Enunciado de la práctica

Esta práctica tiene dos partes. En la primera, deberéis entregar el diseño que habéis utilizado para detectar regiones consecutivas, y en la segunda, el sistema de memoria que habéis diseñado para gestionar la región más grande según el enunciado de la **Sesión 4**. El plazo para entregar la práctica termina el próximo **7 de enero**.

Regiones consecutivas

En esta sección deberéis describir el algoritmo para detectar regiones consecutivas. Por un lado, se debe describir las estructuras de datos y algoritmos utilizados. En esta parte no es necesario incluir todo el código desarrollado, si no la parte correspondiente. Para describir bien la solución podéis utilizar imágenes y ejemplos.

Sistema de gestion de memoria

Una vez que se ha identificado la región más grande, tendréis que explicar el sistema de asignación de memoria que habéis diseñado. Al igual que en el apartado anterior, hay que describir las estructuras de datos y los algoritmos utilizados; y tendréis que subir el código implementado a vuestra cuenta de **GitLab**. Tened en cuenta que el código tiene que estar comentado. En el siguiente ejemplo se utilizan comentario estilo Doxygen (www.doxygen.nl).

```
/*
* Esta función detecta dos regiones consecutivas.
* @param Dirección de comienzo de cada región.
* @return true si son consecutivas. Si no, false.
**/

seL4_Uint8 are_consecutive(seL4_Word region1, seL4_Word region2)
```

En esta parte tendréis que explicar cómo habéis diseñado las tres funciones que os propusimos en la Sesión 4[1]:

- `int init_memory_system(seL4_Uint8 aligment);`

  Inicializa el sistema de memoria según el tipo de alineamiento. El parámetro **`aligment`** podrá tomas los valores **`8`**, **`16`**, **`32`** ó **`64`**.
- `seL4_Word allocate(seL4_Uint8 sizeBits);`

  La función **`allocate()`** asignará una región de memoria del tamaño $2^{sizeBits}$ bytes **alineada**. Si la región solicitada no está disponible, la función imprimirá un mensaje de error y cancelará la ejecución.
- `int release(seL4_Word paddr);`

  La función **`release()`** liberará la región de la dirección y del tamaño indicados.

Formato de la documentacion

La estructura de la documentación es libre pero debéis entregarlo en formato **`pdf`**; si utilizáis imágenes o gráficos, aseguraos de que sean de alta calidad (p.e. `svg` o `eps`).

1 La función **`init_memory_system`**( ) está más especificada y la función **`release()`** está corregida con un parámetro menos.

Sesión 1. Entorno de compilación de seL4

seL4 es un microkernel por lo que no dispone de un entorno para que los usuarios lo puedan manejar. El microkernel **seL4** no ofrece más que una interfaz para comunicarse con el hardware los microkernel y para construir un sistema **sel4** tendremos que utilizar otro SO. En esta práctica utilizaremos un sistema GNU/Linux, más concretamente la distribución Ubuntu 18.04.

En este primer laboratorio instalaremos el código de **seL4** y las librerías necesarias para construir un sistema **seL4**. Además de esto, se tienen que instalar otras herramientas pero como son bastantes, utilizaremos una máquina virtual donde ya está instalado todo lo que necesitamos. Puedes encontrar más información acerca de dichas herramientas en la siguiente dirección:

https://docs.sel4.systems/HostDependencies

La máquina virtual

Primero, tenemos que bajar la máquina virtual. Tenemos dos opciones. Por una parte, si vamos a utilizar nuestro propio ordenador, descargaremos la imagen **VirtualBox**. Si, por el contrario, vamos a utilizar los ordenadores de la Facultad, bajaremos la imagen de **VMWare** (en este caso recomendamos trabajar sobre Windows). Las dos imágenes están en el repositorio, en el directorio:

- `repositorio/cursos/2019-20/3/SE-SO/`

Ya que utilizar directamente una máquina virtual no es muy cómodo, la máquina virtual está configurada con un servidor **ssh**.

En **VirtualBox**, **ssh** está configurado en el puerto **8000** y en la direccion IP **127.0.0.1**.

- `ssh -p 8000 seso@127.0.0.1` (username: **seso** | password: **oses**)

En **VMWare ssh** está en el puerto **22**; sin embargo, en este caso, debemos conseguir un dirección IP utilizando el comando **sudo dhclient** (y luego **ip a**).

- `ssh seso@dirección_IP` (username: **seso** | password: **oses**)

Estructura de directorios

Para construir un proyecto **seL4** se necesita la estructura de directorios siguiente. Excepto el directorio **seso**, el sitio web de **seL4** proporciona todas las librerías necesarias:

```
SESO/
├── kernel/
├── tools/
│   └── cmake-tool/
├── build/
└── projects/
    ├── musllibc/
```

```
├── utils_libs/
├── seL4_libs/
└── seso/
```

Todo el código que desarrollemos deberemos ponerlo en el directorio **`seso`** que uniremos con el kernel y las librerías de **`seL4`** para construir el sistema **seL4/SESO**. La herramienta que se utiliza para compilar los kernel **`seL4`** es **`CMake`** y no la habitual **`make`**.

