya que no sabía cómo había que hacer para pasar los argumentos a la función que ejecuta el hilo pthread creado.

Esto me llevó a tener que crear una nueva estructura que almacena los parámetros necesarios para ejecutar la función por un lado, y a crear una función que es la llamada por el hilo, la cual se encarga de que la estructura apunte a los parámetros y luego llamar a la función encargada de ejecutar los procesos extrayendo los parámetros de la estructura y pasandoselos como parámetro. Es posible que hubiera una solución más efectiva o clara pero no logré encontrar una manera mejor que la mencionada anteriormente.

Por otro lado, otro problema que tuve con el scheduler es que en un principio, una vez se cumplía el intervalo, recorría el vector en busca de hilos libres y cada vez que habría uno libre le asignaba un proceso. Esto hacía que con cumplir el intervalo una vez la mitad de los procesos estuvieran ejecutados y con cumplir el intervalo 2 veces todos los procesos habían sido ejecutados.

Al no ser este el comportamiento que buscaba, la solución fue añadir un break; al final después de haber generado el hilo para ejecutar el proceso, y de esa manera, para poder ejecutar otro, tenía que cumplirse el intervalo otra vez. De esta forma, cada vez que se cumple el intervalo, se ejecuta como mucho un proceso.

La parte mala de esa implementación era, que a pesar de probar con los intervalos más bajos posibles, para cuando se cumplía de nuevo el intervalo, el primer hilo ya había terminado con su ejecución por lo que siempre el mismo hilo era el que ejecutaba todos los procesos (a excepción de que en algún caso concreto el segundo hilo ejecutaba algún proceso cuando al primero le tocaba alguno largo).

Es por esto que acabé implementando la política de round robin para no saturar siempre el mismo hilo y dar una oportunidad equitativa a todos los hilos disponibles en el sistema.

i. utilities.c:

El archivo utilities.c juega un papel imprescindible en la infraestructura del simulador, conteniendo funciones auxiliares cruciales que permiten la ejecución del sistema. Una de las funciones más significativas es countElfFiles, que escanea un directorio específico para contar archivos con la extensión .elf, utilizados en este caso como programas ejecutables para el simulador.

Esta función emplea las funciones pertinentes para leer el contenido del directorio, filtrando los archivos por su extensión, lo cual es importante para filtrar los archivos adecuados para la inicialización del simulador, ya que determina la cantidad de programas a cargar.

La función iniProcessQueue inicializa la cola de procesos, preparándose para contener los PCBs de los programas que serán ejecutados. La implementación reserva memoria para la cola basada en la cantidad de archivos .elf encontrados, estableciendo así una relación directa entre los programas disponibles y el espacio de almacenamiento reservado.

enqueuePCB y dequeuePCB gestionan la adición y eliminación de procesos en la cola, respectivamente. enqueuePCB inserta un nuevo PCB al final de la cola si hay espacio disponible, mientras que dequeuePCB retira y retorna el primer PCB de la cola, reorganizando el resto de elementos para llenar el espacio vacío.

Finalmente, iniciarEjecucion es la función que lleva a cabo la ejecución de los procesos. Inicia estableciendo la dirección de memoria inicial para la ejecución almacenándolo en el registro PC, basándose en la dirección virtual de la primera instrucción contenida en el PCB del proceso.

La ejecución se realiza mediante un bucle que interpreta y ejecuta instrucciones específicas hasta que se encuentra la instrucción de terminación (exit) analizando sus opcodes.

Durante este proceso, se hacen traducciones de direcciones virtuales a físicas usando la MMU y la TLB para acceder y manipular datos correctamente en la memoria física. Este proceso de ejecución es fundamental para simular el comportamiento de los programas en un sistema operativo.

En cuanto a la dificultad de este apartado, las funciones de inicialización y gestión de la cola no me han traído ningún problema; mientras que la función de iniciar la ejecución no me ha traído más que problemas.

Para empezar, como el enunciado indica que la memoria física tendría un bus de direcciones de 24 bytes y posiciones de tamaño de una palabra, que son 4 bytes, lo cual se puede almacenar en un uint32_t, a lo largo de todo mi código había utilizado tanto punteros como variables normales de tamaño uint32_t. En la implementación de la función iniciarEjecución, el registro IR y PC eran variables de tipo uin32_t, por lo que a pesar de que al almacenar la dirección física en el PC no me daba problemas, a la hora de castear un puntero (también de tipo uint32_t) al valor del PC y diferenciarlo para almacenar el contenido en IR, siempre me daba SEGMENTATION FAULT.

Inicialmente, creía que el problema estaba en cómo había almacenado los datos en la memoria por lo que estuve mucho tiempo pensando y analizando las funciones encargadas de almacenar los datos y las instrucciones en memoria, pero analizándolo con el debugger de Clion (el IDE que estaba utilizando) todo parecía funcionar perfecto, y además, no saltaba el SEGMENTATION FAULT lo cual complicaba aún más las cosas.

Después de asegurarse de que no era un problema de cómo almacenar la información en memoria, me centré en el tema de los punteros, dobles punteros y como desreferenciarlos; pero a pesar de dedicarle mucho tiempo, no lograba encontrar la solución. Fue entonces cuando se me ocurrió probar ejecutar el programa en un ordenador linux de la universidad (hasta ese momento estaba utilizando Windows), y lo mismo me ocurría tanto al ejecutar el programa.

