Práctica seL4

MEMORIA

02/07/2020
Daniel Ruskov y Veselin Solenkov
Ingeniería Informática UPV/EHU
Arquitectura y Tecnología de Computadores
Sistemas Operativos, 3er curso

ÍNDICE

INTRODUCCIÓN

El microkernel seL4 es un sistema operativo diseñado para ser seguro y fiable, base para sistemas. Como microkernel, proporciona una pequeña cantidad de servicios para aplicaciones, como abstracciones para crear y administrar espacios de direcciones virtuales, hilos e intercomunicación de procesos (IPC).

En esta práctica hemos utilizado este microkernel para implementar funciones que administran el espacio de memoria asignado al Root task, que hereda de seL4 el contexto inicial, es decir, el TCB, el CSpace y el VSpace, de forma que detectamos la región libre más grande y realizar operaciones de reserva y liberación de espacio sobre ella.

Para construir un sistema sel4 es necesario utilizar otro Sistema Operativo, en nuestro caso un sistema GNU/Linux (Ubuntu 18.04) mediante máquina virtual usando VMware.

OBJETIVO

El objetivo principal de este proyecto es crear estructura(s) de memoria, en la que se puedan reservar y liberar secciones de la misma, y de esta forma comprender mejor y consolidar lo aprendido en la parte teórica de la asignatura de Sistemas Operativos.

Dicha gestión debe seguir una alineación especificada en el principio del programa. La siguiente tabla muestra ejemplos de alineación de direcciones por tipo de alineación (según el tamaño del bloque de datos que maneje el sistema de memoria):

	¿La dirección está alineada a frontera de...			
dirección	**byte** (2^0 B, **1 B**, 8 bits)?	**half-word** (2^1 B, **2 B**, 16 bits)?	**word** (2^2 B, **4 B**, 32 bits)?	**double-word** (2^3 B, **8 B**, 64 bits)?
0x00000000 bin[···0000]	Sí	Sí	Sí	Sí
0x00000001 bin[···0001]	Sí	-	-	-
0x00000002 bin[···0010]	Sí	Sí	-	-
0x00000003 bin[···0011]	Sí	-	-	-
0x00000004 bin[···0100]	Sí	Sí	Sí	-
0x00000005 bin[···0101]	Sí	-	-	-
0x00000006 bin[···0110]	Sí	Sí	-	-
0x00000007 bin[···0111]	Sí	-	-	-
0x00000008 bin[···1000]	Sí	Sí	Sí	Sí

COMPONENTES

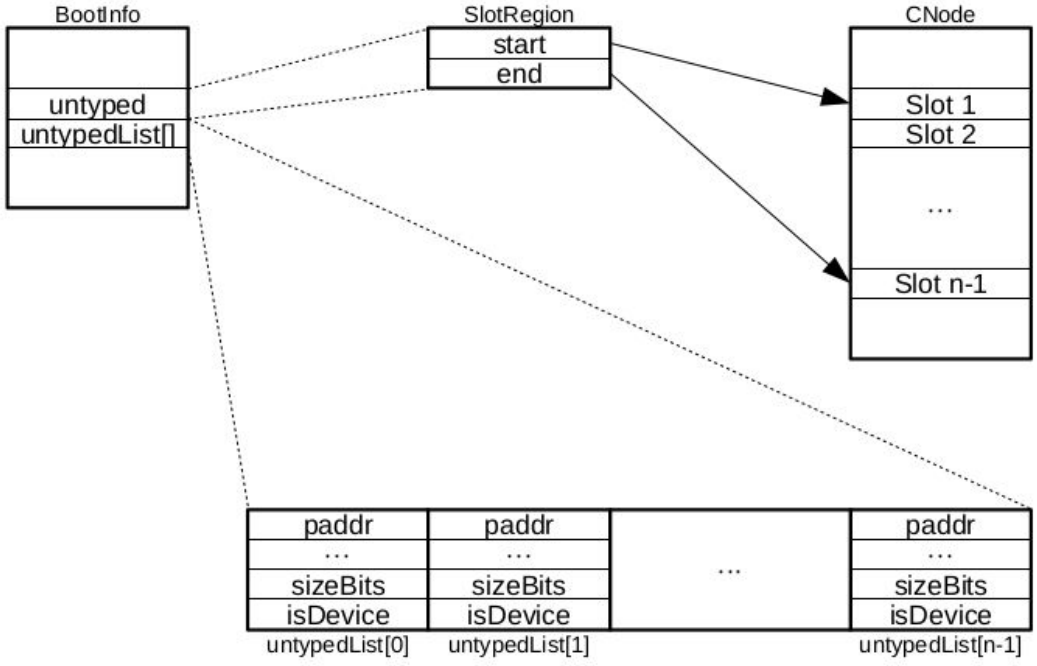
Propios de seL4

CSpace

seL4 utiliza capabilities como control de acces. A cada thread a nivel de usuario se le asigna un capability space (CSpace). Las capabilities de un thread se almacenan en objetos Cnode. El CSpace de la práctica contiene un único Cnode.