CMake

El software **`CMake`** es una herramienta para compilar y testear. Para configurar la compilación se utilizan los ficheros **`CMakeLists.txt`** donde se ponen los comandos. Los comandos que utilizaremos principalmente serán los siguientes:

- **`cmake_minimum_required(VERSION v)`**: Mínima versión requerida de **`CMake`**.
- **`project(nombre_proyecto)`**: Se especifica el nombre del proyecto.
- **`set(variable valor)`**: Establece el valor **`valor`** en la variable **`variable`**.
- **`add_executable(nombre_proyecto fichero)`**: Añade al proyecto **`nombre_proyecto`** un **`fichero`** que se tiene que compilar.
- **`target_link_libraries(nombre_proyecto librerías)`**: Las **`librerías`** que se utilizan en el proyecto **`nombre_proyecto`**.
- **`include(fichero)`**. Carga y ejecuta código **`CMake`** desde un **`fichero`** dado.

Una vez configurada la compilación, se tiene que utilizar la herramienta **`ninja`** para construir los ejecutables (o binarios). Puedes conseguir más información en las siguientes direcciones:

- https://www.cmake.org
- https://www.ninja-build.org

Instalación de seL4

El código del microkernel **`seL4`** y el de las librerías que necesitaremos está en la siguiente dirección de gihub:

- https://github.com/seL4/

Tenemos que crear la estructura de directorios vista anteriormente utilizando la herramienta **`git`**. Después de crear el directorio de mayor nivel, tendremos que bajar **`seL4`**:

- `mkdir SESO`
- `cd SESO`

El kernel seL4

- `git clone --branch 10.1.1 https://github.com/seL4/seL4.git kernel`
- `git clone --branch 10.1.x-compatible https://github.com/seL4/seL4_tools.git tools`

Librerías seL4

- `mkdir projects`
- `git clone --branch 10.1.x-compatible https://github.com/seL4/seL4_libs.git projects/sel4_libs`
- `git clone --branch 10.1.x-compatible https://github.com/seL4/musllibc.git projects/musllibc`
- `git clone --branch 10.1.x-compatible https://github.com/seL4/util_libs.git projects/util_libs`

seL4 proporciona un *script* (`init-build.sh`) para configurar automáticamente el proceso de compilación. Utilizaremos el siguiente comando para crear en el directorio de raíz (**`~/SESO/`**) un enlace con el *script*:

- `ln -s tools/cmake-tool/init-build.sh init-build.sh`

A continuación tenemos que crear el **`CMakeLists.txt`** para compilar **seL4**. Para ello, crearemos en el directorio de raíz un fichero llamado **`CMakeLists.txt`** con las sentencias incluidas en la imagen siguiente:

```
cmake_minimum_required(VERSION 3.7.2)

if (${PLATFORM} IN_LIST KernelX86Sel4Arch_all_strings)
        set(KernelArch x86 CACHE STRING "" FORCE)
        set(KernelX86Sel4Arch ${PLATFORM} CACHE STRING "" FORCE)
endif()

include(tools/cmake-tool/default-CMakeLists.txt)

if(SIMULATION)
        ApplyCommonSimulationSettings("x86")
else()
        if(KernelArchX86)
                set(KernelIOMMU ON CACHE BOOL "" FORCE)
        endif()
endif()

# We must build the debug kernel because the tutorials rely on
# seL4_DebugPutChar
# and they don't initialize a platsupport driver.

ApplyCommonReleaseVerificationSettings(FALSE FALSE)

GenerateSimulateScript()
```

CMakeLists.txt

Ahora probaremos a compilar **seL4**.

- `mkdir build`
- `cd build`
- `../init-build.sh  -DPLATFORM=x86_64 -DSIMULATION=TRUE`

Pero la compilación fallará porque no hemos especificado cuál es la **`Root_task`**.

Crear la Root_task

Como hemos visto, un microkernel no ofrece nada más que los **mecanismos** para gestionar los recursos. Por lo tanto, le tenemos que indicar cuál es la tarea principal. En el microkernel **seL4** esta tarea se llama **Root_task**.

En los siguientes laboratorios utilizaremos la **Root_task** para implementar el sistema operativo **SESO** pero, por ahora, crearemos un proyecto nuevo y comprobaremos que todo lo hemos instalado correctamente.

Entonces, vamos a crear los directorios del nuevo proyecto y así como una **Root_task** muy sencilla (**main.c**) en el directorio **src**:

- `mkdir projects/seso`
- `mkdir projects/seso/src`

```
#include <stdio.h>

int main(void)
{
    printf("Bienvenido al sistema operativo SESO\n");
    return 0;
}
```

main.c

Además, para compilar el proyecto **SESO** tenemos que crear un fichero **CMakeLists.txt** en el directorio **~/SESO/projects/seso**.

```
cmake_minimum_required(VERSION 3.7.2)

project(SESO C) # create a new C project called 'Hello'

# add files to our project. Paths are relative to this file.
add_executable(SESO src/main.c)

# we need to link against the standard C lib for printf
target_link_libraries(SESO sel4muslcsys  muslc)

# Set this image as the rootserver
DeclareRootserver(SESO)
```

`CMakeLists.txt` del proyecto

Ejecución del sistema (simulación)

Como hemos visto, la compilación de nuestro sistema se realiza sobre el directorio **build**. Ahora, para compilar nuestro proyecto, tenemos que borrar todo lo que hay en el directorio **build**:

- `rm -r *`

Tenemos que ejecutar de nuevo el *script* **`init-build.sh`**:

- `../init-build.sh  -DPLATFORM=x86_64 -DSIMULATION=TRUE`

Y tenemos que rehacer el proceso **`build`**:

- `ninja`

Si no hay errores, ejecutaremos la simulación del sistema **`seL4`**:

- `./simulate`

Para salir de la simulación se tiene que pulsar **`<ctrl>+<a>`** y luego **`<x>`**.

