Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

Trabajo Práctico Final: DeliriOS

Organización del Computador II

Marzo 2014

Apellido y Nombre	LU	E-mail
Silvio Vileriño	106/12	svilerino@gmail.com
Ezequiel Gambaccini	xxx/xx	xxx@xxx.com

Índice

1. Introduccion

El objetivo de este trabajo práctico fue inicialmente experimentar con la arquitectura intel, realizando un microkernel de 64 bits inicializando multinúcleo. Más tarde en el desarrollo del proyecto se decidió extender el alcance del mismo, realizando algunos experimentos para analizar las posibles mejoras de rendimiento de algoritmos que se pueden paralelizar en varios núcleos. La ganancia que esperamos obtener, es una reducción considerable en el overhead que genera el manejo de varios hilos sobre sistemas operativos utilizando librerias como por ejemplo pthreads, y de esta forma poder determinar si podría aprovecharse de mejor manera el hardware disponible para resolver problemas de mayor tamaño que el que nos permiten las librerias actuales para multihilo.

El informe estará dividido en secciones, cada una describiendo una parte del trabajo.

2. Desarrollo: Inicialización y contexto del sistema

2.1. Estructura de carpetas: Compilación, Linkeo y Scripts

- **ap code:** Contiene el código de los Application processors, tanto de inicialización desde modo protegido a modo nativo x64 como parte del código de los algoritmos implementados para los experimentos.
- **ap startup code:** Contiene el código de inicializacion de modo real a modo protegido de los Application processors
- **bsp code:** Contiene el código de booteo del kernel principal de nivel 2, parte del código de los algoritmos implementados, y el encendido por parte del BSP de los Application Processors.
- **common code:** Contiene el código común al Bootstrap Processor y los Application Processors.
- **grub init:** Contiene scripts y archivos de configuracion de grub, en la subcarpeta src se encuentra el primer nivel de booteo que realiza el pasaje entre la máquina en estado mencionado en la especificación multiboot al segundo nivel de booteo del BSP.
- **run.sh:** Script para compilación y distribución del tp.
- **macros:** Esta carpeta contiene macros utilizadas en el código.
- **informe:** Contiene el informe del trabajo práctico en formato Latex y PDF.
- **include:** Contiene las cabeceras de las librerias incluidas en el código.

El tp está compilado en varios ejecutables, de esta forma cargamos algunas partes como módulos de grub. Hay scripts encargados de compilar todo lo necesario, cada módulo tiene su Makefile y el comando make es llamado oportunamente por los scripts en caso de ser necesario.

El linkeo está realizado con linking scripts en las carpetas donde sea necesario, cada sección esta alineada a 4k por temas de compatibilidad, asimismo hay símbolos que pueden ser leidos desde el codigo si es necesario saber la ubicación de estas secciones en memoria. Los módulos de 32 bits estan compilados en formato elf32 y los de 64 bits en binario plano de 64 bits, esto se debe a temas de compatibilidad de grub al cargar modulos y kernels.

Para correr el trabajo practico unicamente hace falta tipear $./run.sh - r$ en consola y se recompilará automaticamente para luego ejecutarse en bochs.

2.2. Integración con grub y división en módulos

Utilizamos grub en el nivel mas bajo del booteo para lograr un contexto inicial estable y que posibilitara la ejecución del trabajo practico desde un pendrive usb.

Grub permite iniciar un kernel por medio de una especificación publicada en la web, en la que se detalla un contrato que debe cumplir tanto el kernel a iniciar como grub, entre ellas, un header que debe contener el ejecutable del kernel para ser identificado por grub como un sistema operativo, y por otro lado, las obligaciones que debe cumplir grub al entregarle el control a dicho kernel, es decir, un contexto determinado: selectores de código y datos válidos, registros de propósito general con valores específicos, etc.
(`http://www.gnu.org/software/grub/manual/multiboot/multiboot.html#Machine-state`)

Esta revisión de grub no permite cargar ejecutables compilados en 64 bits de manera sencilla, por este motivo decidimos que lo mejor era utilizar a una herramienta provista por grub, llamada carga de módulos, que nos permitió cargar distintas partes no críticas del sistema por encima del primer mega de memoria y tener en un mapa de memoria provisto por grub, las posiciones exactas en la RAM de dichos módulos.

Junto con la carga de módulos y el booteo del BSP, se prepara el entorno para el booteo en etapas de los AP.

