Sistemas Operativos
El Proceso

De Programa a Proceso

Conceptos clave:

- Proceso de compilación
- Object files y ejecutables
- Linker y Loader
- Formato ELF

El Programa

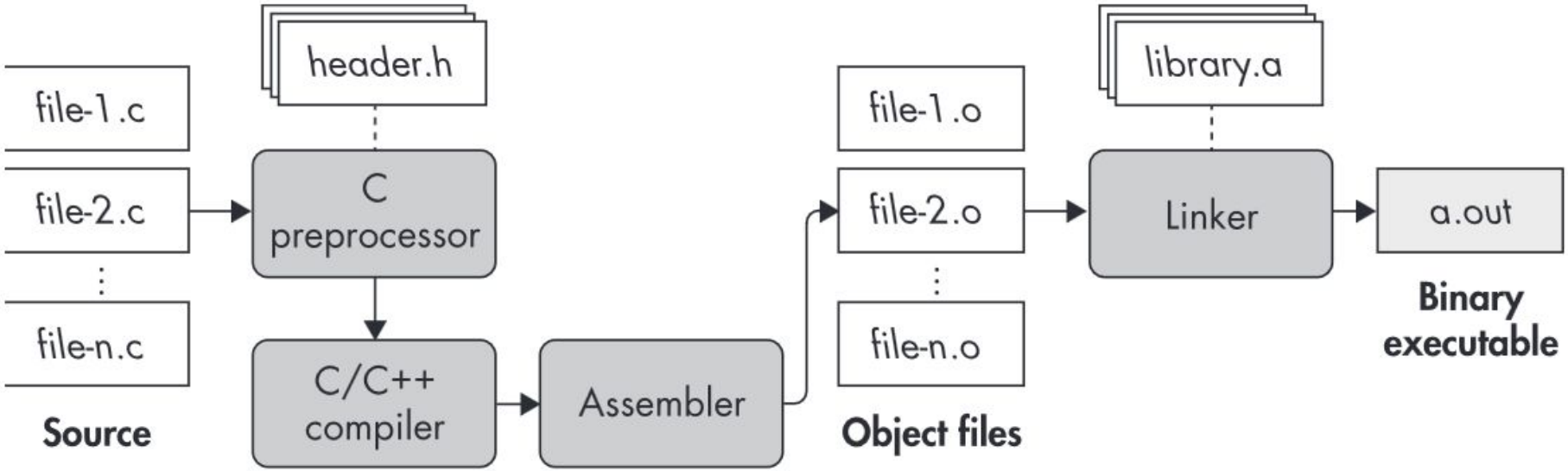
1. Programador edita código fuente
2. El compilador compila el source code en una secuencia de instrucciones de máquina.
3. El compilador guarda en disco esa secuencia, junto con datos y metadata del programa: programa ejecutable.

El Programa: Edición

```
#include<stdio.h>

#define FORMAT_STRING “%s”
#define MESSAGE     “Hello , World\n”

int main(){
        printf(FORMAR_STRING,MESSAGE);
        return 0;

}
```

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

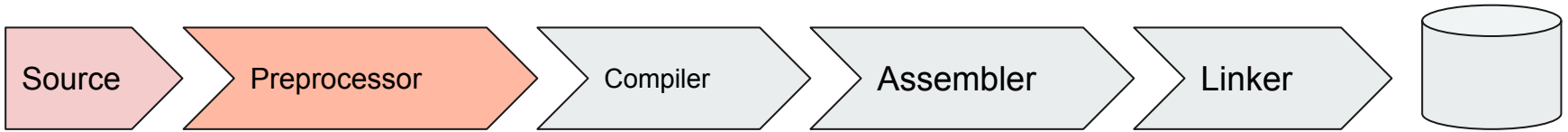
El Programa: Compilación



El Programa: Compilación

La fase de procesamiento. El preprocesador (cpp) modifica el código de fuente original de un programa escrito en C de acuerdo a las directivas que comienzan con un caracter(#). El resultado de este proceso es otro programa en C con la extensión .i

$gcc -E -P ejemplo.c

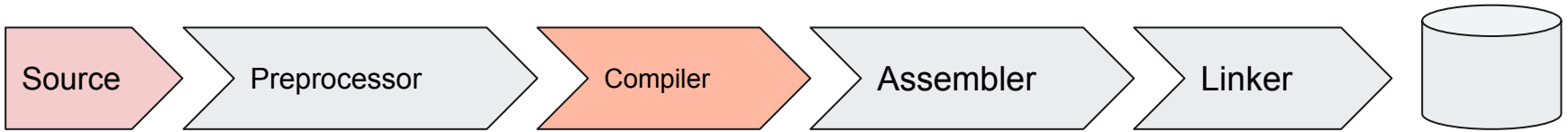


El Programa: Compilación

La fase de compilación. El compilador (cc) traduce el programa .i a un archivo de texto .s que contiene un programa en lenguaje assembly.

$gcc -S -masm=intel ejemplo.c

$cat

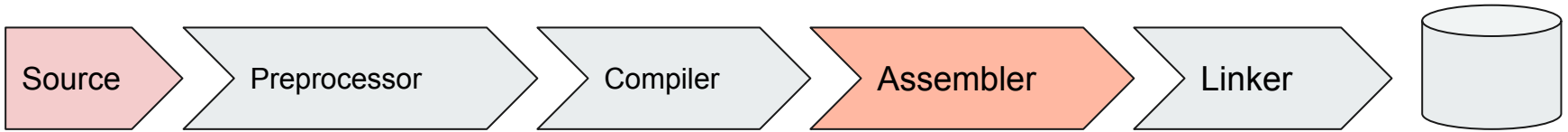


El Programa: Compilación

La fase de ensamblaje. A continuación el ensamblador (as) traduce el archivo .s en instrucciones de lenguaje de máquina empaquetándolas en un formato conocido como programa objeto realocable. Este es almacenado en un archivo con extensión .o. En este punto es donde se genera código máquina real. Que pertenece al formato ELF

$gcc -c ejemplo.c

$file ejemplo.o

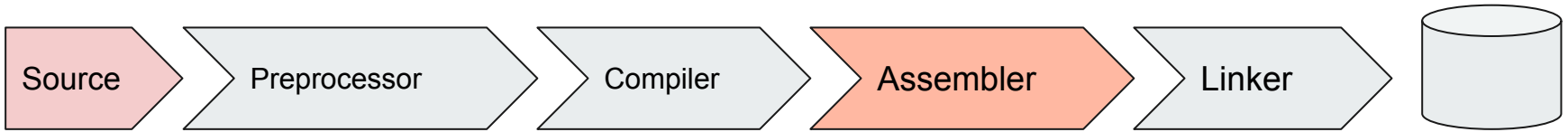


El Programa: Compilación

Un **archivo reubicable (relocatable)** es un tipo de archivo que, como su nombre indica, puede ser movido o reubicado en diferentes áreas de la memoria sin afectar su ejecución o funcionamiento.

Los archivos objeto se compilan independientemente unos de otros, por lo que el ensamblador no tiene forma de conocer las direcciones de memoria de otros archivos objeto al ensamblar un archivo objeto.

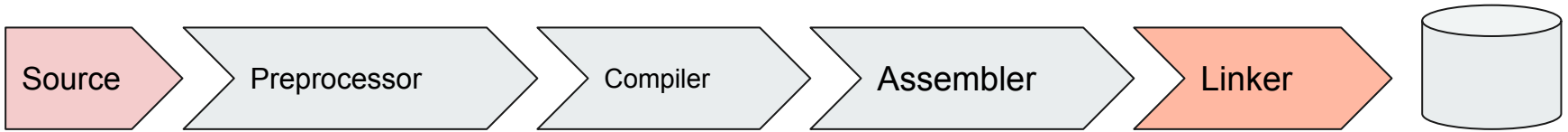
Por eso los archivos objeto necesitan ser reubicables; de esa manera, puedes enlazarlos juntos en cualquier orden para formar un ejecutable binario completo. Si los archivos objeto no fueran reubicables, esto no sería posible.



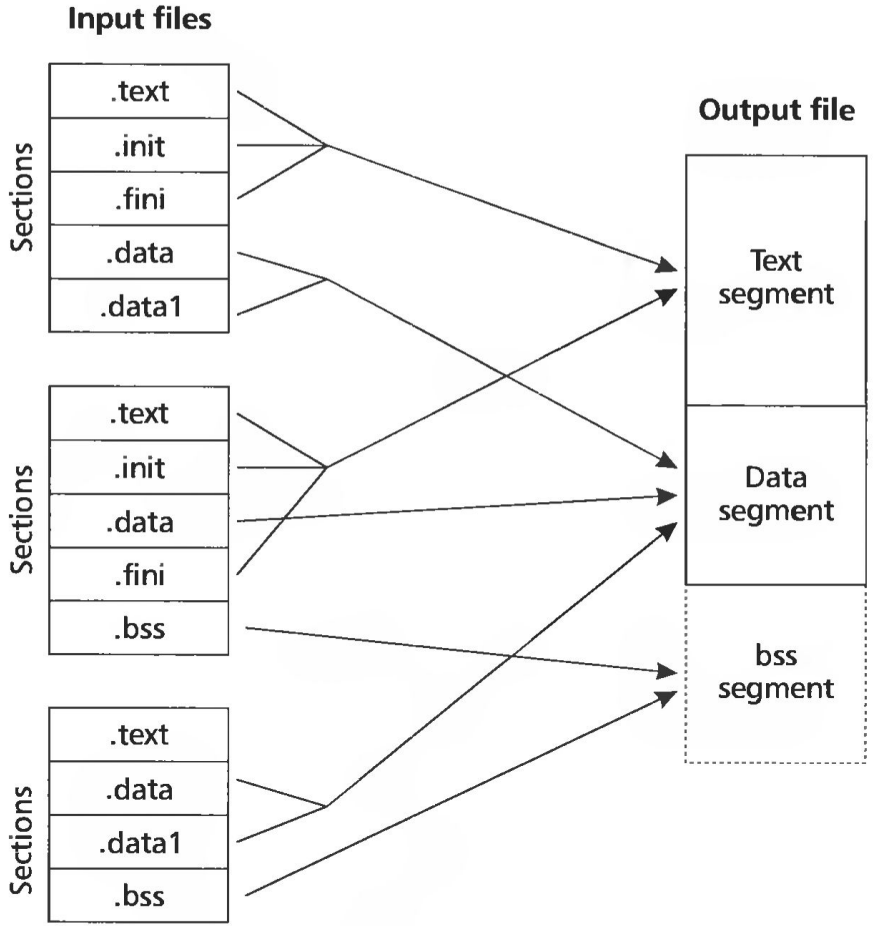
El Programa: Compilación

La fase de link edición. El proceso de linkedición es la última de las fases del proceso de compilación. Esta fase enlaza todos los archivos objetos en un único archivo binario ejecutable. Aqui el link editor ordena los archivos objetos en un único y coherente archivo ejecutable.

Además se enlazan las funciones de las bibliotecas estáticas (en C .a) y por último deja referencias simbólicas a las bibliotecas compartidas (shared library)



