Sistemas Operativos
El Proceso

De Programa a Proceso

El Programa

1. Programador edita código fuente

2. El compilador compila el source code en una secuencia de instrucciones de máquina y datos llamada.

3. El compilador genera esa secuencia y posteriormente se guarda en disco: programa ejecutable.

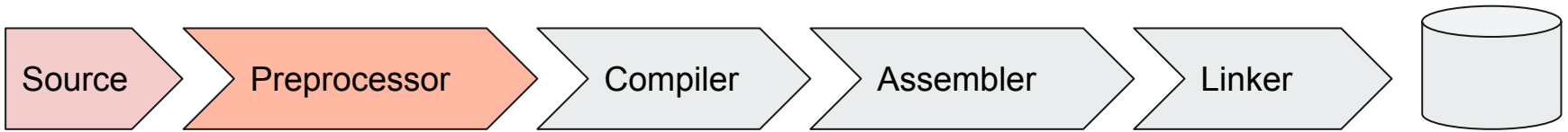
El Programa: Edicion

```
#include<stdio.h>
int main(){
        printf("hello, world\n");
}
```

#	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

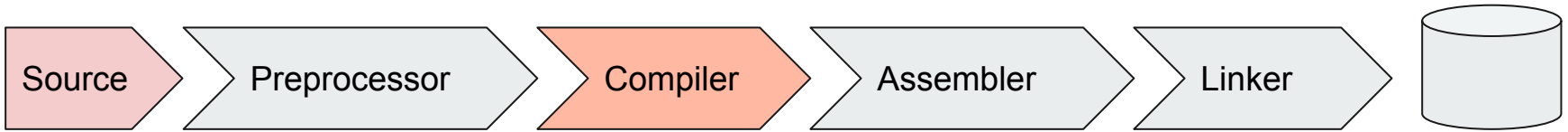
El Programa: Compilación

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 extinción .i



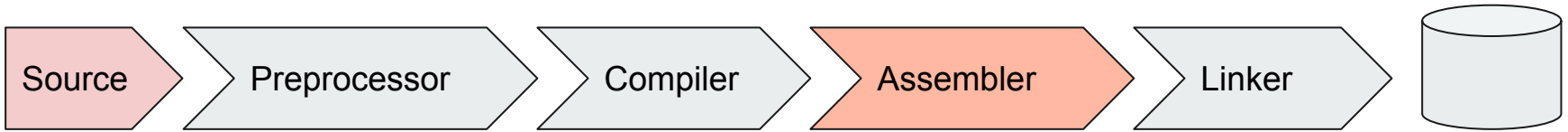
El Programa: Compilación

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



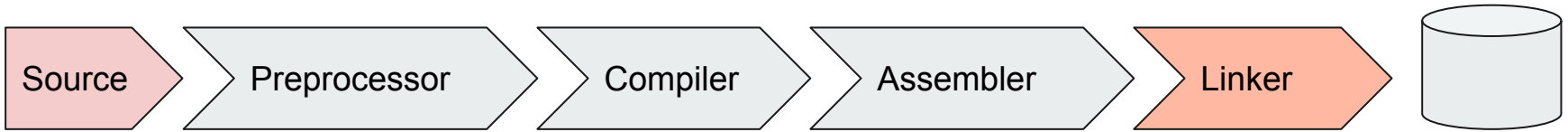
El Programa: Compilación

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

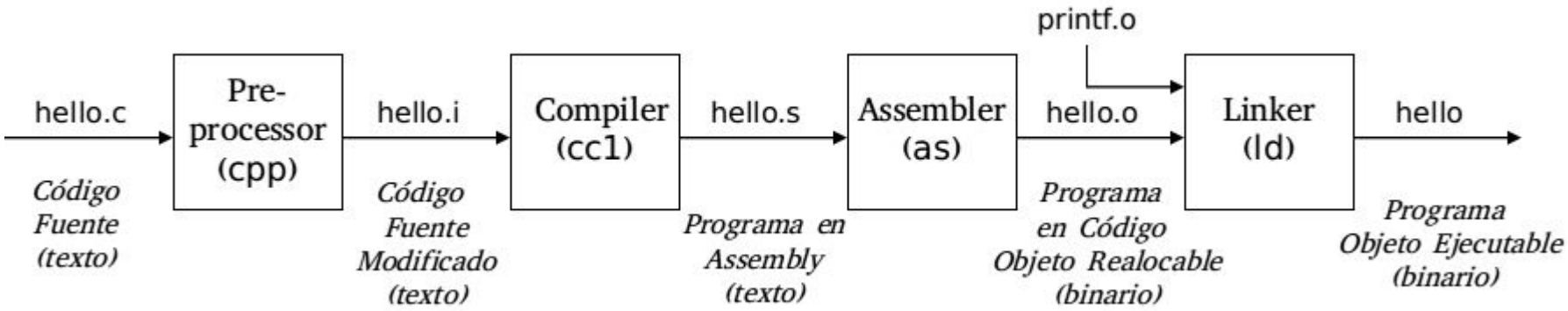


El Programa: Compilación