- **Niveles de booteo del BSP y preparación del entorno de inicio de los AP:**

  1. Grub inicializa la máquina a un estado conocido y brinda el control al loader de nivel 1 del BSP brindándole estructuras con metadatos de grub.
     a) Se realizan validaciones requeridas por la especificación multiboot, verificación de firmas, etc.
     b) Se obtienen de la metadata provista por grub las posiciones de memoria donde estan cargados los módulos, ellos son:
        1) kernel64.bin64
        2) ap_full_code.bin64
        3) ap_startup_code
     c) Los Application Processors inician en modo real, es por esto que deben comenzar su ejecución por debajo del primer mega de memoria principal. Por este motivo se copia el módulo ap_startup_code a una dirección arbitraria alineada a página 0x2000.
     d) Como los AP comienzan su ejecución por debajo del primer mega, esto hace posible la superposicion de código con grub y otras estructuras del kernel, para evitarlo, minimizamos el tamaño del startup de modo real del ap, haciendo lo antes posible un salto a un loader de nivel 2 por encima del mega, este loader de nivel 2 es ap_full_code.bin64.
     e) Al tener el Application Processor que hacer un salto entre los dos niveles de booteo, se le inyecta al primer módulo la direccion donde esta cargado el segundo nivel de booteo.
     f) Finalmente, se realiza un salto en la ejecución a donde comienza el modulo kernel64.bin64 donde el BSP finaliza la inicializacion del contexto hasta modo largo de 64 bits y enciende a los demás núcleos del sistema.

2.2.1. Booteo e integración con grub: Mapa de memoria: Memoria baja y modulos en memoria alta

TODO: INSERTAR GRAFICO DE MEMORIA A LO TP3 DE ORGA2.
TODO: INSERTAR GRAFICO DE TODOS LOS MODULOS, A LO UML, INDICANDO LAS VARIABLES GLOBALES QUE COMPARTEN, IE. EL 0xABBAABBA PARA HACER EL JUMP DE LOS AP Y EL JMP AL BINARIO DE 64 BITS DESDE EL LOADER DE GRUB

2.3. Inicialización del Bootstrap processor: Pasaje entre modo protegido a modo legacy x64

2.3.1. Modo Legacy x64: GDT, Paginación de los primeros 4gb

La especificación multiboot nos asegura que estamos en modo protegido, pero no tenemos la certeza de tener una GDT válida, es por esto que asignamos una GDT con 3 entradas todas de nivel 0, una comun a 32 y 64 bits de datos y 2 de código, esta diferenciación de descriptores de código es necesaria para realizar los jump far para pasar de modo real hacia modo protegido y de modo protegido-compatibilidad x64 hacia modo largo x64.

TODO: IMAGEN DE LA GDT

Se utilizó un modelo de paginación en identity mapping donde se cubren los primeros 64 GB de memoria. El modo de paginación elegido fue IA32e en 3 niveles con páginas de 2 megas, es importante remarcar que como para pasar a modo largo de 64 bits es obligatorio tener paginación activa, el mapeo de la memoria virtual fue realizado en 2 etapas, en la primera se mapearon unicamente los primeros 4gb pues desde modo protegido puedo direccionar como máximo hasta 4gb y luego desde modo largo, se completa el esquema de paginacion a 64gb agregando las entradas necesarias a las estructuras.

Esquema de paginación IA32-e:

- **PML4:** 512 entries disponibles de 8 bytes de ancho cada una. Solo fue necesario instanciar la primera entrada de la tabla.
- **PDPT:** 512 entries disponibles de 8 bytes de ancho cada una. Solo fue necesario instanciar las primeras 64 entradas de esta tabla.
- **PDT:** 32768 entries disponibles de 8 bytes de ancho cada una. Se instancian en modo protegido 2048 entradas para cubrir los primeros 4gb y luego desde modo largo se completan las 30720 entradas restantes completando 64 gb.

TODO: IMAGEN DE LOS NIVELES DE PAGINACION CON SUS FLECHITAS MOSTRANDO EN ROJO QUE POR ENCIMA DE 4GB NO SE PUEDE MAPEAR

TODO: TABLAS CON DIRECCIONES DE LAS ESTRUCTURAS Y CANTIDAD DE ENTRADAS DE CADA UNA.

Luego de establecer estas estructuras, realizamos una comprobación de disponibilidad de modo x64, y encendemos los bits del procesador para habilitar dicho modo.