El Programa: Compilación



El proceso de link edicion

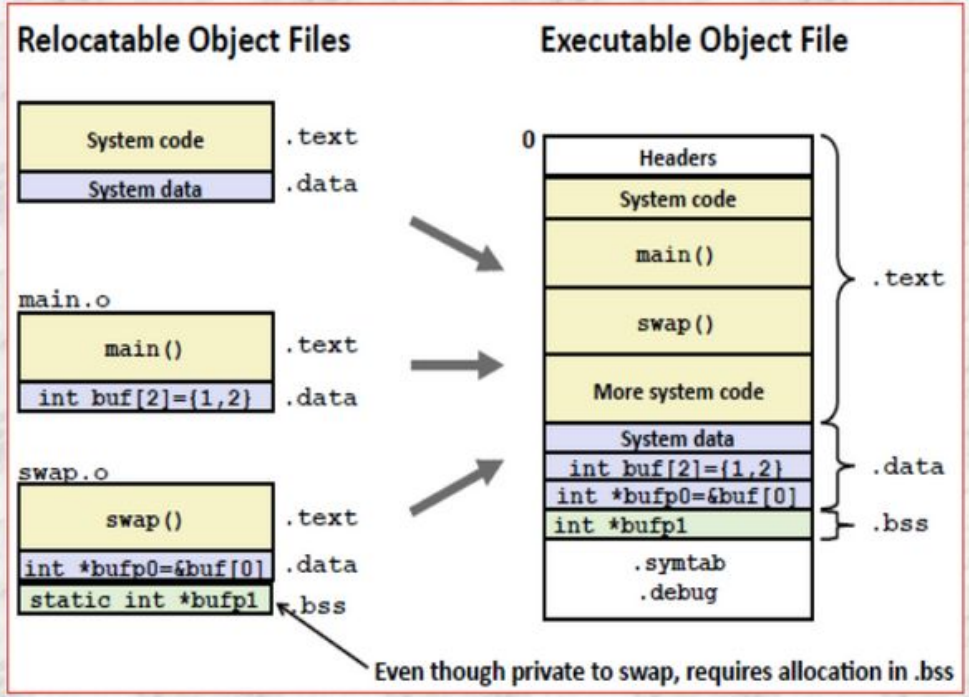
El Programa: Compilación

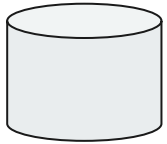
main.c

```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```



Source → Preprocessor → Compiler → **Assembler** → Linker → 

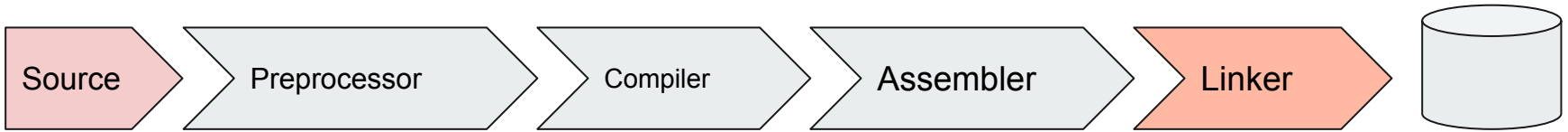
El Programa: Compilación

$ gcc ejemplo.c

$ file a.out

Salida:

a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,BuildID[sha1]=d0e23ea731bce9de65619cadd58b14ecd8c015c7,not stripped



El Programa: Formato Ejecutable

Un ***programa*** es un archivo que posee toda la información de cómo construir un proceso en memoria [KER](cap. 6).

- **Instrucciones de Lenguaje de Máquina**: Almacena el código del algoritmo del programa.

- **Dirección del Punto de Entrada del Programa**: Identifica la dirección de la instrucción con la cual la ejecución del programa debe iniciar.

- **Datos**: El programa contiene valores de los datos con los cuales se deben inicializar variables, valores de constantes y de literales utilizadas en el programa.

El Programa: Formato Ejecutable

- **Símbolos y Tablas de Realocación**: Describe la ubicación y los nombres de las funciones y variables de todo el programa, así como otra información que es utilizada por ejemplo para debugg.

- **Bibliotecas Compartidas o Dinamicas**: describe los nombres de las bibliotecas compartidas que son utilizadas por el programa en tiempo de ejecución así como también la ruta del linker dinámico que debe ser usado para cargar dicha biblioteca.

- **Otra información**: El programa contiene además otra información necesaria para terminar de construir el proceso en memoria.

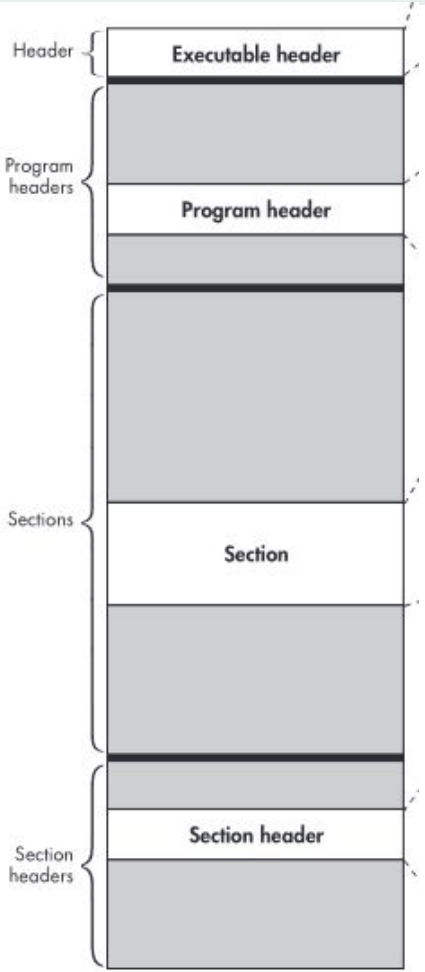
Un archivo ejecutable en Linux: ELF

ELF: **E**xecutable and **L**inking **F**ormat

Magic number 0x7F 'E' 'L' 'F

https://greek0.net/elf.html

Un archivo ejecutable en Linux: ELF



Un archivo ejecutable en Linux: ELF

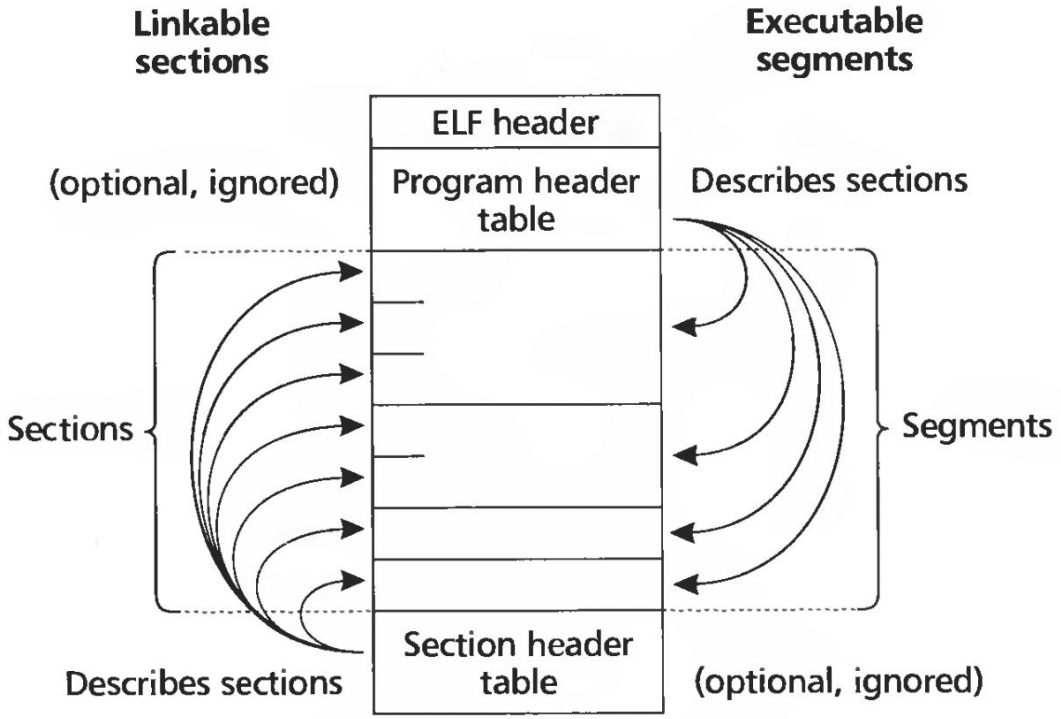
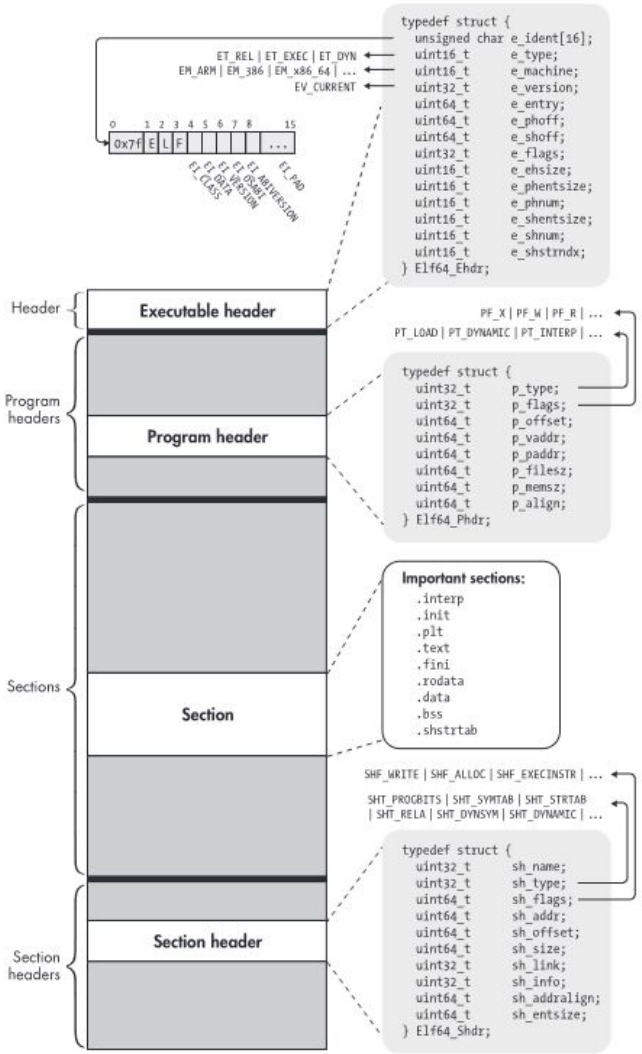


FIGURE 3.10 • Two views of an ELF file.

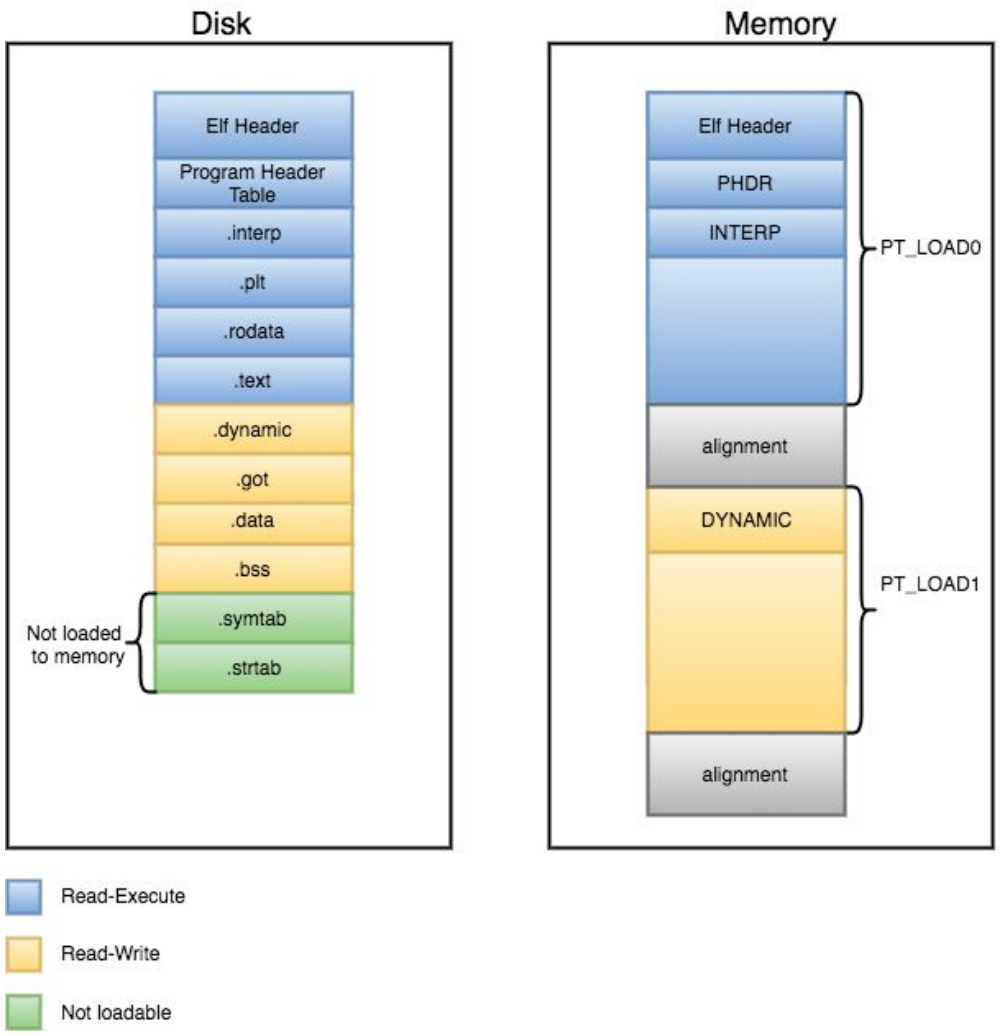

typedef struct {
unsigned char e_ident[16];
uint16_t e_type;
uint16_t e_machine;
uint32_t e_version;
uint64_t e_entry;
uint64_t e_phoff;
uint64_t e_shoff;
uint32_t e_flags;
uint16_t e_ehsize;
uint16_t e_phentsize;
uint16_t e_phnum;
uint16_t e_shentsize;
uint16_t e_shnum;
uint16_t e_shstrndx;
} Elf64_Ehdr;
ET_REL | ET_EXEC | ET_DYN
EM_ARM | EM_386 | EM_x86_64 | ...
EV_CURRENT
0 1 2 3 4 5 6 7 8 15
0x7f E L F ...
EI_CLASS
EI_DATA
EI_VERSION
EI_OSABI
EI_ABIVERSION
EI_PAD
Header
Executable header
Program headers
Program header
PF_X | PF_W | PF_R | ...
PT_LOAD | PT_DYNAMIC | PT_INTERP | ...
typedef struct {
uint32_t p_type;
uint32_t p_flags;
uint64_t p_offset;
uint64_t p_vaddr;
uint64_t p_paddr;
uint64_t p_filesz;
uint64_t p_memsz;
uint64_t p_align;
} Elf64_Phdr;
Sections
Section
Important sections:
.interp
.init
.plt
.text
.fini
.rodata
.data
.bss
.shstrtab
SHF_WRITE | SHF_ALLOC | SHF_EXECINSTR | ...
SHT_PROGBITS | SHT_SYMTAB | SHT_STRTAB
| SHT_RELA | SHT_DYNSYM | SHT_DYNAMIC | ...
typedef struct {
uint32_t sh_name;
uint32_t sh_type;
uint64_t sh_flags;
uint64_t sh_addr;
uint64_t sh_offset;
uint64_t sh_size;
uint32_t sh_link;
uint32_t sh_info;
uint64_t sh_addralign;
uint64_t sh_entsize;
} Elf64_Shdr;
Section headers
Section header

Un Programa en Linux: ELF