Sesión 2. Inicialización de seL4 (Cspaces, BootInfo Frame)

Como hemos visto en el laboratorio anterior, **seL4** no ofrece más que una interfaz. Además de esto, también vimos que tenemos que especificar cuál es la **`Root_task`** para que se ejecute el microkernel **seL4**

La tarea principal hereda de **seL4** el contexto inicial, es decir, el **TCB**, el **CSpace** y el **VSpace**. Ese contexto está compuesto por *frames* donde se encuentran el código de la tarea y los datos, así como el buffer **IPC**.

CSpace inicial

seL4 utiliza *capabilities* como control de acceso. A cada *thread* a nivel de usuario se le asigna un *capability space* (**CSpace**), a través del cual **seL4** controla a qué recursos puede acceder. Las *capabilities* de un *thread* se almacenan en objetos **CNode**, y cada **Cnode** tiene varios *slots* donde hay o puede haber una *capability*.

El **Cspace** de la tarea principal contiene un único **Cnode**. La siguiente tabla muestra los 12 *slots* que tiene el **Cnode** inicial.

Enum Constant	**Capability**
`seL4_CapNull`	null
`seL4_CapInitThreadTCB`	initial thread's TCB
`seL4_CapInitThreadCNode`	initial thread's CNode
`seL4_CapInitThreadVSpace`	initial thread's VSpace
`seL4_CapIRQControl`	global IRQ controller
`seL4_CapASIDControl`	global ASID controller
`seL4_CapInitThreadASIDPool`	initial thread's ASID pool
`seL4_CapIOPort`	global I/O port cap
`seL4_CapIOSpace`	global I/O space cap
`seL4_CapBootInfoFrame`	BootInfo frame
`seL4_CapInitThreadIPCBuffer`	initial thread's IPC buffer
`seL4_CapDomain`	domain cap

Table 1: Contenido del **Cnode** del *thread* inicial.

BootInfo Frame

Los *slots* del **CNode** se rellenan dinámicamente al inicializarse **seL4**. El contenido de cada *slot* depende de diversas condiciones como pueden ser la arquitectura del sistema o las características del `Root_task`. Para que la tarea principal conozca el contenido de dichos slots, **seL4** mapea un **BootInfo Frame** en su espacio de direccionamiento. **seL4** establece cuál es la dirección de la estructura de datos y se la pasa a la `Root_task` a través de un registro de la CPU.

BootInfo Frame es una estructura de C cuyos campos se indican en la tabla siguiente.

Field Type	Field Name	Description
`seL4_Word`	`extraLen`	length of additional bootinfo information in bytes
`seL4_Word`	`nodeID`	node ID
`seL4_Word`	`numNodes`	number of nodes
`seL4_Word`	`numIOPTLevels`	number of I/O page-table levels (-1 if CONFIG_IOMMU unset)
`seL4_IPCBuffer*`	`ipcBuffer`	pointer to the initial thread's IPC buffer
`seL4_SlotRegion`	`empty`	empty slots (null caps)
`seL4_SlotRegion`	`sharedFrames`	reserved
`seL4_SlotRegion`	`userImageFrames`	frames containing the userland image
`seL4_SlotRegion`	`userImagePaging`	userland-image paging structure caps
`seL4_SlotRegion`	`ioSpaceCaps`	I/O space capabilities for ARM SMMU
`seL4_SlotRegion`	`extraBIPages`	frames backing additional bootinfo information
`seL4_UntypedDesc[]`	`untypedList`	array of information about each untyped
`seL4_Uint8`	`initThreadCNodeSizeBits`	CNode size (2^n slots)
`seL4_Word`	`initThreadDomain`	domain of the initial thread (see Section 6.3)
`seL4_SlotRegion`	`untyped`	Untyped-memory capabilities

Table 2: BootInfo Frame.