La fase de link edicion. Generalmente los programas escritos en lenguaje C hacen uso de funciones que forman parte de la biblioteca estándar de C que es provista por cualquier compilador de ese lenguaje. Por ejemplo la función printf(), la misma se encuentra en un archivo objeto pre compilado que tiene que ser mezclado con el programa que se está compilando, para ello el linker realiza esta tarea teniendo como resultado un archivo objeto ejecutable.



El Programa: Compilación



El Programa: Formato Ejecutable

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

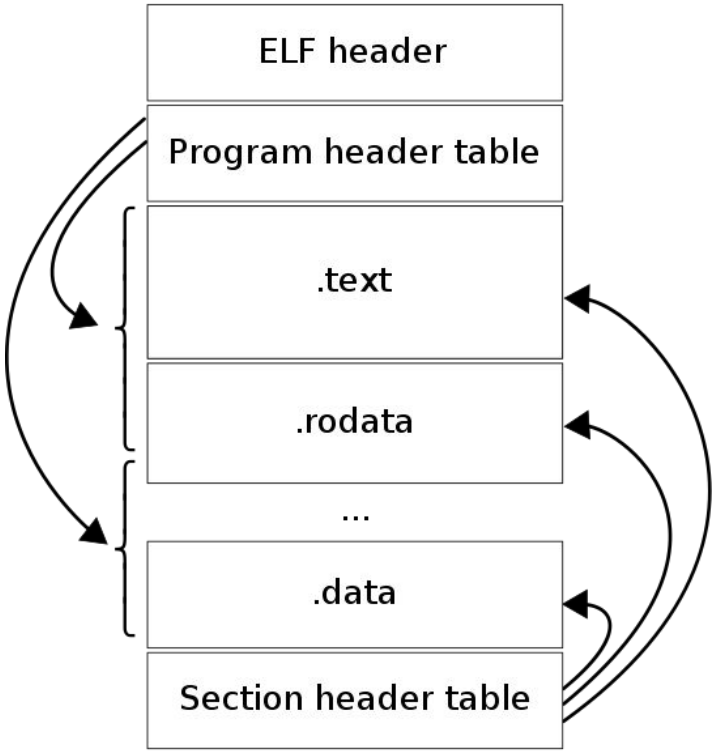
- **Instrucciones de Lenguaje de Máquina**: Almacena el código del algoritmo del programa.
- **Dirección del Punto de Entrada del Programa**: Identifica la dirección de la instrucción con la cual la ejecución del programa debe iniciar.
- **Datos**: El programa contiene valores de los datos con los cuales se deben inicializar variables, valores de constantes y de literales utilizadas en el programa.
- **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**: 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 Programa en Linux: ELF