```
struct Elf64_Ehdr {
    unsigned char e_ident[EI_NIDENT];
    Elf64_Half e_type;
    Elf64_Half e_machine;
    Elf64_Word e_version;
    Elf64_Addr e_entry;
    Elf64_Off e_phoff;
    Elf64_Off e_shoff;
    Elf64_Word e_flags;
    Elf64_Half e_ehsize;
    Elf64_Half e_phentsize;
    Elf64_Half e_phnum;
    Elf64_Half e_shentsize;
    Elf64_Half e_shnum;
    Elf64_Half e_shstrndx;
};
```

```
struct Elf64_Phdr {
    Elf64_Word p_type;
    Elf64_Word p_flags;
    Elf64_Off p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    Elf64_Xword p_filesz;
    Elf64_Xword p_memsz;
    Elf64_Xword p_align;
};
```

El archivo ELF además de tener el código, es como un “plano” del proceso que el **Loader** (parte del sistema operativo) va a usar para “construir” el proceso durante **execve()**

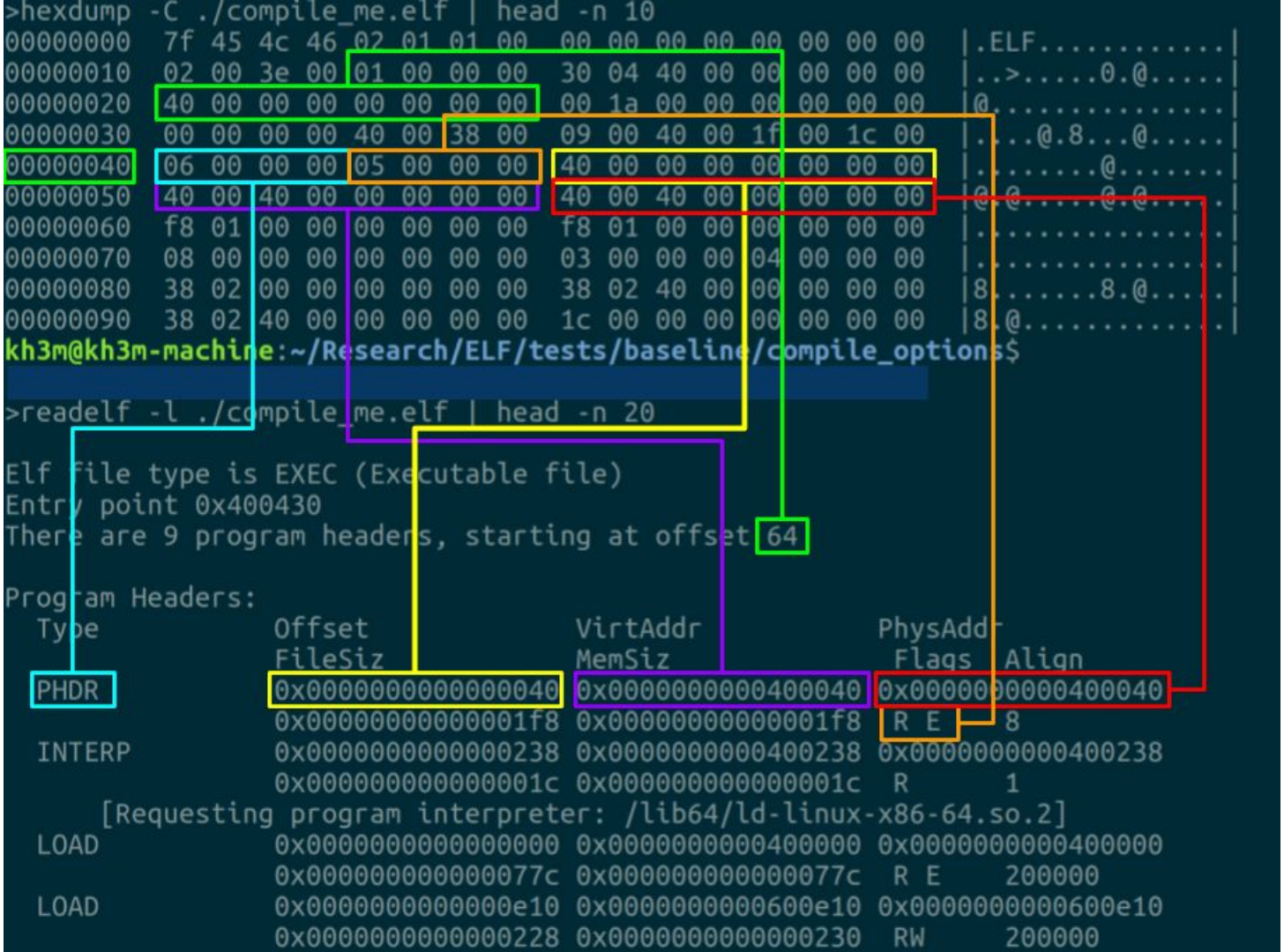


Un Programa en Linux: ELF

- **readelf** is a Unix binary utility that displays information about one or more ELF files. A free software implementation is provided by GNU Binutils.
- **elfutils** provides alternative tools to GNU Binutils purely for Linux.[11]
- **objdump** provides a wide range of information about ELF files and other object formats. objdump uses the Binary File Descriptor library as a back-end to structure the ELF data.
- The Unix **file** utility can display some information about ELF files, including the instruction set architecture for which the code in a relocatable, executable, or shared object file is intended, or on which an ELF core dump was produced.

El Comando readelf

Ver el SHT (Section Header Table)	readelf -S <archivo>
Ver el PHT (Program Header table)	readelf -l <archivo>
Ver el ELF Header	readelf -h <archivo>
Ver la Realocation Table	readelf -r <archivo>
Ver el dump hexa	hexdump -C archivo | head -n 10


>hexdump -C ./compile_me.elf | head -n 10
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |.ELF............|
00000010 02 00 3e 00 01 00 00 00 30 04 40 00 00 00 00 00 |..>.....0.@.....|
00000020 40 00 00 00 00 00 00 00 00 1a 00 00 00 00 00 00 |@...............|
00000030 00 00 00 00 40 00 38 00 09 00 40 00 1f 00 1c 00 |....@.8...@.....|
00000040 06 00 00 00 05 00 00 00 40 00 00 00 00 00 00 00 |........@.......|
00000050 40 00 40 00 00 00 00 00 40 00 40 00 00 00 00 00 |@.@.....@.@.....|
00000060 f8 01 00 00 00 00 00 00 f8 01 00 00 00 00 00 00 |................|
00000070 08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00 |................|
00000080 38 02 00 00 00 00 00 00 38 02 40 00 00 00 00 00 |8.......8.@.....|
00000090 38 02 40 00 00 00 00 00 1c 00 00 00 00 00 00 00 |8.@.............|
kh3m@kh3m-machine:~/Research/ELF/tests/baseline/compile_options$
>readelf -l ./compile_me.elf | head -n 20
Elf file type is EXEC (Executable file)
Entry point 0x400430
There are 9 program headers, starting at offset 64
Program Headers:
Type Offset VirtAddr PhysAddr
FileSiz MemSiz Flags Align
PHDR 0x0000000000000040 0x0000000000400040 0x0000000000400040
0x00000000000001f8 0x00000000000001f8 R E 8
INTERP 0x0000000000000238 0x0000000000400238 0x0000000000400238
0x000000000000001c 0x000000000000001c R 1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD 0x0000000000000000 0x0000000000400000 0x0000000000400000
0x000000000000077c 0x000000000000077c R E 200000
LOAD 0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
0x0000000000000228 0x0000000000000230 RW 200000

Proceso

Conceptos clave:

- Definición y componentes
- Estructura en memoria del proceso
- System call: brk()

El Proceso

“Un proceso es la ejecución de un programa de aplicación con derechos restringidos; el proceso es la abstracción que provee el Kernel del sistema operativo para la ejecución protegida”- [DAH]

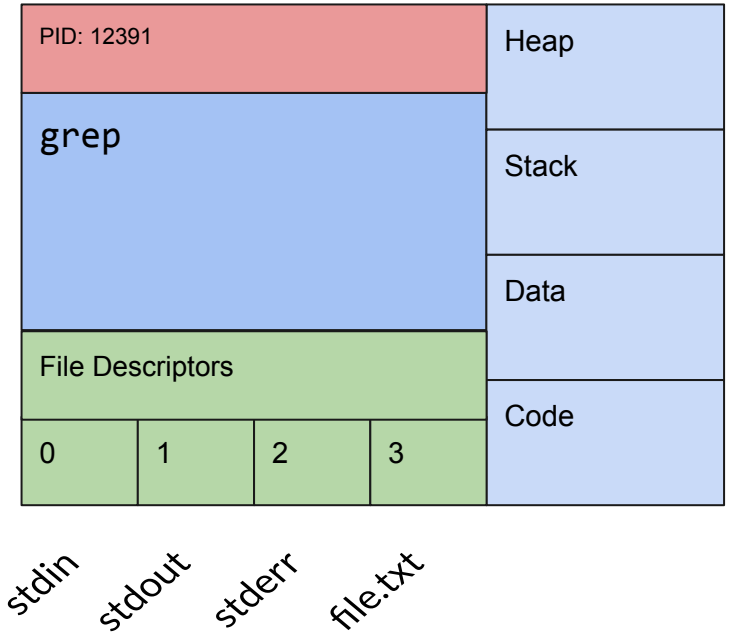
“Es simplemente un programa que se está ejecutando en un instante dado” - [ARP]

“Un Proceso es la instancia de un programa en ejecución” - [VAH]

“Un proceso es un programa en medio de su ejecución” - [LOV]

User Space

El habitante de User Land: **el proceso**



El Proceso

Por supuesto que no está más lejos de la verdad, decir que un proceso es sólo un programa en ejecución. Un proceso incluye:

- Los Archivos abiertos
- Las señales(signals) pendientes
- Datos internos del kernel
- El estado completo del procesador
- Un espacio de direcciones de memoria
- Uno o más hilos de ejecución. Cada thread contiene
  - Un único contador de programa
  - Un Stack
  - Un Conjunto de Registros
  - Una sección de datos globales

El Proceso

“Un **Proceso** es una entidad abstracta, definida por el Kernel, en la cual los recursos del sistema son asignados” [KER]

El Proceso: Virtualización

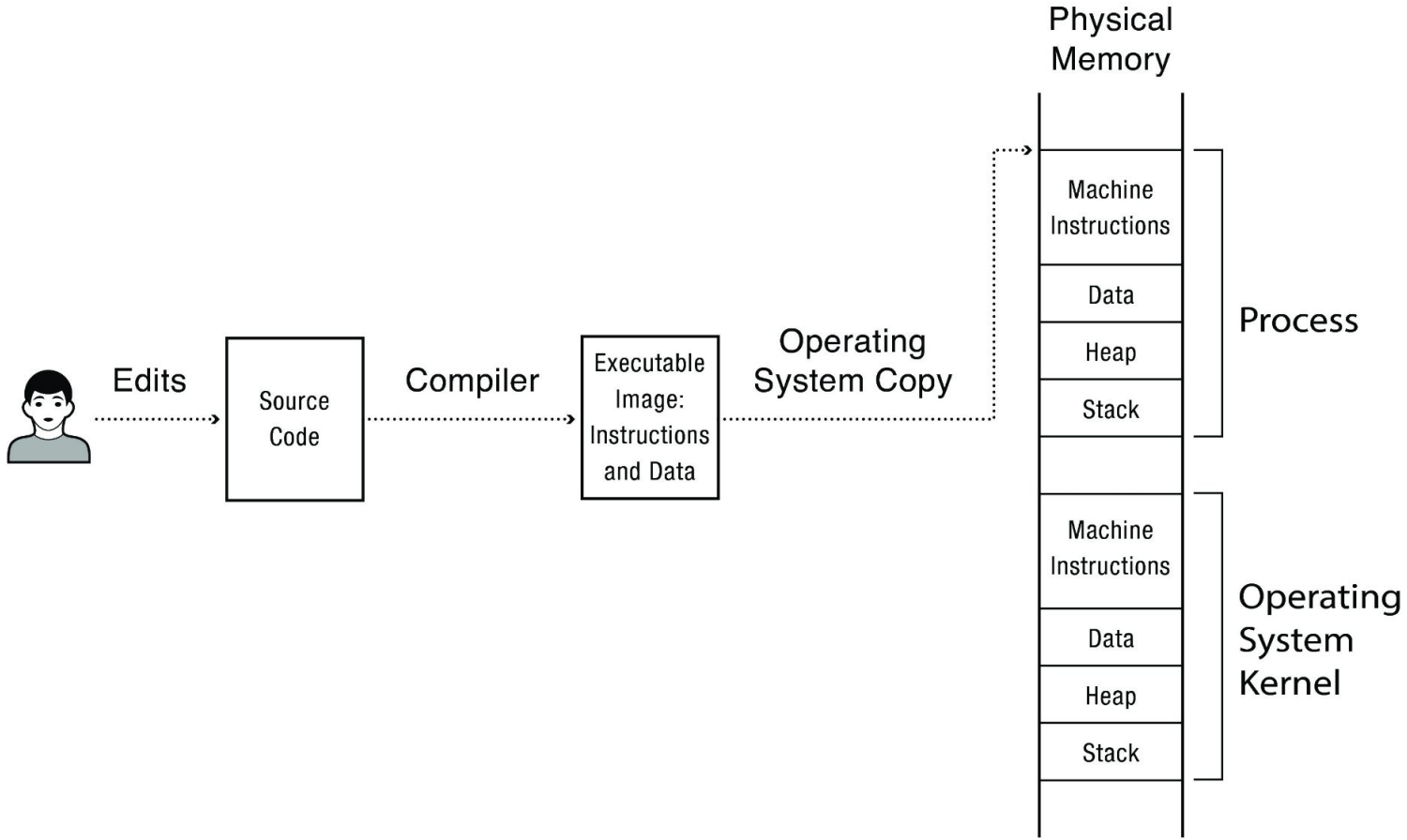
- Virtualización de Memoria
- Virtualización de Procesamiento

De Programa a Proceso

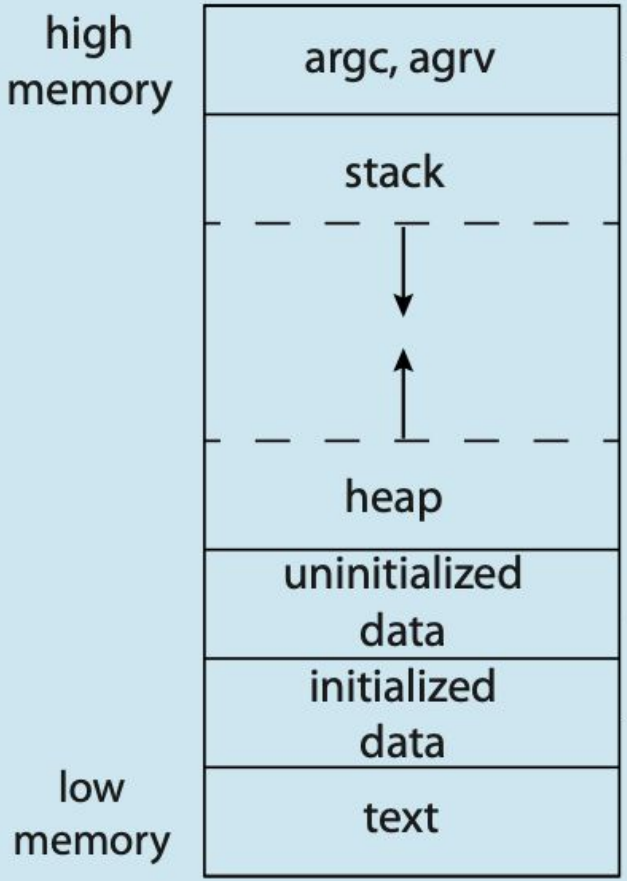
El Sistema Operativo más precisamente el Kernel se encarga de:

1. Cargar instrucciones y Datos de un programa ejecutable en memoria.
2. Crear el Stack y el Heap
3. Transferir el Control al programa
4. Proteger al SO y al Programa

El proceso en memoria



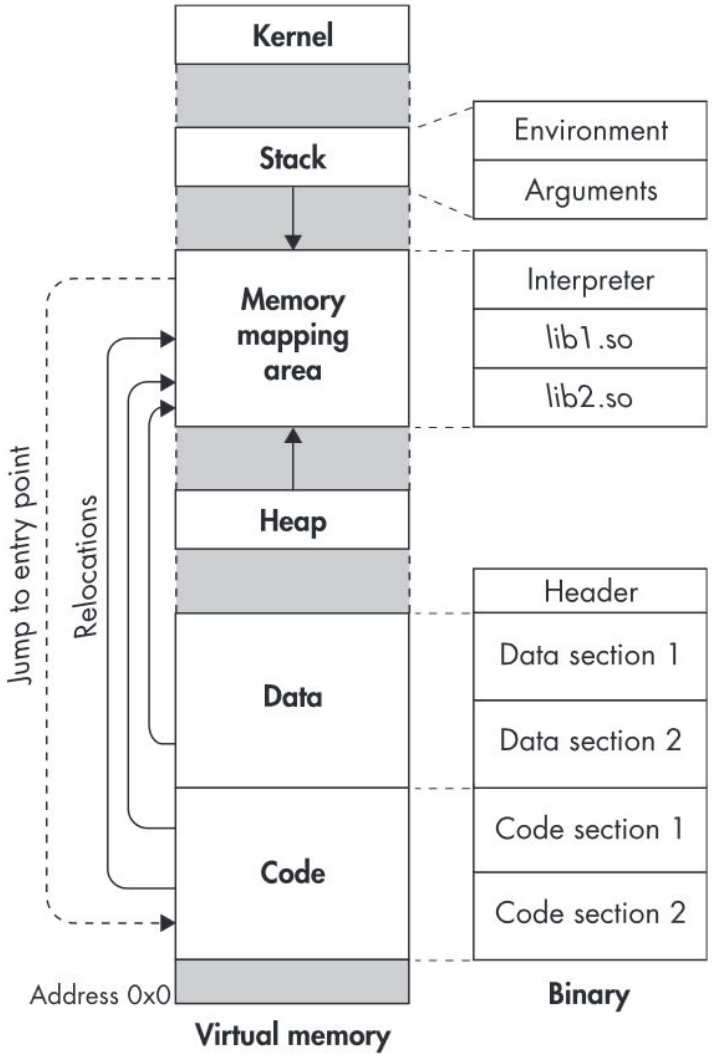
El proceso en memoria