Estructura BootInfo Frame

En el código de tabla siguiente se muestran las estructuras de datos y todos los campos que se utilizan para definir el **BootInfo Frame**. En eGela hay un manual de **seL4** para poder consultar el significado de dichos campos.

```
enum {
    seL4_CapNull                =  0, /* null cap */
    seL4_CapInitThreadTCB       =  1, /* initial thread's TCB cap */
    seL4_CapInitThreadCNode     =  2, /* initial thread's root CNode cap */
    seL4_CapInitThreadVSpace    =  3, /* initial thread's VSpace cap */
    seL4_CapIRQControl          =  4, /* global IRQ controller cap */
    seL4_CapASIDControl         =  5, /* global ASID controller cap */
    seL4_CapInitThreadASIDPool  =  6, /* initial thread's ASID pool cap */
    seL4_CapIOPortControl       =  7, /* global IO port control cap (null cap if not supported) */
    seL4_CapIOSpace             =  8, /* global IO space cap (null cap if no IOMMU support) */
    seL4_CapBootInfoFrame       =  9, /* bootinfo frame cap */
    seL4_CapInitThreadIPCBuffer = 10, /* initial thread's IPC buffer frame cap */
    seL4_CapDomain              = 11, /* global domain controller cap */
    seL4_NumInitialCaps         = 12
};

/* Legacy code will have assumptions on the vspace root being a Page Directory
 * type, so for now we define one to the other */
#define seL4_CapInitThreadPD seL4_CapInitThreadVSpace

/* types */
typedef seL4_Word seL4_SlotPos;

typedef struct {
    seL4_SlotPos start; /* first CNode slot position OF region */
    seL4_SlotPos end;   /* first CNode slot position AFTER region */
} seL4_SlotRegion;

typedef struct {
    seL4_Word  paddr;   /* physical address of untyped cap  */
    seL4_Uint8 padding1;
    seL4_Uint8 padding2;
    seL4_Uint8 sizeBits;/* size (2^n) bytes of each untyped */
    seL4_Uint8 isDevice;/* whether the untyped is a device  */
} seL4_UntypedDesc;

typedef struct {
    seL4_Word         extraLen;        /* length of any additional bootinfo information */
    seL4_NodeId       nodeID;          /* ID [0..numNodes-1] of the seL4 node (0 if uniprocessor)
*/
    seL4_Word         numNodes;        /* number of seL4 nodes (1 if uniprocessor) */
    seL4_Word         numIOPTLevels;   /* number of IOMMU PT levels (0 if no IOMMU support) */
    seL4_IPCBuffer*   ipcBuffer;       /* pointer to initial thread's IPC buffer */
    seL4_SlotRegion   empty;           /* empty slots (null caps) */
    seL4_SlotRegion   sharedFrames;    /* shared-frame caps (shared between seL4 nodes) */
    seL4_SlotRegion   userImageFrames; /* userland-image frame caps */
    seL4_SlotRegion   userImagePaging; /* userland-image paging structure caps */
    seL4_SlotRegion   ioSpaceCaps;     /* IOSpace caps for ARM SMMU */
    seL4_SlotRegion   extraBIPages;    /* caps for any pages used to back the additional bootinfo
information */
    seL4_Word         initThreadCNodeSizeBits; /* initial thread's root CNode size (2^n slots) */
    seL4_Domain       initThreadDomain; /* Initial thread's domain ID */
    seL4_SlotRegion   untyped;         /* untyped-object caps (untyped caps) */
    seL4_UntypedDesc  untypedList[CONFIG_MAX_NUM_BOOTINFO_UNTYPED_CAPS]; /* information about each
untyped */
    /* the untypedList should be the last entry in this struct, in order
     * to make this struct easier to represent in other languages */
} seL4_BootInfo;

/* If extraLen > 0 then 4K after the start of bootinfo is a region of extraLen additional
```

```
 * bootinfo structures. Bootinfo structures are arch/platform specific and may or may not
 * exist in any given execution. */
typedef struct {
    /* identifier of the following chunk. IDs are arch/platform specific */
    seL4_Word id;
    /* length of the chunk, including this header */
    seL4_Word len;
} seL4_BootInfoHeader;
```

Table 3: Estructuras de datos de **BootInfo Frame**.

Imprimir los campos de BootInfo Frame

El objetivo de esta segunda sesión práctica es comprender la estructura de datos **BootInfo Frame**. Para ello, tenéis que **desarrollar una función que imprima todos sus campos**. Se necesitan las siguientes includes:

```
#include <stdio.h>
#include <sel4/sel4.h>
#include <sel4platsupport/bootinfo.h>
```

Además, para leer el **Bootinfo Frame**, necesitamos utilizar la variable y la función siguientes que proporciona **seL4**. De esta manera, en la variable **boot_info**, tendremos la información del **BootInfo Frame**.

```
const seL4_BootInfo *boot_info;
boot_info = platsupport_get_bootinfo();
```

Por tanto, desarrolla para ello un código bien organizando, es decir, crea una nueva función -p.e. **print_bootinfo()**- que deberás llamarla desde el **main**. Por el momento, sólo utilizaremos un único fichero, el **main.c**.

Sesión 3. Identificación de regiones de memoria

En la sesión de laboratorio anterior analizamos la estructura de datos **BootInfo Frame** e imprimos sus campos. Ente ellos, volcamos el contenido del array `seL4_UntypedDesc untypedList[]` que contiene información sobre las regiones de memoria asignadas a la **`Root_task`**, conocida como `Untyped Memory`.

Durante el arranque, **seL4** asigna la memoria requerida para el Kernel en sí. Luego crea la **`Root_task`** y le entrega toda la memoria restante a través del referido campo `untypedList[]`. Hay que tener encuentra que esta memoria *untyped* se proporciona **sin un orden específico**.