ELF: Extensible Linking Format

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

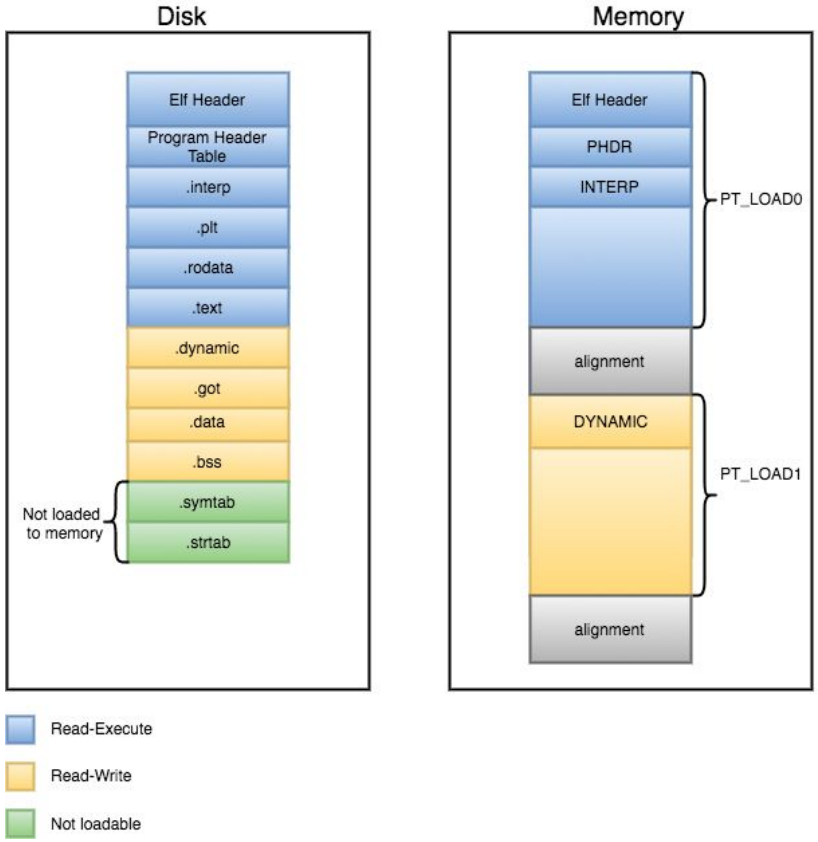
https://greek0.net/elf.html



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;
};
```



Un Programa en Linux: ELF

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