```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

El proceso en memoria

El **Loader** usa el “plano del proceso” que es el archivo ELF y arma el espacio de direcciones



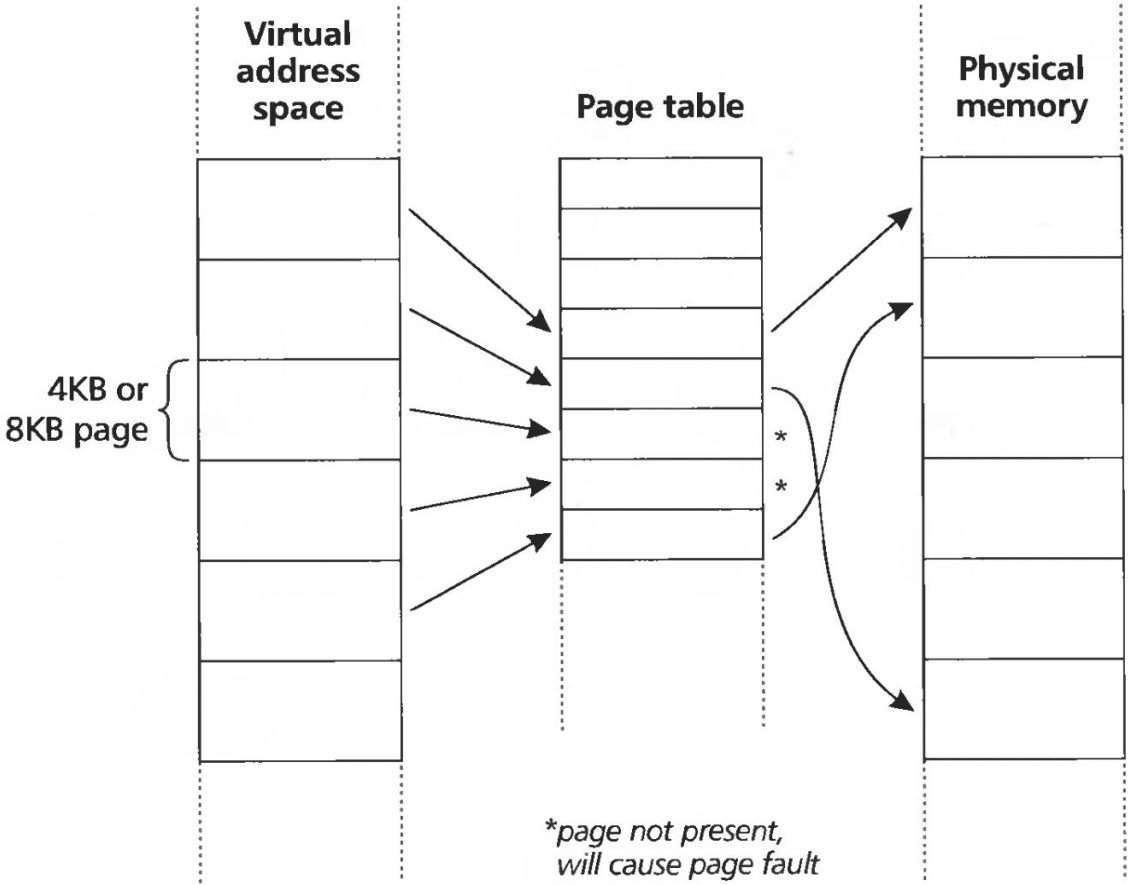
Virtualización de Memoria

- Uno de estos mecanismos es denominado **Memoria Virtua**l, la memoria virtual es una abstracción por la cual la memoria física puede ser compartida por diversos procesos.

- Un componente clave de la memoria virtual son las **direcciones virtuales**, con las direcciones virtuales, para cada proceso su memoria inicia en el mismo lugar, la dirección 0.

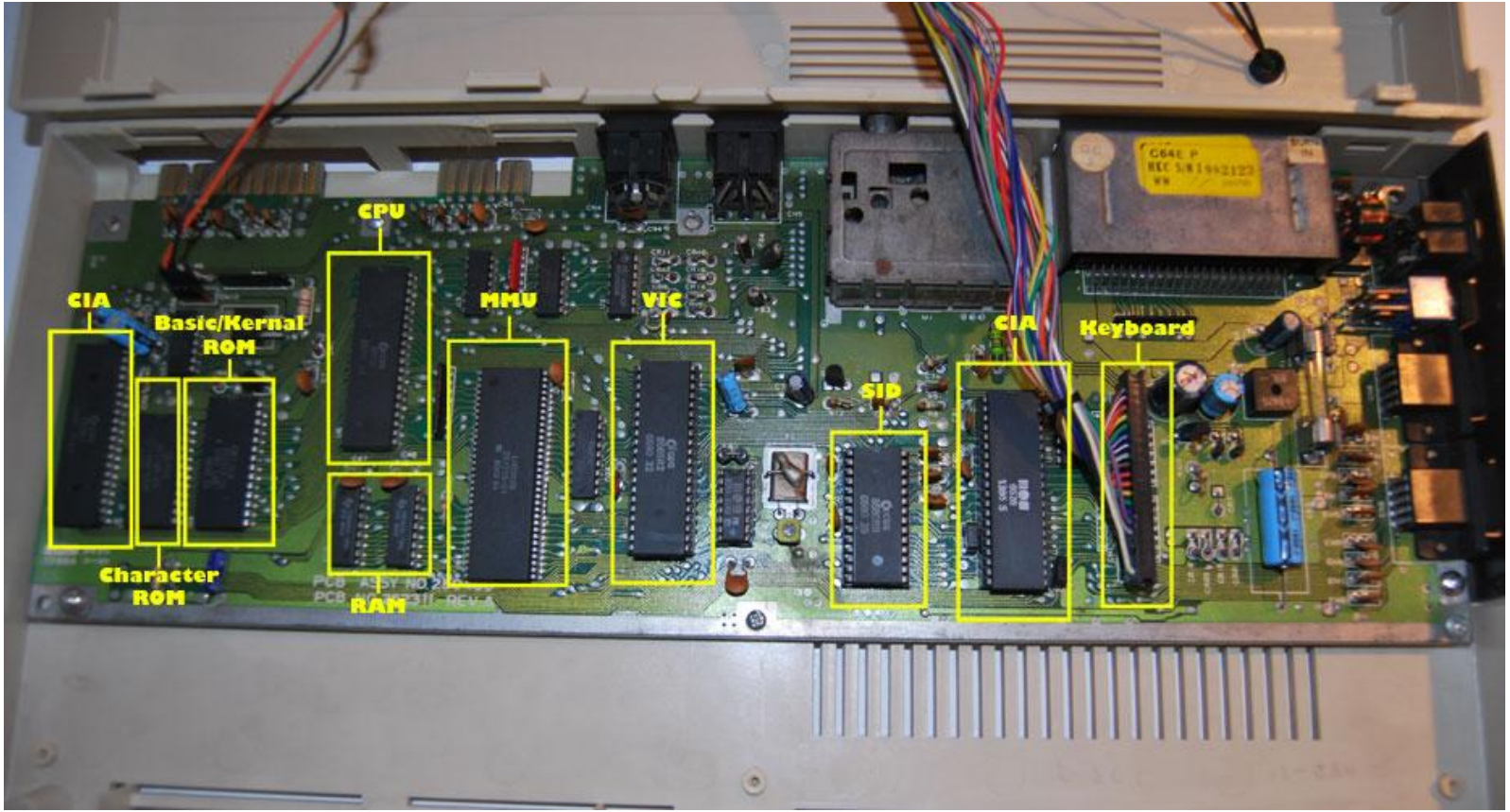
- Cada proceso piensa que tiene toda la memoria de la computadora para sí mismo, si bien obviamente esto en la realidad no sucede. El hardware traduce la dirección virtual a una dirección física de memoria.