BootInfo Frame

Los slots del CNode se rellenan dinámicamente al inicializarse seL4. Para que la tarea principal conozca el contenido de dichos slots, seL4 mapea un BootInfo Frame en su espacio de direccionamiento.



Implementados en la práctica

Slots

Se trata de una lista estática, implementada mediante un array de elementos UntypedDesc y un contador de cuantas posiciones están ocupadas. Sirve como estructura auxiliar para copiar la información de los slots de Boot Info y ordenarlos en la función init_memory_system() (definida como myOrderedSlots).

Slots

seL4_UntypedDesc myUntypedList[]

seL4_Word paddr	seL4_Word paddr		seL4_Word paddr
seL4_Uint8 padding1	seL4_Uint8 padding1		seL4_Uint8 padding1
seL4_Uint8 padding2	seL4_Uint8 padding2	. . .	seL4_Uint8 padding2
seL4_Uint8 sizeBits	seL4_Uint8 sizeBits		seL4_Uint8 sizeBits
seL4_Uint8 isDevice	seL4_Uint8 isDevice		seL4_Uint8 isDevice
0	1		CONFIG_MAX_NUM_BOOTINFO_UNTYPED_CAPS

int countSlots

Region

Es una estructura que contiene la información de una región (varios slots consecutivos) de memoria (libre o reservada).

Region

seL4_Word paddr
unsigned int sizeBitsPow
seL4_Bool isAllocated

Regions

Es otra lista estática de estructuras Region. Se utilizan dos en el código generado (memoryRegions en init_memory_system() que guarda todas las regiones de memoria y maxMemoryRegionAllocates definida de forma global que contiene la región más grande troceada según los allocate o unida por los release realizados).

Regions

Region regions[]

seL4_Word paddr	seL4_Word paddr		seL4_Word paddr
unsigned int sizeBitsPow	unsigned int sizeBitsPow	. . .	unsigned int sizeBitsPow
seL4_Bool isAllocated	seL4_Bool isAllocated		seL4_Bool isAllocated
0	1		MAX_MEMORY_REGIONS

int countRegions

IMPLEMENTACIÓN

Para realizar la práctica implementamos las siguientes tres funciones esenciales:

```
/**
* Inicializa la memoria asignada a Root Task
* ordenando untypedList[] y a partir de ella
* identificando las regiones de memoria libres
* @aligment alineación de memoria 8, 16, 32 o 64
* @return 0 en finalización correcta, !0 e.o.c
*/
int init_memory_system(seL4_Uint8 aligment) {...}
```

```
/**
* Reserva la primera región de memoria alineada de tamaño 2^sizeBits
* Politica first fit
* @sizeBits tamaño de memoria a reservar
* @return puntero a la región de memoria reservada, 0 e.o.c con msg de error
*/
seL4_Word allocate(seL4_Uint8 sizeBits) {...}
```

```
/**
* Libera la región de memoria apuntada por paddr
* @paddr puntero al inicio de la región de memoria a liberar
* @return 0 en ejecución correcta, codigo de error e.o.c
*/
int release(seL4_Word paddr) {
```

Además, utilizamos funciones auxiliares para imprimir el contenido de BootInfo, ordenar los slots de memoria que nos proporciona UntypedList[] mediante el método de quicksort y ver si dos slots son consecutivos para identificar las regiones de memoria.

```
static void print_bootinfo(const seL4_BootInfo *info) {...}

void swap_UntypedDesc(seL4_UntypedDesc *a, seL4_UntypedDesc *b) {...}

void quickSort_UntypedDesc(seL4_UntypedDesc *arr, int low, int high) {...}

seL4_Uint8 are_consecutive(seL4_Word region1, seL4_Word region2, seL4_Uint8
sizeBits) {...}
```

Por último, el programa principal, desde el cual hacemos llamadas a las funciones anteriores para inicializar el sistema y hacer los casos de prueba correspondientes al apartado siguiente.

```
int main(void) {...}
```

Todo el código se recoge en un único fichero llamado main.c, enlazado al final de este documento.

CASOS DE PRUEBA

Para los casos de prueba hemos inicializado el sistema con una alineación de 64, ya que es la más “exigente” a la hora de reservar espacio de memoria.