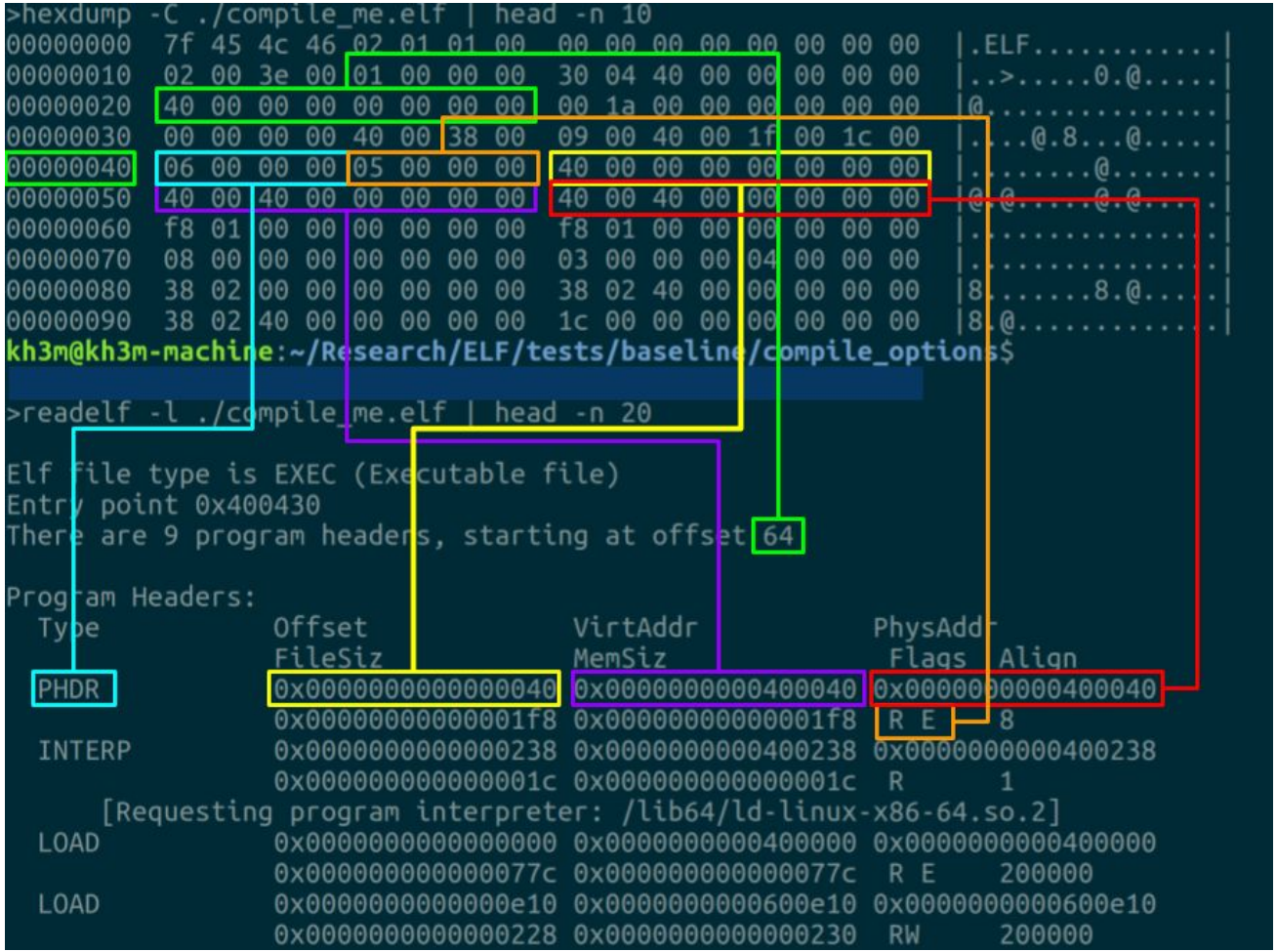
Ver el SHT (Section Header Table) readelf -S <archivo>

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

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

“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]