Virtualización de Memoria

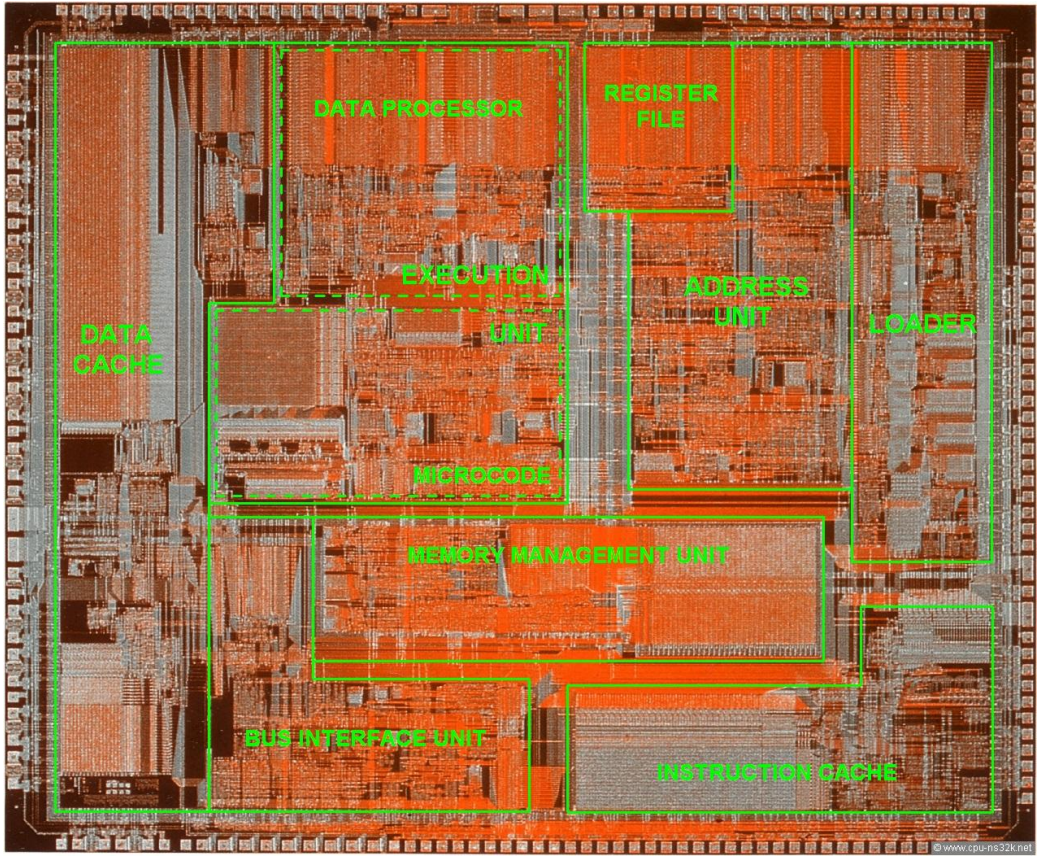


Traducción de Direcciones

Se traduce una Dirección Virtual (emitida por la CPU) en una Dirección Física (la memoria). Este mapeo se realiza por hardware, más específicamente por Memory Management Unit (MMU).



Commodore 64

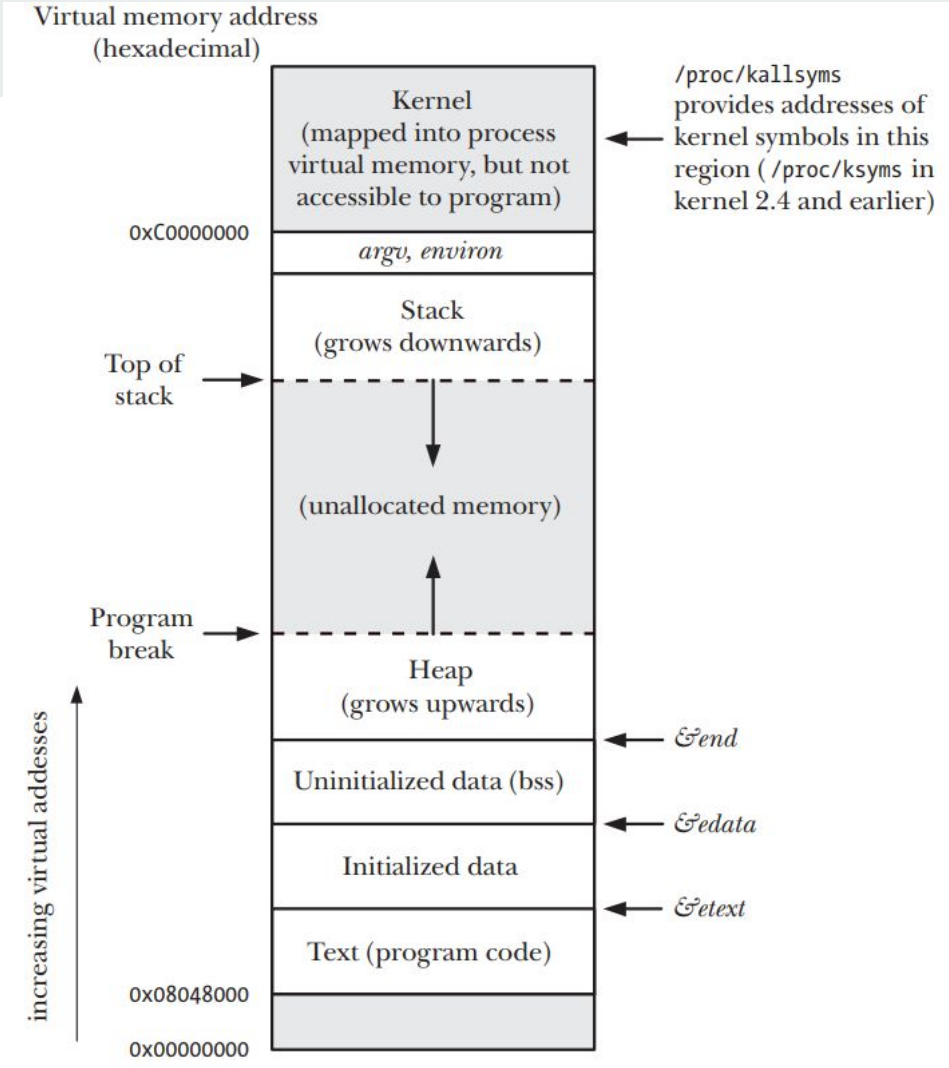


La MMU Dentro del Procesador

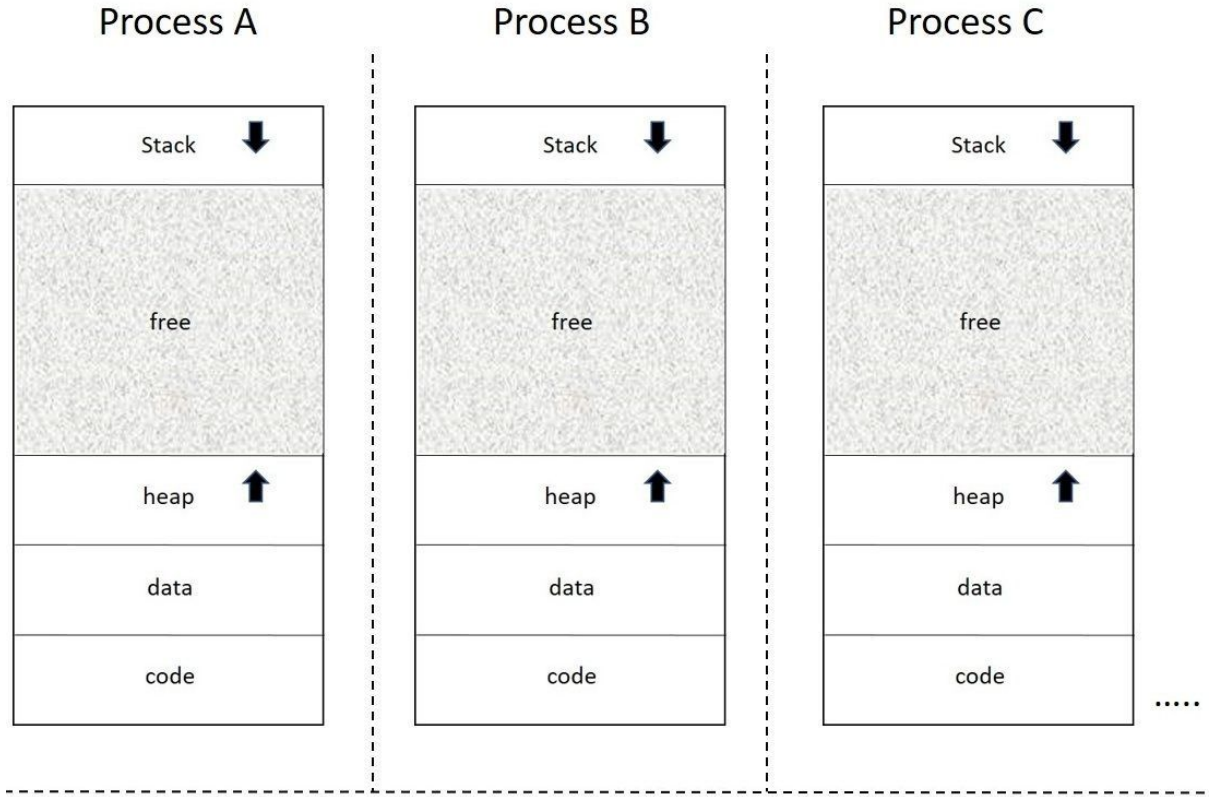
El proceso en memoria

La virtualización de memoria le hace creer al proceso que este tiene toda la memoria disponible para ser reservada y usada como si este estuviera siendo ejecutado sólo en la computadora (ilusión). Todos los procesos en Linux, está dividido en 4 segmentos:

Text: Instrucciones del Programa.
Data: Variables Globales (extern o static en C)
Heap: Memoria Dinámica Alocable
Stack: Variable Locales y trace de llamadas



El proceso en memoria



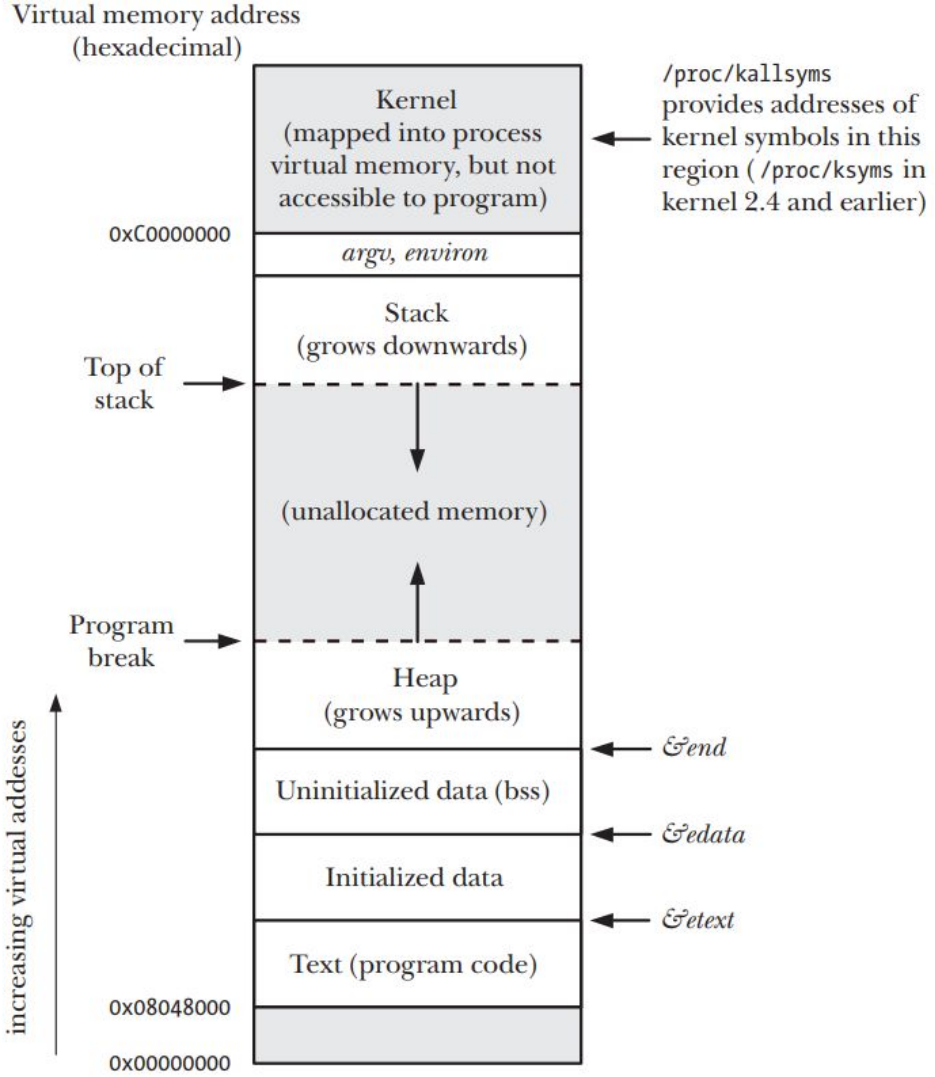
Todos los procesos tienen la misma estructura del Address Space

System call brk()

Inicialmente el break del programa está ubicado justo en el final de los datos no inicializados.