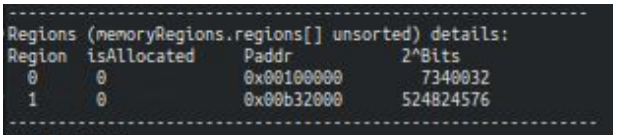
Una vez ejecutado el programa, mostrará un mensaje de saludo y en primer lugar imprimirá la información contenida de BootInfo con todos los slots de memoria proporcionados al Root task de la siguiente manera:

```
>>>
>>> Welcome to the SESO operating system
>>>
============================================================
Info Page:              0x522000
IPC Buffer:             0x521000
Node ID:                0 (of 1)
IOPT levels:            -1
Init cnode size bits:   12
------------------------------------------------------------
Cap details:
Type                    Start           End
Empty                   0x0000018c      0x00001000
Shared frames           0x00000000      0x00000000
User image frames       0x00000010      0x00000131
User image PTs          0x0000000c      0x0000000f
Untypeds                0x00000131      0x0000018c
------------------------------------------------------------
Untyped (info->untypedList[] unsorted) details:
Untyped Slot            Paddr           Bits    Device
  0     0x00000131      0x00100000      20      0
  1     0x00000132      0x00200000      21      0
  2     0x00000133      0x00400000      22      0
  3     0x00000134      0x00b32000      13      0
  4     0x00000135      0x00b34000      14      0
  5     0x00000136      0x00b38000      15      0
  6     0x00000137      0x00b40000      18      0
  7     0x00000138      0x00b80000      19      0
  8     0x00000139      0x00c00000      22      0
  9     0x0000013a      0x01000000      24      0
 10     0x0000013b      0x02000000      25      0
 11     0x0000013c      0x04000000      26      0
 12     0x0000013d      0x08000000      27      0
 13     0x0000013e      0x10000000      27      0
 14     0x0000013f      0x18000000      26      0
 15     0x00000140      0x1c000000      25      0
 16     0x00000141      0x1e000000      24      0
 17     0x00000142      0x1f000000      23      0
 18     0x00000143      0x1f800000      22      0
 19     0x00000144      0x1fc00000      21      0
 20     0x00000145      0x1fe00000      20      0
 21     0x00000146      0x1ff00000      19      0
 22     0x00000147      0x1ff80000      17      0
 23     0x00000148      0x1ffa0000      16      0
 24     0x00000149      0x1ffb0000      14      0
 25     0x0000014a      0x1ffb4000      12      0
------------------------------------------------------------
```

A continuación se ejecutará la función init_memory_system(), la cual hace una copia de la lista UntypedList[] de BootInfo a myUntypedList[] de myOrderedSlots y la ordena de menor a mayor por las direcciones de memoria. Imprime el resultado como muestra a continuación:

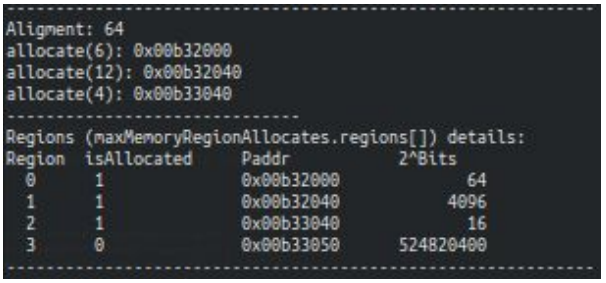
```
------------------------------------------------------------
Untyped (myOrderedSlots.myUntypedList[] by paddr) details:
Untyped Paddr           Bits    Device  2^Bits
  0     0x00100000      20      0         1048576
  1     0x00200000      21      0         2097152
  2     0x00400000      22      0         4194304
  3     0x00b32000      13      0            8192
  4     0x00b34000      14      0           16384
  5     0x00b38000      15      0           32768
  6     0x00b40000      18      0          262144
  7     0x00b80000      19      0          524288
  8     0x00c00000      22      0         4194304
  9     0x01000000      24      0        16777216
 10     0x02000000      25      0        33554432
 11     0x04000000      26      0        67108864
 12     0x08000000      27      0       134217728
 13     0x10000000      27      0       134217728
 14     0x18000000      26      0        67108864
 15     0x1c000000      25      0        33554432
 16     0x1e000000      24      0        16777216
 17     0x1f000000      23      0         8388608
 18     0x1f800000      22      0         4194304
 19     0x1fc00000      21      0         2097152
 20     0x1fe00000      20      0         1048576
 21     0x1ff00000      19      0          524288
 22     0x1ff80000      17      0          131072
 23     0x1ffa0000      16      0           65536
 24     0x1ffb0000      14      0           16384
 25     0x1ffb4000      12      0            4096
------------------------------------------------------------
```