En la tabla siguiente (Table 1) podemos ver un esquema de dicha estructura.

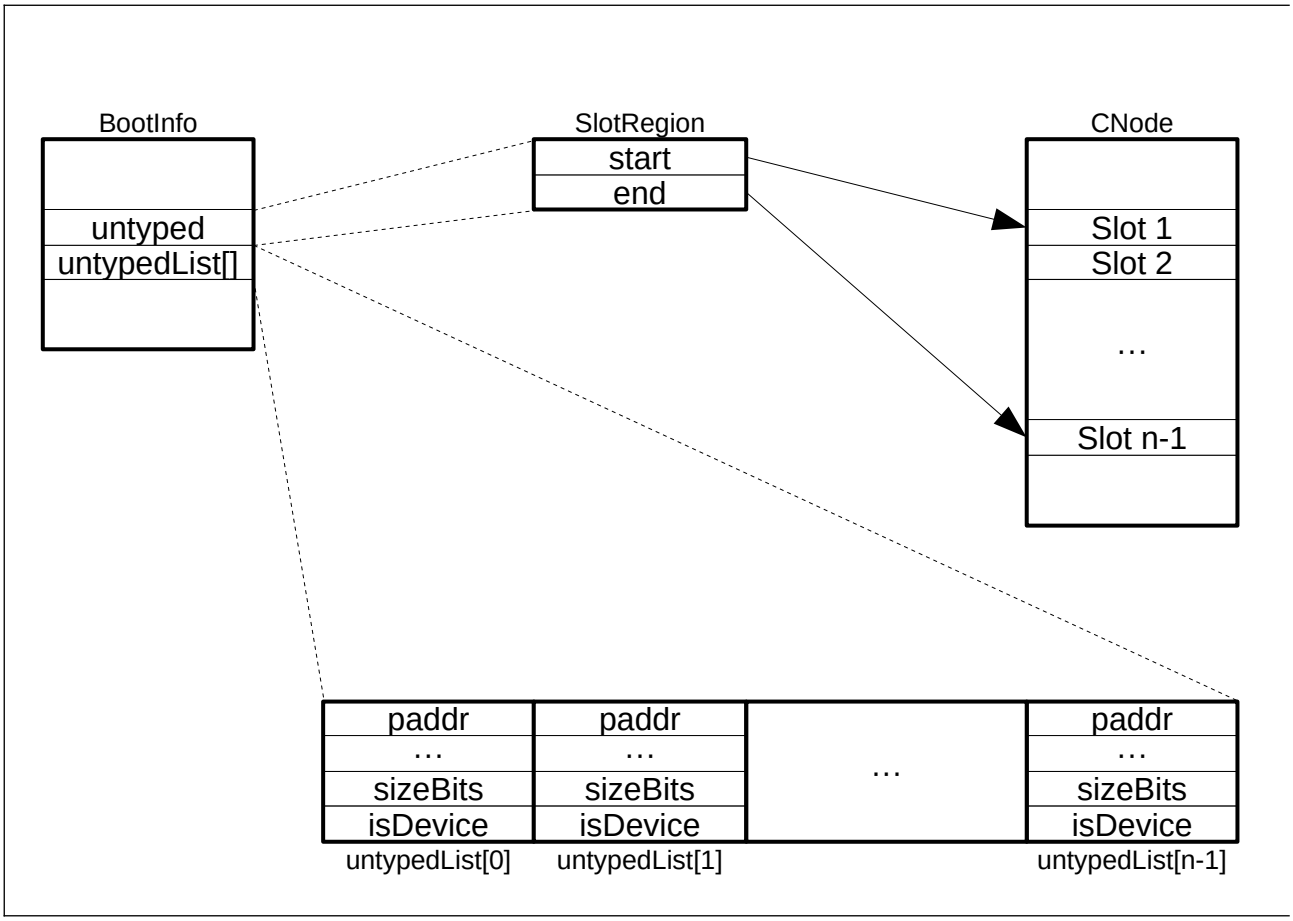


Table 1: Esquema de **BootInfo Frame**.

El campo **`untyped`** del **BootInfo Frame**, es del tipo **`seL4_SlotRegion`** (ver Table 2) se utiliza para calcular dónde están las *capabilities* en la memoria sin tipo (*untyped*). Con los campos **`start`** y **`end`** sabemos dónde están, desde el principio, dentro del **`Cnode`**.

```
typedef struct {
    seL4_SlotPos start; /* first CNode slot position OF region */
    seL4_SlotPos end;   /* first CNode slot position AFTER region */
} seL4_SlotRegion;
```

Table 2: Estructura del campo `untyped` de **BootInfo Frame**.

El campo **`paddr`** de cada entrada del array **`untypedList[]`** (ver Table 3), señala la dirección física de comienzo de cada objero de memoria *untyped*. El campo `sizeBits` indica el tamaño del objero de memoria *untyped* ($2^{\texttt{sizeBits}}$).

```
typedef struct {
    seL4_Word  paddr;   /* physical address of untyped cap  */
    seL4_Uint8 padding1;
    seL4_Uint8 padding2;
    seL4_Uint8 sizeBits;/* size (2^n) bytes of each untyped */
    seL4_Uint8 isDevice;/* whether the untyped is a device  */
} seL4_UntypedDesc;
```

Table 3: Estructura de una entrada de **`untypedList[]`**.