Después que brk() se ejecuta, el break es incrementado, el proceso puede acceder a cualquier memoria en la nueva área reservada, pero no accede directamente a la memoria física.

Esto se realiza automáticamente por el kernel en el primer intento del proceso en acceder al área reservada.



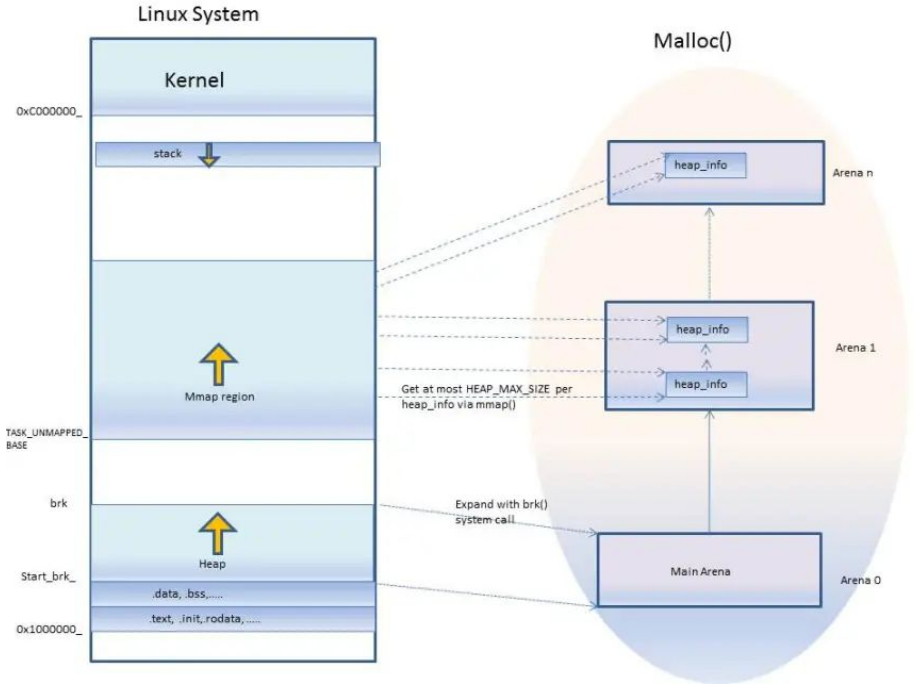
brk() vs malloc()

brk()

- Es una system call. Opera con bloques grandes (generalmente páginas completas)

malloc()

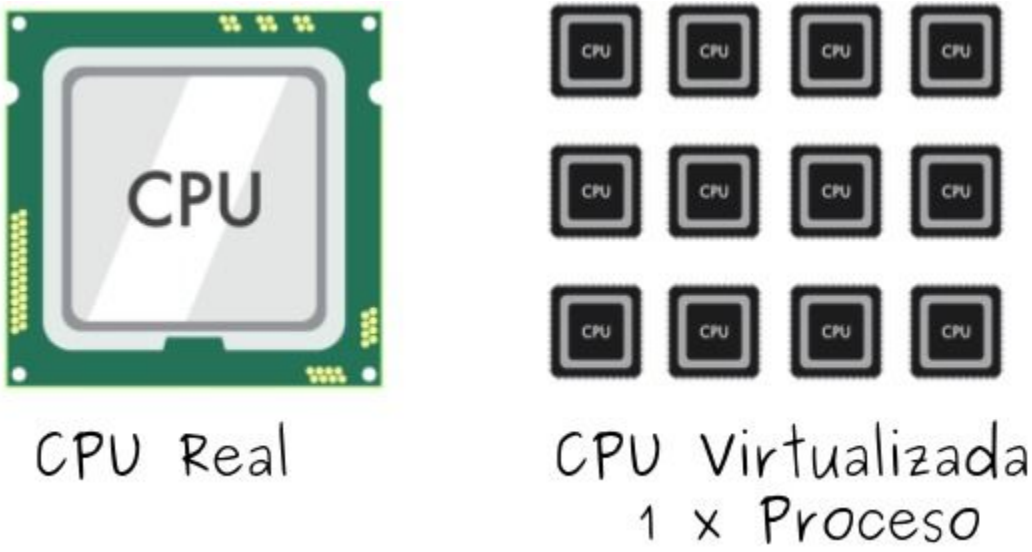
- Se implementa en espacio de usuario
- Utiliza brk().
- Maneja estructuras de datos propias (en user space) para optimizar la memoria provista por brk



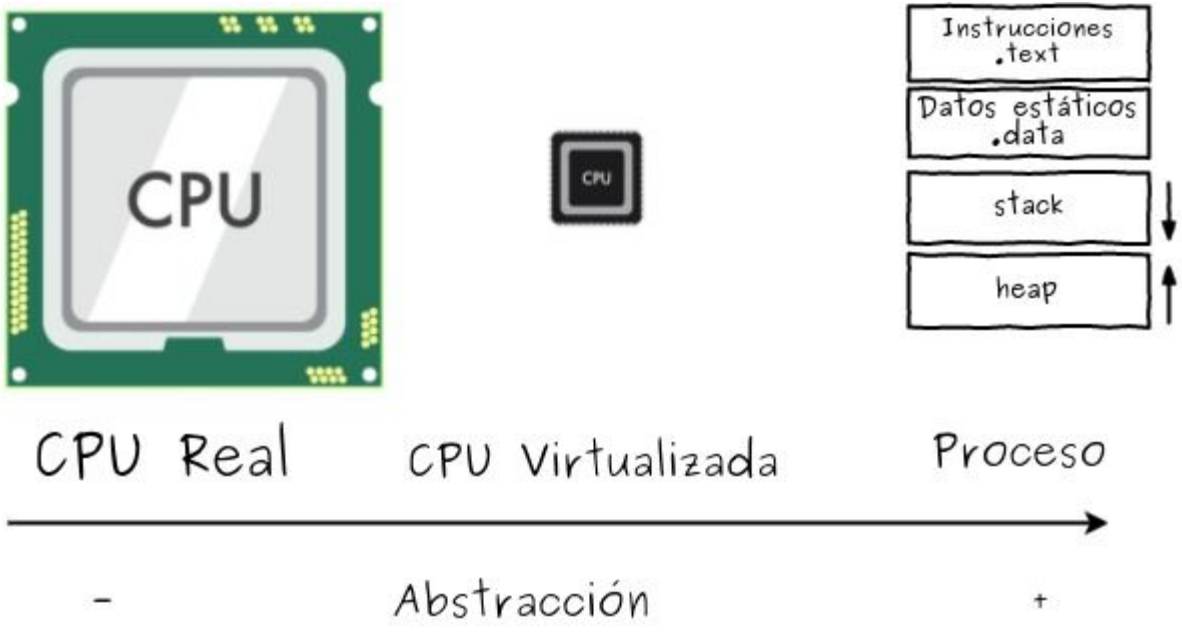
Más en la unidad 5 del programa, sobre Memoria

Virtualización de Procesador

La virtualización de procesamiento es la forma de virtualización más primitiva, **consiste en dar la ilusión de la existencia de un único procesador para cualquier programa que requiera de su uso.**



El SO crea esta ilusión mediante la virtualización de la CPU a través del kernel.


CPU
CPU
Instrucciones
.text
Datos estáticos
.data
stack
heap
CPU Real
CPU Virtualizada
Proceso
-
Abstracción
+

Estados del proceso

Conceptos clave:

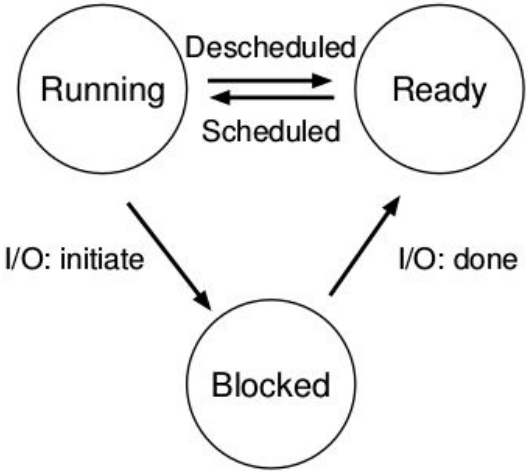
- Diagrama de estados
- Llamadas bloqueantes

Estados de un Proceso

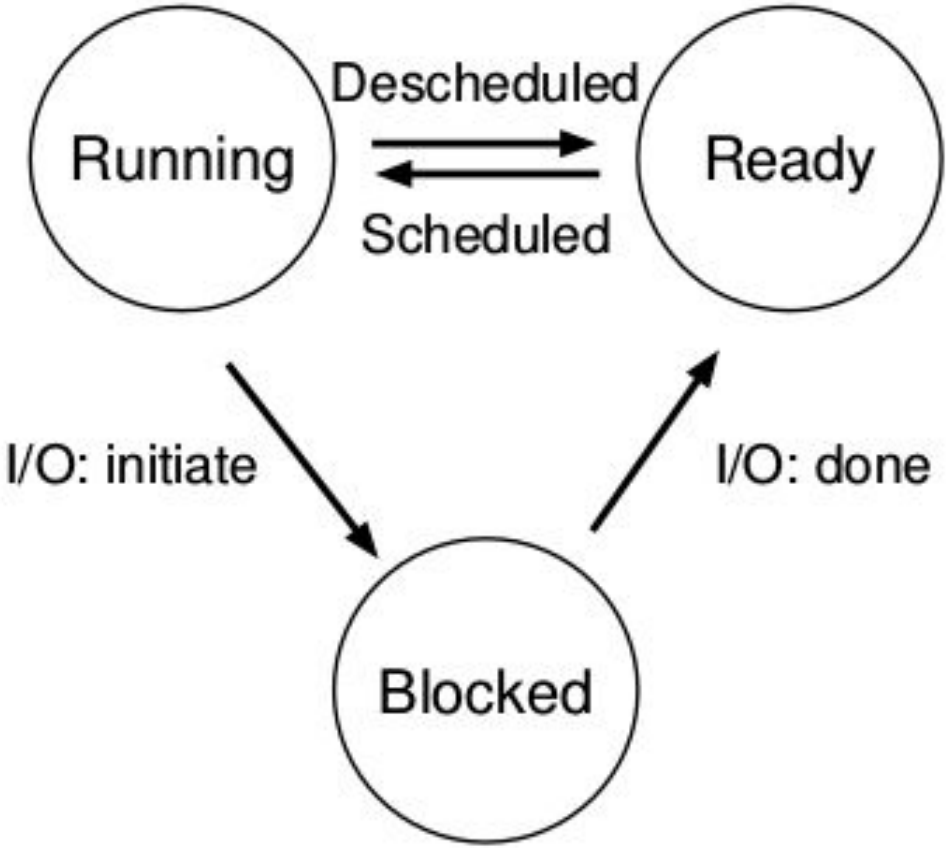
Corriendo (Running): el proceso se encuentra corriendo en un procesador. Está ejecutando instrucciones.

Listo (Ready): en este estado el proceso está listo para correr pero por algún motivo el SO ha decidido no ejecutarlo por el momento.

Bloqueado (Blocked): en este estado el proceso ha ejecutado algún tipo de operación que hace que éste no esté listo para ejecutarse hasta que algún evento suceda.