El siguiente paso de init_memory_system() es detectar todas las regiones consecutivas de la lista anterior obtenida, guardando la información de cada una de ellas en regions[] de memoryRegions y la mayor de todas las regiones la copia a regions[] de maxMemoryRegionAllocates para su futura administración de reservar y liberar espacio de memoria. Aquí ell resultado de la lista regions[] de memoryRegions:

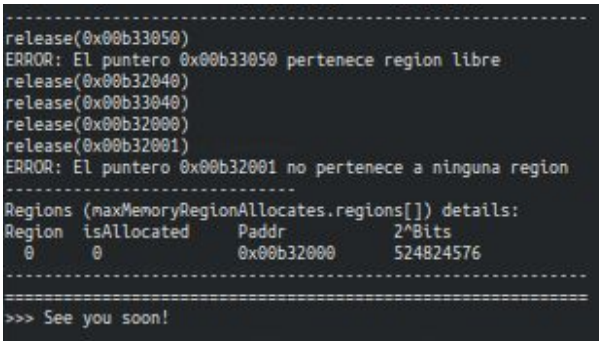


Una vez inicializado el sistema de memoria e identificada la región más grande, sigue probar las funciones allocate() y release().

La reserva de memoria es mediante la política de First Fit, que busca el primer hueco con espacio suficiente para que el trozo de memoria sea alineado y devuelve el puntero hacia esa región. En caso de no poder realizar la operación por falta de memoria, muestra mensaje de error y devuelve 0. Probamos a reservar tres espacios de memoria e imprimimos el resultado de regions[] de maxMemoryRegionAllocates:



El último paso es liberar el espacio de memoria reservado anteriormente mediante la función release(). Esta busca la región apuntada por la dirección de memoria pasada por parámetro y libera la región. En caso de que la dirección recibida apunte a una región ya libre o no apunte a ninguna región, muestra mensaje de error y devuelve el código de error correspondiente. A continuación su demostración:



Se ve como se restaura la región de memoria inicial tras los release. Finalmente se muestra mensaje de despedida y se queda a la espera de terminar el proceso mediante Ctrl+a, x.

INCIDENCIAS

Durante el desarrollo de la práctica hemos sufrido dos incidencias que cabe mencionar:

- En la implementación de init_memory_system(), al hacer la copia de UntypedList[] de BootInfo a myUntypedList[] de Slots no inicializamos countSlots a 0 y daba problemas en la ejecución. Por lo visto el valor inicial por defecto era diferente a cero.
- En una de las asignaciones de init_memory_system() mostrada a continuación, aun correspondiendo los tipos (seL4_Uint8), no se copiaba correctamente el resultado en el campo destino, simplemente se copiaba el valor cero. Esto se soluciono tras múltiples pruebas de casting y cambios de tipos, únicamente cambiando el tipo del campo sizeBitsPow de struct Region a unsigned int. Se desconoce la razón, de que aun coincidiendo los tipos de origen y destino diese tal fallo.

```
memoryRegions.regions[memoryRegions.countRegions].sizeBitsPow =
2<<(myOrderedSlots.myUntypedList[i].sizeBits-1);
```

MEJORAS

A continuación una lista de las posibles mejoras a implementar para optimizar el tiempo de ejecución y/o la memoria utilizada del programa:

- Crear una estructura de tamaño menor y usarla para struct Slots ahorrando espacio de memoria, en vez del tipo seL4_UntypedDesc. Esto se refiere a los campos padding1 y padding2 de seL4_UntypedDesc que no utilizamos para nada. Incluso es posible omitir el campo isDevice según la implementación.
- Usar listas dinamicas para ahorrar espacio de memoria en vez de arrays para listas estáticas en lo que refiere a struct Slots y struct Regions.
- Disminuir la E/S bajando el tiempo de ejecución. Esto se puede conseguir mediante un buffer, el cual se imprime con más información acumulada en vez de múltiples prints para ello.
- Posible (pero no comprobado) evitar hacer copia de UntypedList[] de BootInfo, de manera que el sort por paddr se haga sobre la propia lista UntypedList[] y a partir de ella se detecten las regiones consecutivas.
- Separar el código en más funciones y que cada función tenga menos código y tareas concretas (modularidad). Esto facilita la lectura del código y reutilización, pero aumenta las latencias de salto.

ARCHIVOS RELACIONADOS

- Enunciado completo de la práctica:
  https://drive.google.com/file/d/1uHE9valqm2zjuWKHleqIahI-m-JT9RSP/view?usp=sharing

- Manual seL4:
  https://drive.google.com/file/d/1XvTzbeyCDvsPxT7ddbUNgjCrlwqN-Px7/view?usp=sharing

- Fichero main.c:
  https://drive.google.com/file/d/1ncTkP0Wuw0r5iLKhvNvxkiGQemSNRYJW/view?usp=sharing