Un ejemplo del volcado de dicha estructura puede ser el siguiente:

```
Untypeds start: 0x0000012e end: 0x00000188
Untyped details:
Untyped    slot      paddr    sizeBits isDevice
   0    0x0000012e 0x00100000    20       0
   1    0x0000012f 0x00200000    21       0
   2    0x00000130 0x00400000    22       0
   3    0x00000131 0x00b2f000    12       0
   4    0x00000132 0x00b30000    16       0
   5    0x00000133 0x00b40000    18       0
   6    0x00000134 0x00b80000    19       0
   7    0x00000135 0x00c00000    22       0
   8    0x00000136 0x01000000    24       0
   9    0x00000137 0x02000000    25       0
  10    0x00000138 0x04000000    26       0
  11    0x00000139 0x08000000    27       0
  12    0x0000013a 0x10000000    27       0
  13    0x0000013b 0x18000000    26       0
  14    0x0000013c 0x1c000000    25       0
  15    0x0000013d 0x1e000000    24       0
  16    0x0000013e 0x1f000000    23       0
  17    0x0000013f 0x1f800000    22       0
  18    0x00000140 0x1fc00000    21       0
  19    0x00000141 0x1fe00000    20       0
  20    0x00000142 0x1ff00000    19       0
```

```
 21    0x00000143 0x1ff80000    17       0
 22    0x00000144 0x1ffa0000    16       0
 23    0x00000145 0x1ffb0000    14       0
 24    0x00000146 0x1ffb4000    12       0
```

Table 4: Ejemplo de volcado de la memoria *untyped*, no reservada para dispositivos (`isDevice == 0`), asignada a la **`Root_task`**.

Propuesta de trabajo

Como se puede ver en la Tabla 4, algunas regiones son consecutivas. Por ejemplo, los tres primeros objetos son consecutivos:

i	paddr	sizeBits	
0	0x00100000	20	(2^{20} = 1 MB)
1	0x00200000	21	(2^{21} = 2 MB)
2	0x00400000	22	(2^{22} = 4 MB)

El objetivo final de la práctica va a ser desarrollar un gestor memoria que asigne memoria a procesos a través de la correspondiente interfaz de llamadas al sistema. De cara a realizar una gestión más eficiente de la memoria nos interesa tener identificadas todo el conjunto de regiones consecutivas de memoria con su tamaño.

Por tanto, os pedimos que desarrolléis el código necesario que consiga los siguientes tres hitos:

1. Identificar si dos regiones son consecutivas.
2. Encontrar la región consecutiva de mayor tamaño, indicando su dirección de comienzo y tamaño.
3. Construir una estructura de datos (p.e. ordenada de mayor a menor) de todas las regiones consecutivas, guardando su dirección de comienzo y su tamaño.

Recuerda que esta memoria se proporciona sin un orden específico.

El trabajo de esta sesión se deberá añadir a la entrega final como un apartado específico que se evaluará.

Ejemplo de solución de la Sesión 2.

```
#include <stdio.h>
#include <sel4/sel4.h>
#include <sel4platsupport/bootinfo.h>

const seL4_BootInfo *boot_info;

static void print_bootinfo(const seL4_BootInfo* info) {
int i;

/* General info */
    printf("Info Page:            %p\n",         info);
    printf("IPC Buffer:           %p\n",         info->ipcBuffer);
    printf("Node ID:              %d (of %d)\n", info->nodeID, info->numNodes);
    printf("IOPT levels:          %d\n",         info->numIOPTLevels);
    printf("Init cnode size bits: %d\n",         info->initThreadCNodeSizeBits);

/* Cap details */
    printf("\nCap details:\n");
    printf("Type                   Start       End\n");
    printf("Empty               0x%08x 0x%08x\n",  info->empty.start,
                                                   info->empty.end);
    printf("Shared frames       0x%08x 0x%08x\n",   info->sharedFrames.start,
                                                   info->sharedFrames.end);
    printf("User image frames   0x%08x 0x%08x\n",   info->userImageFrames.start,
                                                   info->userImageFrames.end);
    printf("User image PTs      0x%08x 0x%08x\n",   info->userImagePaging.start,
                                                   info->userImagePaging.end);
    printf("Untypeds            0x%08x 0x%08x\n",   info->untyped.start,
                                                   info->untyped.end);

/* Untyped details */
    printf("-----------------------------------------------------------\n");
    printf("\nUntyped details:\n");
    printf("Untyped    Slot      Paddr     Bits Device\n");
    for (i = 0; i < info->untyped.end-info->untyped.start; i++) {
      if (!(info->untypedList[i].isDevice))
         printf(" %3d    0x%08x 0x%08x   %2d     %d\n", i,
                                          info->untyped.start + i,
                                          info->untypedList[i].paddr,
                                          info->untypedList[i].sizeBits,
                                          info->untypedList[i].isDevice);
    }
    printf("-----------------------------------------------------------\n");
}

int main(void) {
    printf(">>>\n>>> Bienvenido al sistema operativo SESO\n>>>\n");
    boot_info = platsupport_get_bootinfo();
    print_bootinfo(boot_info);

    return 0;
}
```

Table 5: **`main.c`**.