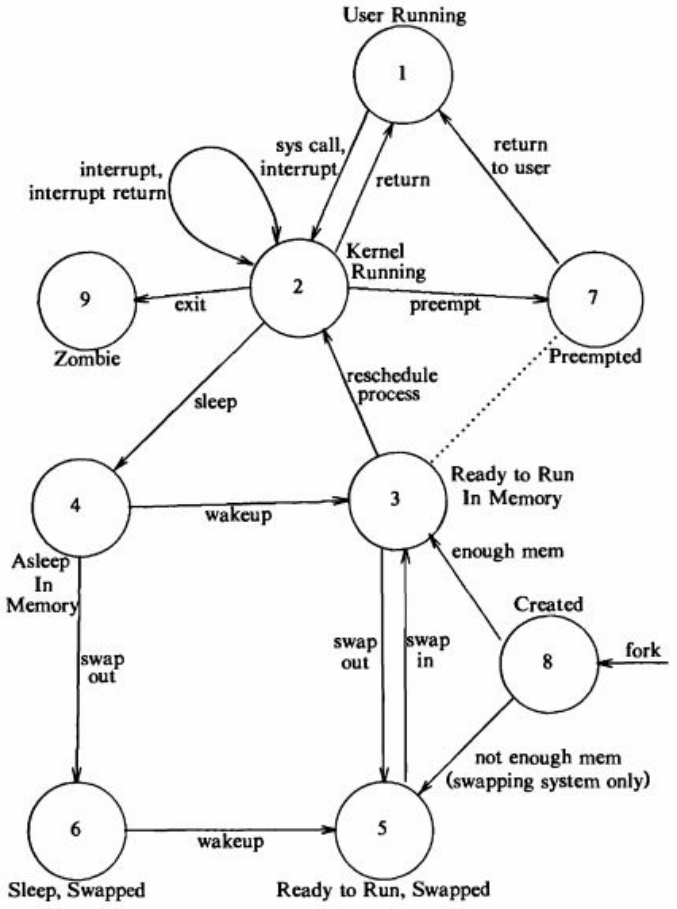


Estados de un Proceso



Ejemplo Histórico: System V

- **Corriendo User Mode(Running User Mode):** El proceso se encuentra corriendo en un procesador. Está ejecutando instrucciones.
- **Corriendo kernel Mode(Running Kernel Mode):** d
- **Listo para Correr en Memoria (Ready to Run on Memory):** En este estado el proceso está listo para correr pero por algún motivo el SO ha decidido no ejecutarlo por el momento.
- **Durmiendo en Memoria (Asleep In Memory) :** El proceso está bloqueado en memoria.
- **Listo para Correr pero Swapeado (Ready to Run but swapped):** El proceso está bloqueado en memoria secundaria.
- **Durmiendo en Memoria Secundaria (Asleep Swapped):** El proceso se encuentra bloqueado en memoria secundaria.
- **Preempt(Preempt):** Es igual a 1 pero un proceso que pasó antes por Kernel mode solo puede pasar a preentive.
- **Creado (Created):** El proceso está recién creado y en un estado de transición.
- **Zombie (Zombie):** El proceso ejecutó la S.C. exit(), ya no existe más, lo único que queda es el exit state.



Estructuras del Kernel

Conceptos clave:

- El contexto del proceso
- [User|Register|System]-Level context
- Kernel Stack

Proceso: El Contexto

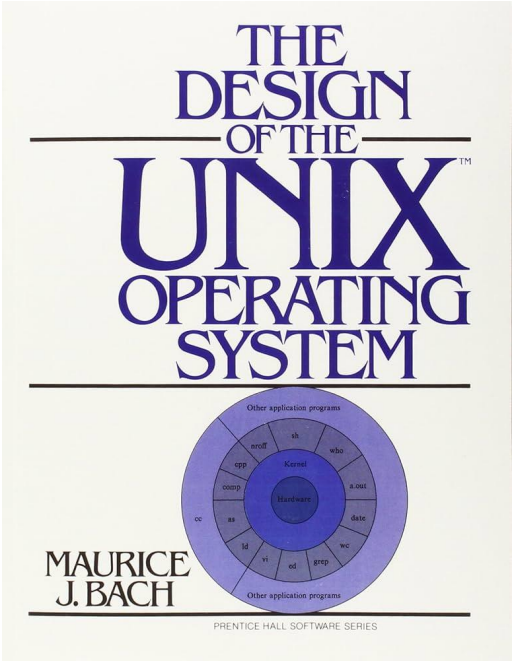
- El **contexto** de un proceso **es la información necesaria para describir completamente el estado de un proceso**.
- Cada proceso posee su propio contexto.

Proceso: El Contexto

¿Qué se necesita para describir “el estado de un proceso”?

Según Bach: el contexto de un proceso comprende:

- el contenido del address space (**memoria**)
- el contenido de los registros de hardware (**cpu**)
- las estructuras de datos que pertenecen al kernel relacionadas con el proceso (**kernel**)



Proceso: El Contexto

Formalmente, el contexto de un proceso es la unión de:

- User-level context
- Register context
- System-level context

Proceso: El Contexto

Formalmente, el contexto de un proceso es la unión de:

- User-level context (**memoria**)
- Register context (**cpu**)
- System-level context (**estructuras del kernel**)

Cada proceso ve solamente su contexto, ie. su memoria, su procesador, sus archivos abiertos, etc.

Proceso: El Contexto User-level

- Consiste en las secciones que conforman el espacio de direccion virtual del proceso
  - Text
  - Data
  - Stack
  - Heap

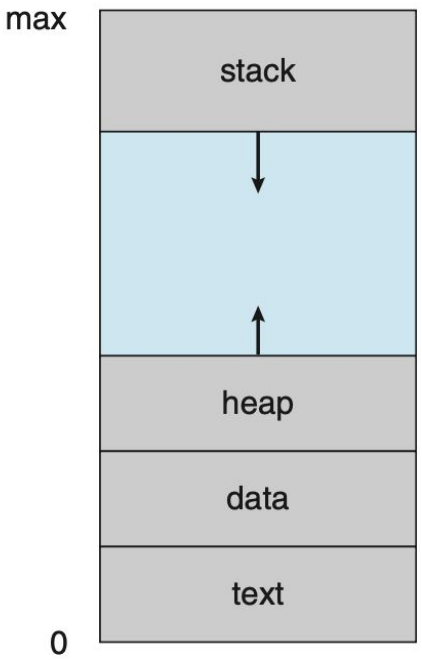


Figure 3.1 Layout of a process in memory.

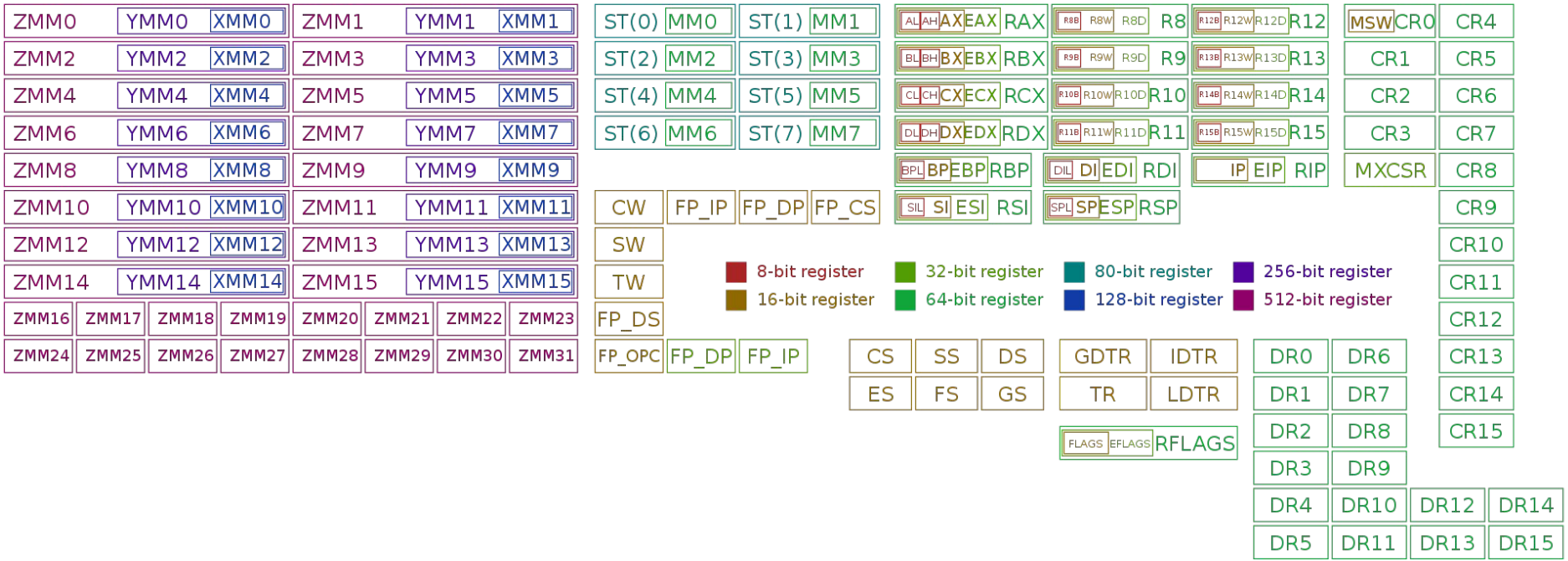
Proceso: El contexto de registros

Representa el estado del CPU, es decir, los registros del CPU

Eg. x86

- Program Counter Register
- Processor Status Register
- Stack Pointer Register
- General Purpose Registers

Proceso: El contexto de registros



System-level context

- **Process Table Entry:** La entrada en la **tabla de procesos**
- **Configuración de memoria:** define el mapeo de la memoria virtual vs memoria física del proceso. Eg. Page Tables
- **Kernel Stack**. contiene los stack frames de las llamadas a funciones hechas dentro del kernel.
- La **u area:** en desuso en sistemas modernos. Pero casi todo se incluye ahora en la Process Table Entry

System-level context: U Area

La u-area era una estructura por proceso usada en sistemas antiguos (UNIX Version 6) para almacenar información específica de cada proceso, como descriptores de archivos abiertos, registros en modo usuario y manejo de señales.

Con el tiempo, los sistemas Unix y similares, como Linux, han evolucionado y reestructurado la gestión de procesos.

En general la Process Table Entry contiene todo lo que antes estaba en la u-area

System-level context: Process Table Entry (o PCB)

Cada proceso está representado en el sistema operativo por un Process Control Block (PCB) . Un PCB se muestra en la Figura 3.3.

Contiene muchos elementos de información asociados con un proceso específico.

Como se guardan en una tabla, se suelen llamar también Process Table Entry

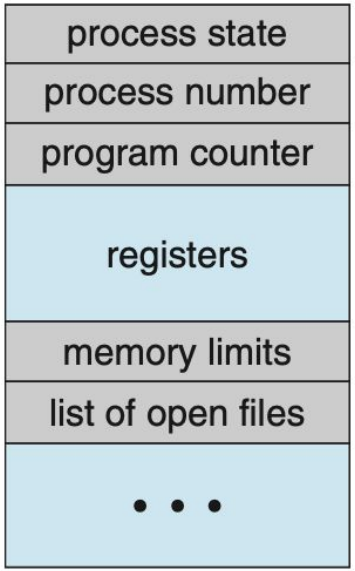
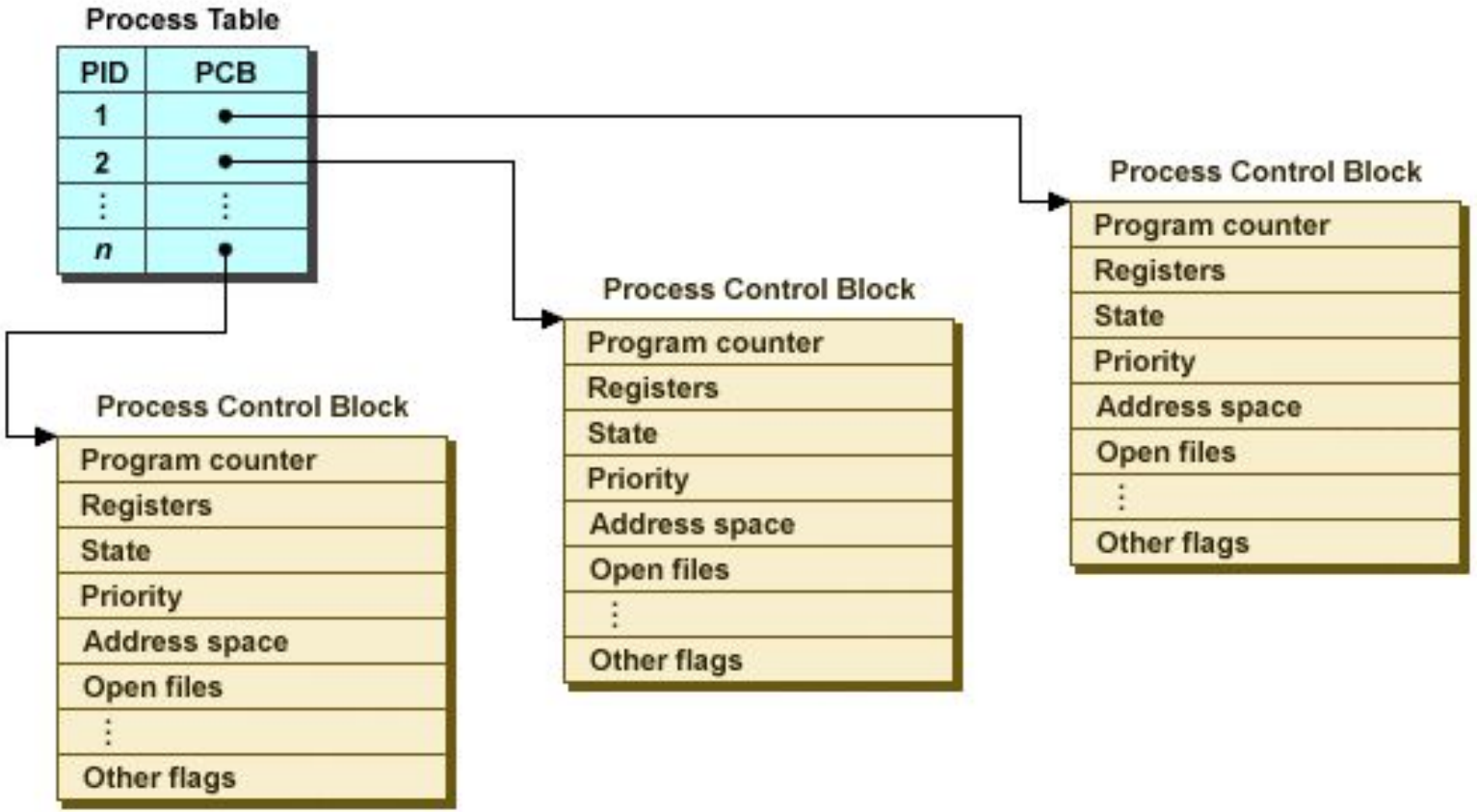


Figure 3.3 Process control block (PCB).