2.4. Inicialización del Bootstrap processor: Pasaje a modo largo x64 nativo

Para pasar de modo compatibilidad a modo nativo de 64 bits, es necesario realizar un salto largo en la ejecucion a un descriptor de la GDT de código de 64 bits.

Luego de realizar el salto al segmento de código de x64 de la GDT establecemos un contexto seguro con los registros en cero, seteamos los selectores correspondientes de la GDT y establecemos los punteros a una pila asignada al BSP.

2.4.1. Modo Largo x64: Extensión de paginación a 64 gb

En este punto ya podemos direccionar arriba de los 4gb, entonces completamos las entradas en las estructuras de paginación para completar el mapeo hasta 64gb.

TODO: IMAGEN DEL MAPEO COMPLETO EN 3 NIVELES

2.4.2. Modo Largo x64: Inicialización del PIC - Captura de excepciones e interrupciones

Enviamos señales al PIC para programarlo de forma que atienda las interrupciones enmascarables y asignamos una IDT que captura todas las excepciones e interrupciones y de ser necesario, realiza las acciones correspondientes con su ISR asociada. Particularmente las excepciones son capturadas y mostradas en pantalla y se utiliza la interrupcion de reloj para sincronizacion y esperas, las demás interrupciones son ignoradas.

TODO: IMAGEN DE LA IDT Y FLECHITAS A LAS ISR

2.4.3. Modo Largo x64: Mapa de memoria del kernel

TODO: IMAGEN DEL MAPA DE MEMORIA DEL KERNEL.

2.5. Multicore: encendido de los AP's

2.5.1. TODO: Ingrese cosas de inicializacion multicore aqui.

Nota: Dado que se realizan experimentos con un máximo de 2 cores, se limita el encendido de los Application Processors a uno, que junto con el Bootstrap Processor son 2 núcleos.

2.6. Multicore: inicialización de modo real a modo nativo x64 de los AP's

Como vimos en la sección anterior, los Application Processors comienzan su ejecucion en una posicion por debajo del primer mega en modo real, nosotros necesitamos hacer saltar la ejecucion a una posicion conocida por encima del mega que no se solape con estructuras del kernel ni otros modulos, la solucion que proponemos es un booteo por etapas.

2.6.1. Booteo por niveles: Modo real a modo protegido y modo protegido en memoria alta

En este primer nivel el núcleo se encuentra en modo real, se inicializa una GDT básica en el mismo código y se salta a modo protegido, esto es necesario para poder direccionar posiciones de memoria por encima del primer mega.

Recordando secciones anteriores, cuando el BSP iniciaba el primer nivel de booteo preparaba el contexto de los APS para inicializar en niveles, en este proceso se inyecta al primer nivel de booteo del AP la posicion de memoria donde esta el segundo nivel de booteo por encima del mega.

Luego resta unicamente realizar el salto al segundo bootloader con un jump para continuar el booteo del AP, de manera similar al BSP pasamos luego a modo nativo de 64 bits.
Notemos que por ejemplo la línea A20 ya esta habilitada, y algunas estructuras del kernel que fueron inicializadas por el BSP son comúnes a todos los núcleos, como por ejemplo la GDT, IDT, la estructura jerárquica de paginación, etc.
Entonces directamente inicializamos los registros del núcleo correspondiente con punteros a dichas estructuras.

Como el número de application processors puede ser variable a priori, cuando un núcleo comienza su ejecucion, es obtenido su código de identificacion dentro del procesador y luego se obtiene una posicion de memoria única para cada núcleo con el fin de poder ubicar los punteros de la pila RSP y RBP, de ser necesaria la reserva de memoria para alguna estructura única por núcleo se puede utilizar este recurso.

3. Desarrollo: Algoritmos implementados

3.1. Sorting de arreglos

3.1.1. Conjuntos de numeros pseudoaleatorios utilizados para los experimentos

3.1.2. Implementación con un unico core

3.1.3. Implementación con dos cores: Paralelización del algoritmo

3.1.4. Implementación con dos cores: Sincronización con espera activa

3.1.5. Implementación con dos cores: Sincronización con inter processor interrupts

3.2. Modificación de elementos de un arreglo

3.2.1. Saturación del canal de memoria

3.3. Fast Fourier Transform

4. Resultados

4.1. Lectura e interpretación de resultados por pantalla

4.2. Resultados: Forma de medición

4.3. Resultados: Plataformas de testing, incluir fsb, ram,cache, etc

4.4. Resultados: Comparación de resultados

5. Conclusión Final