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]

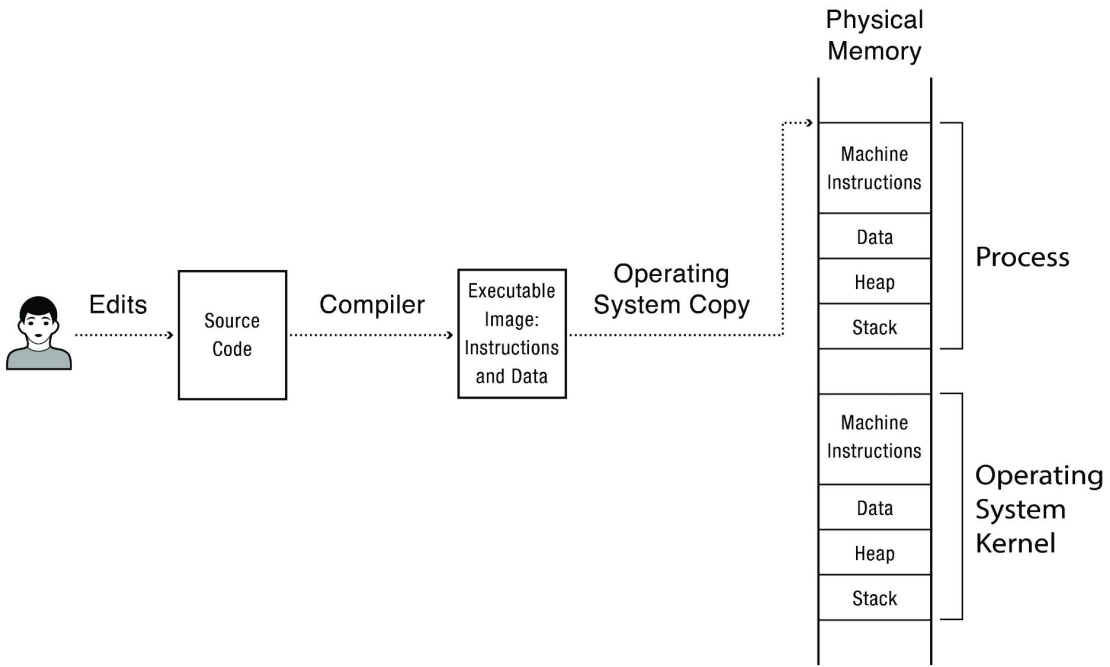
El Proceso: Virtualización

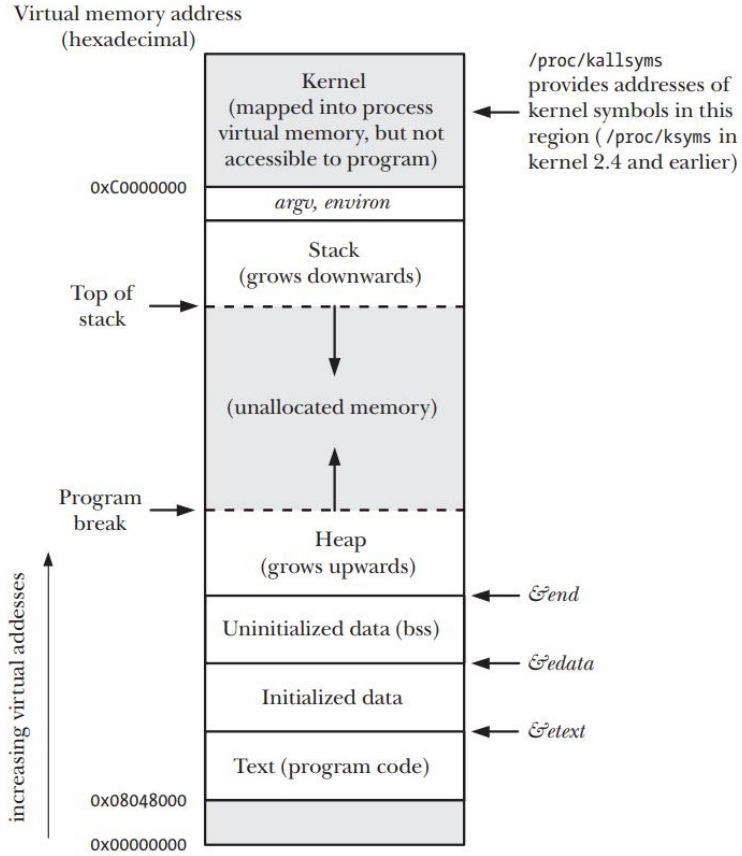
- Virtualización de Memoria
- Virtualización de Procesamiento

El Pr

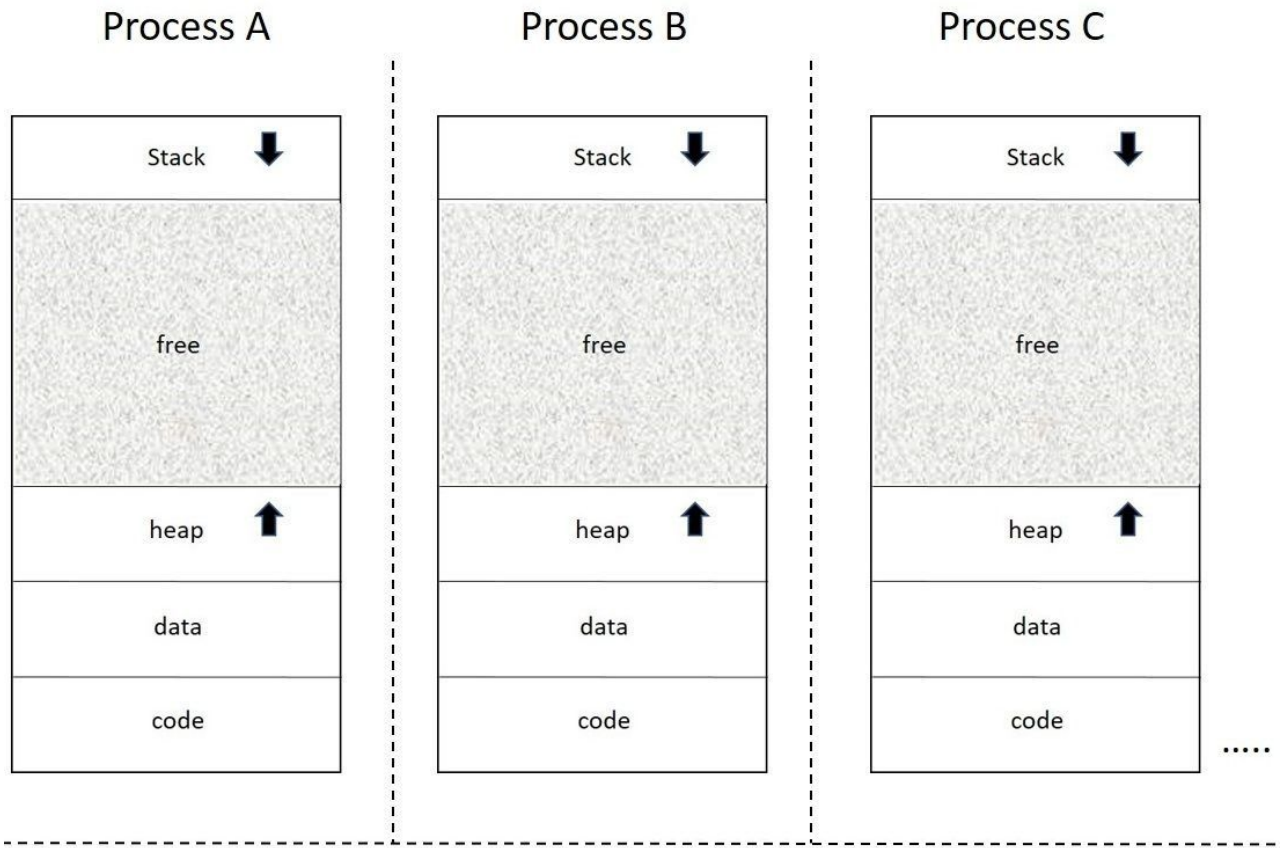
La vir este tiene
toda l no si este
estuvi). Todos