Llegados a este punto, probé debuggear una vez más con gdb para acabar descubriendo que la dirección que se almacenaba en el registro PC era la última mitad de la dirección física en la que se encontraba el dato en memoria y fue entonces cuando me di cuenta de que estaba utilizando punteros de 32 bits cuando el ordenador utilizaba direcciones de 64 bits.

Una vez detectado el problema, sustituí los punteros de 32 bits que apuntaban a secciones de la memoria física por punteros de 64 bits (uint64_t *). Gracias a esta modificación, la dirección completa se almacenaba en la memoria, y el valor se almacenaba correctamente en el registro IR. Lo que a día de hoy sigo sin entender, es porque a la hora de debugear no saltaba el segmentation fault y los valores se cargaban correctamente mientras que al ejecutar de forma normal no (lo cual sí tenía sentido).

j. mmu.c:

El archivo mmu.c es fundamental en el simulador para manejar la unidad de gestión de memoria (MMU), que es clave en sistemas operativos modernos para la traducción de direcciones de memoria virtual a física y para mejorar la seguridad y eficiencia de la memoria.

La función initMMU inicializa la MMU de un hilo hardware específico, configurando su TLB para un estado inicial limpio y preparándola para su uso. Esto se hace a través de la función initTLB, que establece todas las entradas de la TLB como inválidas, indicando que no hay traducciones de direcciones activas al inicio.

La función getNextTLBIndex es una función dentro de la MMU para obtener el próximo índice de la TLB en un enfoque circular, asegurando que todas las entradas sean utilizadas equitativamente y manteniendo la TLB como una cache de tamaño fijo que trabaja bajo una política de reemplazo cíclico.

translateAddress es la función más crítica en este archivo, la responsable de traducir una dirección virtual en una física. Primero, intenta encontrar una traducción válida directamente en la TLB, lo que sería un "hit" de TLB, devolviendo la dirección física correspondiente directamente.

Si no se encuentra la entrada, ocurre un "miss" de TLB, y la función calcula la dirección física basándose en la base de la tabla de páginas del proceso actual, actualizando la TLB con esta nueva traducción para usos futuros.

En cuanto a las dificultades encontradas en este apartado, en un principio no entendía correctamente cuál debía ser su funcionamiento ya que no sabía cómo extraer la dirección física a partir de la virtual. Después de buscar información y leer en foros, entendí que la tabla de páginas del proceso es apuntada por el PTBR, por lo que sabía donde buscar la dirección física, pero no entendía cómo saber cuál era la entrada correspondiente a la dirección virtual en la tabla de páginas.

Acabé comprendiendo que para ello, hay que crear una máscara que al ser mis páginas de direcciones de 8 bits y contenido de 4 bytes (2 bits para representarlos), Para la máscara tendría que poner los primeros 10 bits (empezando por la derecha) a 1, de esta forma, al aplicar la operación AND entre la dirección virtual y la máscara, se obtienen únicamente los bits que representan el desplazamiento dentro de la página, mientras que los bits restantes que representan el índice de la página se pueden obtener desplazando la dirección a la derecha por la cantidad de bits del desplazamiento.

Una vez entendido el funcionamiento, puede implementar la función correctamente casi en su totalidad. Después de desarrollarlo y ejecutar el programa, ví que en algunos .elf, había un momento en el que al cargar el dato la dirección física del mismo se cargaba incorrectamente y cargaba una instrucción como dato o algún dato incorrecto. El problema era que en la función de traducción de dirección desarrollada en la mmu, a la hora de sumar el offset a la dirección base, en vez de hacer una suma '+', realizaba una operación de or '|', por lo que en el caso de hacer 1 or 1 el resultado es 1 pero no se tiene en cuenta la llevada, lo que daba como resultado una dirección incorrecta.

Otro punto a tener en cuenta, es que inicialmente almacenaba las instrucciones y los datos en páginas diferentes, pero esto traía problemas ya que tal y como tenía implementada la función de traducción de direcciones, a la dirección virtual le correspondía la misma página que a las instrucciones ya que cabían en la misma página. Para solucionar este problema, simplemente cambié la forma en la que cargaba los datos en la memoria para que se cargaran contiguamente con las instrucciones.

k. tlb.c:

El archivo tlb.c contiene la implementación para inicializar una tabla de búsqueda rápida (Translation Lookaside Buffer) utilizada en la gestión de memoria de un sistema operativo simulado. La función initTLB(TLB *tlb) se encarga de establecer los valores iniciales de la TLB.

4. Conclusiones sobre el proyecto

La conclusión principal que saco después de hacer este proyecto, es que esta forma de aprender realizando proyectos grandes es la mejor opción, ya que he aprendido mucho más profundamente ciertos aspectos de los sistemas operativos que en un principio había estudiado para los exámenes, pero ya no recordaba.

Estoy seguro de que después de realizar el proyecto, estos conceptos los tengo mucho más interiorizados y como consecuencia, será mucho más difícil que se me olviden.

Por otro lado, considero que mis habilidades en C se han visto claramente mejoradas por todas las horas invertidas en el proyecto, ya que ciertos conceptos como los punteros, o los problemas que acarrea el hecho de devolver

la dirección de una variable local en una función, no los tenía bien trabajados hasta ahora.

Para finalizar, creo que puedo estar seguro de que si algo he aprendido en este proyecto es a depurar el código (sobre todo con gdb), ya que las largas horas dedicadas a la depuración, me han enseñado a entender mejor, analizar y a identificar errores de forma mucho más efectiva y rápida.