El Contexto System-level: Process Table

La tabla de PCB es generalmente un array o una lista enlazada



System-level context: Process Table Entry

Depende mucho de la implementación del sistema operativo. Cosas que **podría** contener el PCB:

- identificación: cada proceso tiene un identificador único o process ID (PID) y además perteneces a un determinado grupo de procesos.
- Ubicación del mapa de direcciones del Kernel del u area del proceso.
- Estado actual del proceso
- Un puntero hacia el siguiente proceso en el planificador y al anterior.
- Prioridad
- Información para el manejo de señales.
- Información para la administración de memoria.

Ejemplo PCB: struct proc (xv6)

En xv6 la tabla de procesos es simplemente **un array de 64 entradas**

kernel/param.h

```
#define NPROC        64  // maximum number of processes
```

kernel/proc.c

```
struct proc proc[NPROC];
```

Ejemplo PCB: struct proc (xv6)

Cada PCB tiene toda la información del proceso

kernel/proc.h

```
struct proc {

  ...

  enum procstate state;          // Process state

  int xstate;                    // Exit status to be returned to parent's wait

  int pid;                       // Process ID

  struct proc *parent;           // Parent process

  ...
```

Ejemplo PCB: struct proc (xv6)

kernel/proc.h (continuación)

```
  ...
  uint64 kstack;                // Virtual address of kernel stack
  pagetable_t pagetable;        // User page table
  struct file *ofile[NOFILE];   // Open files
  struct inode *cwd;            // Current directory
  char name[16];                // Process name (debugging)
  ...
};
```

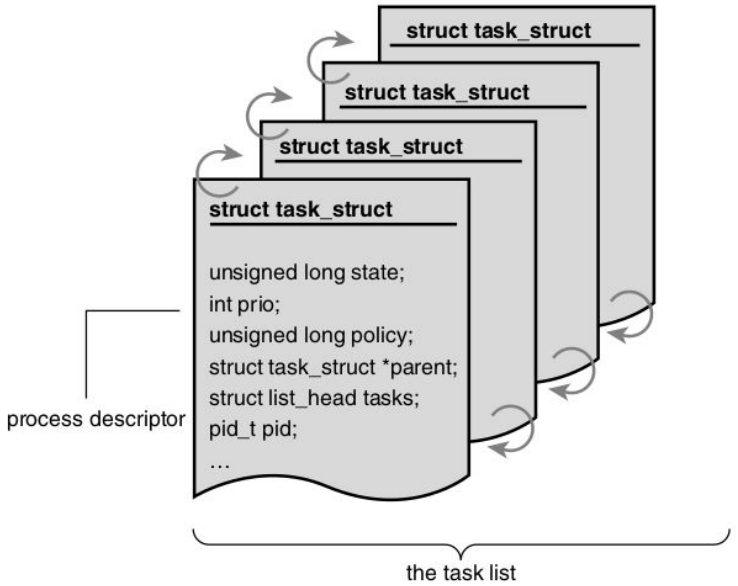
Ejemplo PCB: struct proc (xv6)

Recapitulando. Cada PCB en xv6 tiene:

- Process Id
- Nombre
- Process State
- Exit state
- Parent Process
- Open files
- Current directory
- Page Table
- Kernel Stack
- Estructuras auxiliares para: context switch, locking

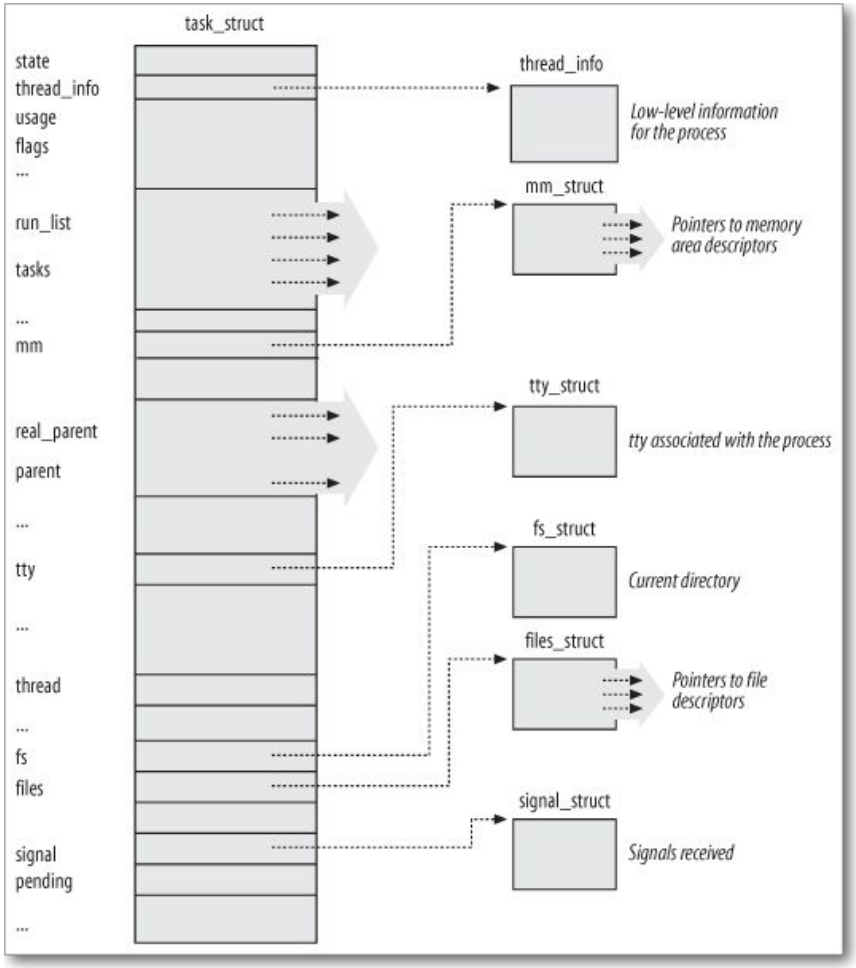
Ejemplo PCB: task_struct (Linux)

El kernel almacena la lista de procesos en una lista circular doblemente enlazada llamada **task list**.



Ejemplo PCB: task_struct (Linux)

La task_struct en sí es una estructura compleja en Linux.



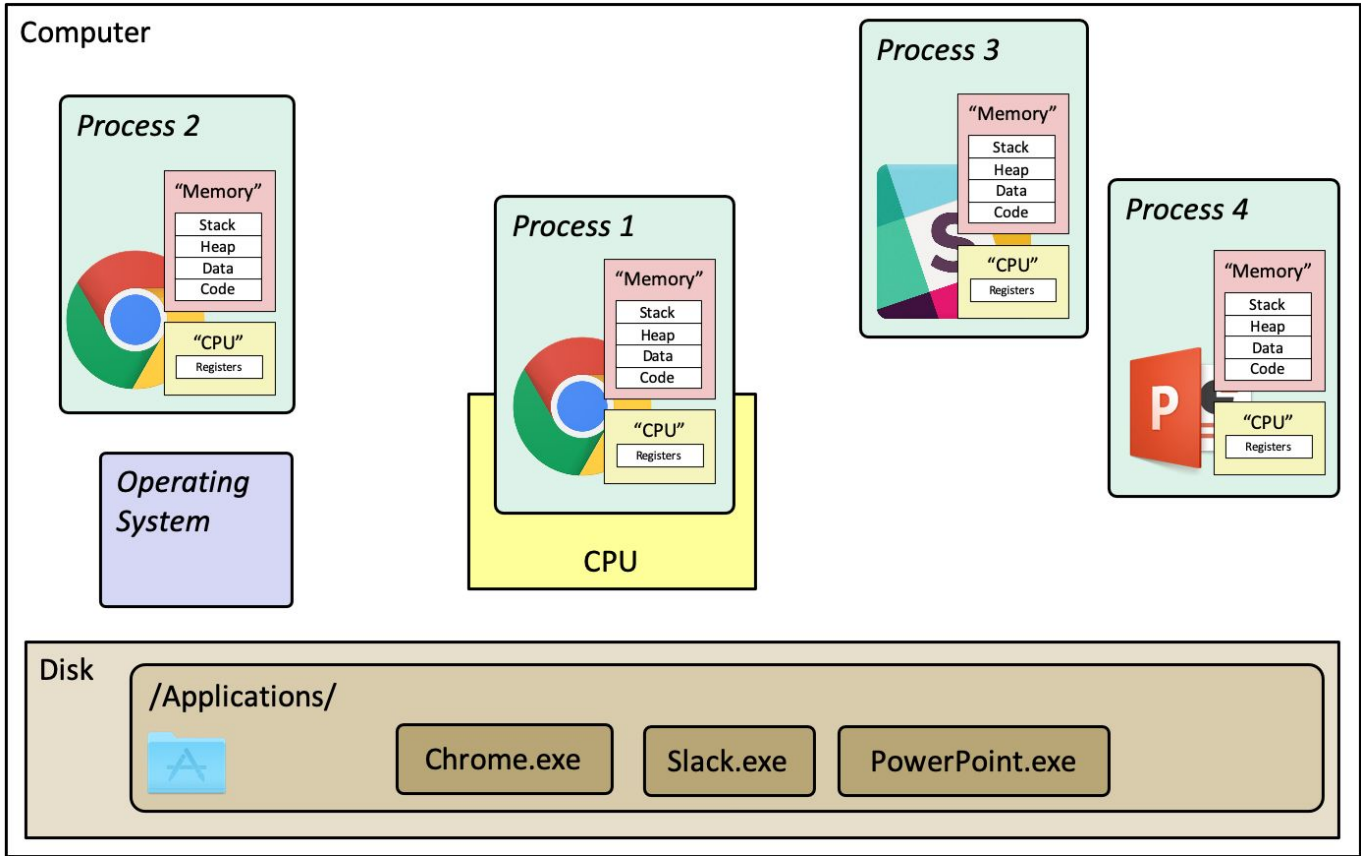
Anticipo: Context Switch

Spoiler alert! **El context switch**

¿Como se cambia de contexto, de un proceso a otro?

- Se cambia el User-level context (**memoria**)
- Se cambia el Register context (**cpu**)
- Se cambia el System-level context (**estructuras del kernel**)

Hágalo muy rápido y obtendrá la ilusión de concurrencia.
Cientos de procesos ejecutándose en un solo procesador!


Computer
Process 2
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
Operating System
Process 1
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
CPU
Process 3
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
Process 4
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
Disk
/Applications/
Chrome.exe
Slack.exe
PowerPoint.exe

Kernel stack

El Kernel usa funciones como cualquier programa, pero no puede usar el stack del user space para implementar sus funciones:

- No es seguro escribir datos del kernel en el user space
- Podria romper el user space (no hay garantías de como el user usa el stack)

El Kernel va a tener su propio stack, separado, en la zona de memoria del Kernel, protegida del usuario.

Kernel stack

¿Por qué un kernel stack por proceso (y no uno solo global)?

Porque cada proceso “dormido” podía estar haciendo cosas completamente diferentes en el espacio Kernel antes de dormirse:

Eg.

- Un proceso se bloqueo ejecutando read()
- Un proceso se bloqueo al hacer sleep()
- Un proceso se bloqueo mientras configuraba mas memoria virtual

Kernel stack

Cuando se hace un context-switch, pasan cosas:

- Se restauran los registros del proceso entrante (EAX, EBX, etc.)
- Se cambia toda la zona de usuario por la del proceso entrante (stack, heap, text, data)
- No se modifica la memoria del Kernel, porque el Kernel es común y compartido a todos los procesos. **Pero:**
- Se apunta el stack pointer al kernel stack del proceso entrante.