los pr

Text: I
Data:
Heap:
Stack:





El Proceso: Espacio de Direcciones



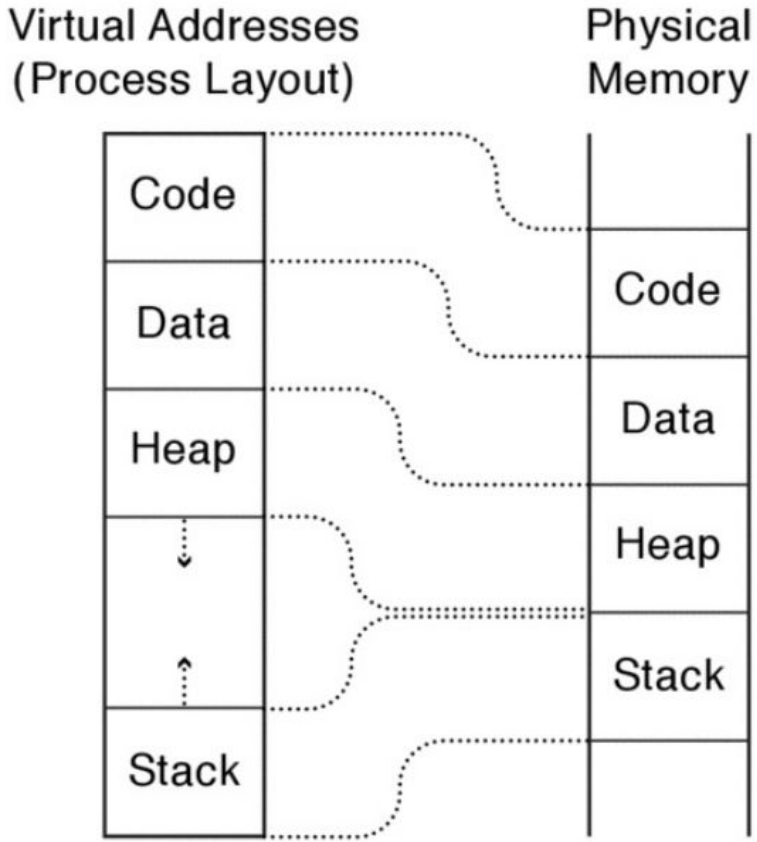
Todos los procesos tienen la misma estructura del Address Space

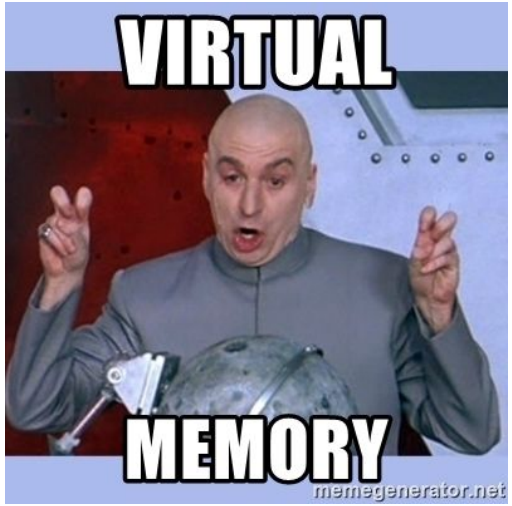
Virtualización de Memoria