Sesión 4. Gestión de memoria

En la sesión de laboratorio anterior construimos una estructura de datos con todas las regiones consecutivas, guardando tanto su dirección de comienzo como su tamaño. En esta cuarta sesión tendremos que **elegir la región de mayor tamaño y gestionarla** sobre la base de las dos funciones siguientes:

- `int` **`init_memory_system`**`(seL4_Uint8` **`aligment`**`);`

  Inicializa el sistema de memoria según el tipo de alineación.

- `seL4_Word` **`allocate`**`(seL4_Uint8` **`sizeBits`**`);`

  La función **`allocate()`** asignará una región de memoria del tamaño $2^{sizeBits}$ bytes **alineada**. Si la región solicitada no está disponible, la función imprimirá un mensaje de error y cancelará la ejecución.

- `int` **`release`**`(seL4_Word` **`paddr`**`, seL4_Uint8` **`sizeBits`**`);`

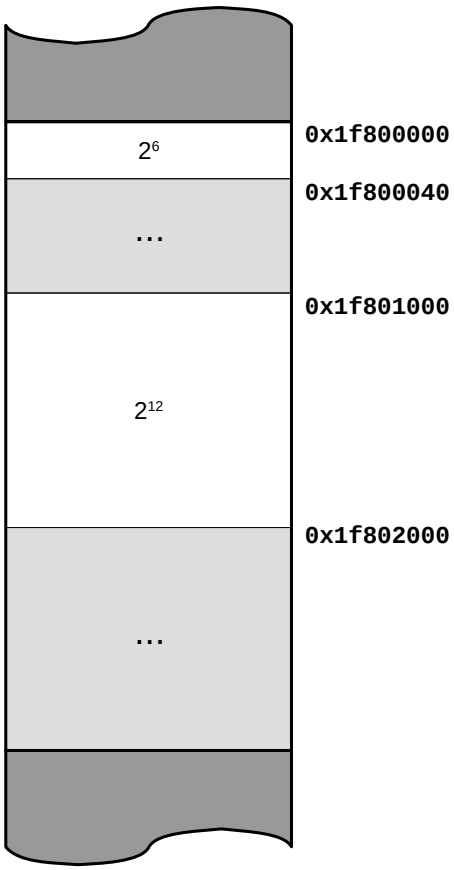
  La función **`release()`** liberará la región de la dirección y del tamaño indicados.

El sistema de memoria direcciona a byte, sin embargo puede manejar bloques de datos de tamaño 2^n B: **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) o **double-word** (2^3 B, 8 B, 64 bits). Estos bloques deben de estar alineados a frontera de 2^n B (ver *Tabla 1*).

Por ejemplo, tenemos un sistema de memoria que maneja bloques de byte. Si la región de mayor tamaño comenzara en la dirección **`0x1f800000`** y consideramos que tenemos capacidad suficiente, si nos solicitan **`allocate(6)`** nos estarían pidiendo 64 B (2^6 B) y deberíamos devolver la propia dirección **`0x1f800000`**, quedando como primera dirección disponible **`0x1f800040`**.

Si a continuación, nos solicitan **`allocate(12)`** nos estarían pidiendo una región de 4 KB (2^{12} B) y deberíamos devolver la dirección **`0x1f801000`**. Como se muestra en la imagen adjunta, quedan dos huecos que comienzan en las direcciones **`0x1f800040`** y **`0x1f802000`** respectivamente.

¿Y si ahora nos solicitan **`allocate(4)`**, 16 B (2^4 B)? ¿Qué dirección deberá devolver la función?



En esta sesión de la práctica debéis desarrollar el código de las tres funciones, **`init_memory_system()`**, **`allocate()`** y **`release()`**, de la forma que consideréis más eficiente.

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

Tabla 1: Alineación de direcciones.

Sesión 5. Configurar Git

Vamos a utilizar **Git** para mantener diferentes versiones del proyecto y para entregar el código y la documentación de la práctica. **Git** es un sistema colaborativo de control de versiones desarrollado inicialmente para **Linux** (por el propio Linus Torvalds). Concretamente vamos a utilizar el servicio proporcionado por **GitLab** (https://gitlab.com) como repositorio remoto donde deberá estar la máquina virtual **seso**.

En la Figure 1 podemos ver la estructura de **Git** y un resumen de los comandos más utilizados.

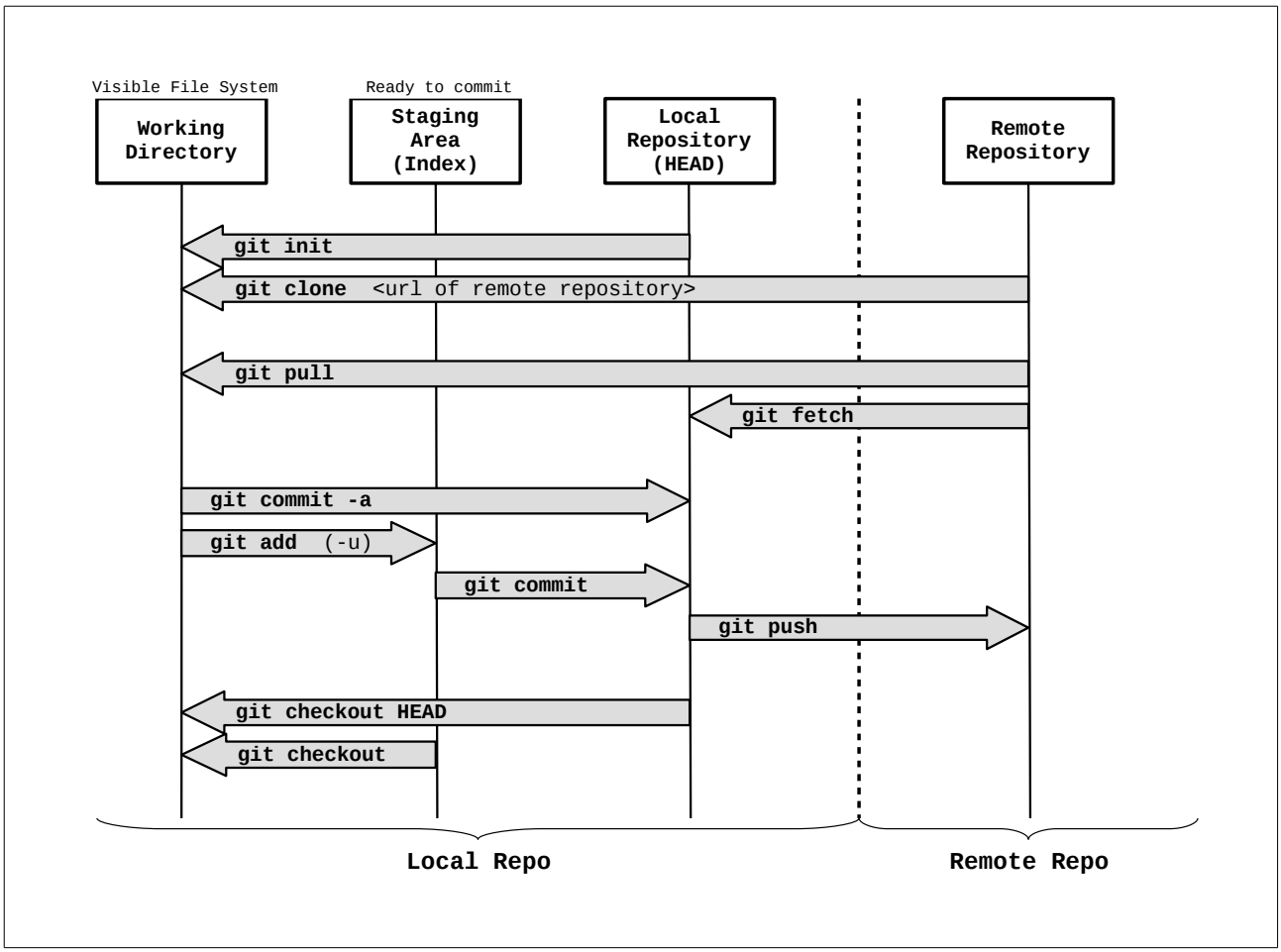


Figure 1: Estructura y comandos habituales de **Git**.

Sesión 5. Configurar Git

Para configura **Git** para nuestra práctica realizaremos los siguientes tres pasos:

1. Autenticación

Para realizar cualquier cambio en un repositorio de **Git** necesitamos estar autenticados. Hay dos opciones para hacer esto: usar el nombre de usuario y la contraseña, o usar claves pública y privada. En esta práctica usaremos la segunda opción.

Para crear las claves **SSH** de autenticación tenemos que ejecutar el comando **`ssh-keygen`**. Este proceso generará dos claves (en el directorio base `/home/seso`): el archivo público **`.ssh/id_rsa.pub`** y el archivo privado **`.ssh/id_rsa`**. A menos que se indique lo contrario, se utiliza el algoritmo **RSA** para generar las claves.

Una vez creadas las claves, debemos copiar la clave pública (`cat id_rsa.pub`) en nuestra sección claves **SSH** de **Gitlab** (**`gitlab.com`** > **`SSH Keys`** > Pegar clave pública **SSH**).

2. Configuración global

A continuación debemos configurar el usuario para poder registrar quién realiza la confirmación de los cambios (*commits*):

- `git config --global user.email "tu_email"`
- `git config --global user.name "tu_username"`

3. Push de una carpeta existente

Finalmente vamos a trabajar en el directorio en el que hemos utilizado en las semanas anteriores. En él inicializaremos un nuevo repositorio de **Git** y lo vincularemos a la cuenta de **GitLab**:

- `cd ~/SESO/projects`
- `git init`
- `git remote add origin git@gitlab.com:SE-SO/XX.git`
  {donde XX es el identificador del grupo que aparece en Gitlab}
- `git add seso`
- `git commit -m "Primer commit del grupo XX"`
- `git push -u origin master`
  {Are you sure you want to continue connecting (yes/no)? yes}

Y a partir de este momento deberéis funcionar con el flujo de trabajo habitual de Git.