- Uno de estos mecanismos es denominado Memoria Virtual, la memoria virtual es una abstracción por al 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 si mismo, si bien obviamente esto en la realidad no sucede. El hardware traduce la dirección virtual a una dirección física de memoria.

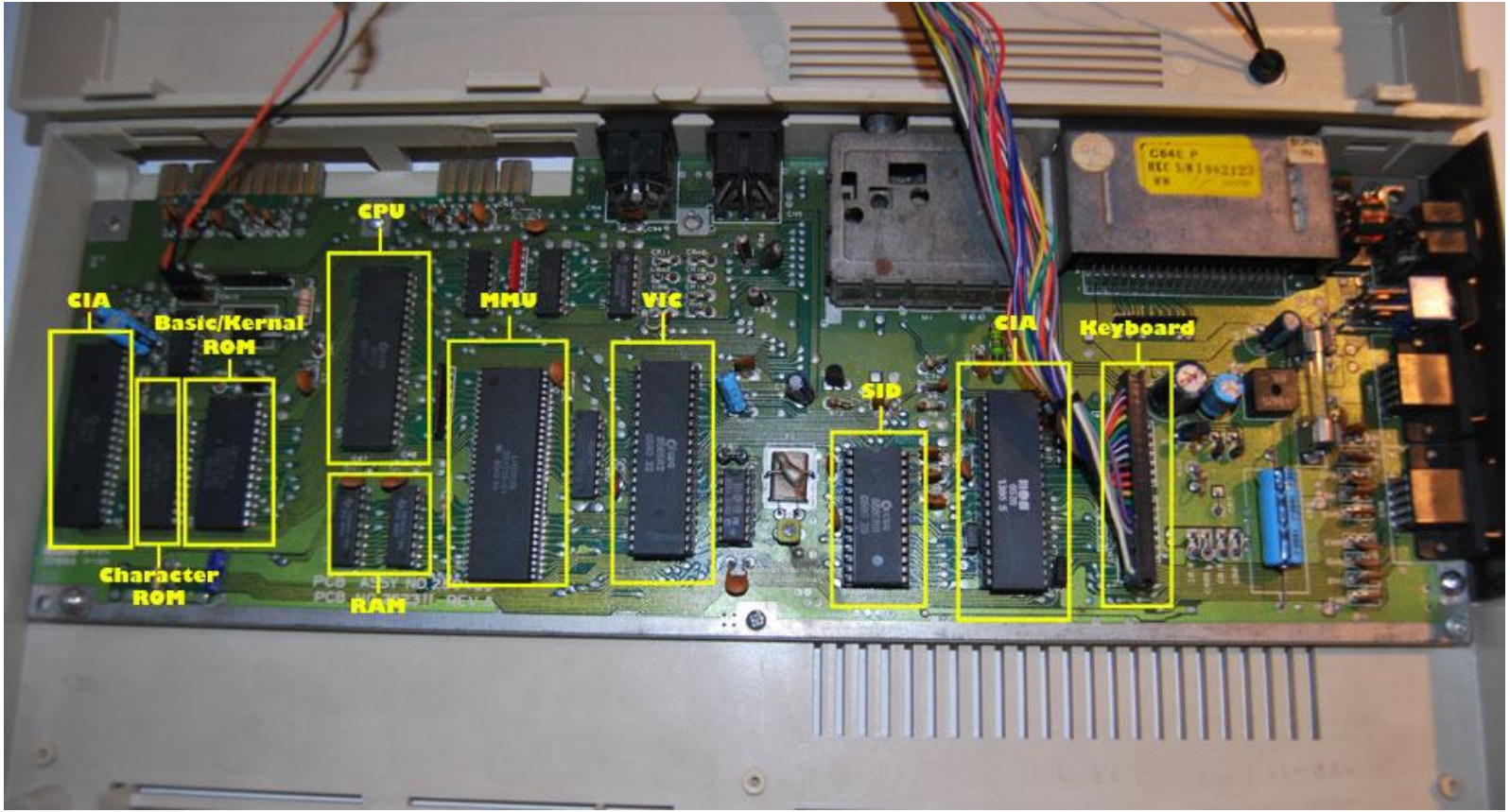




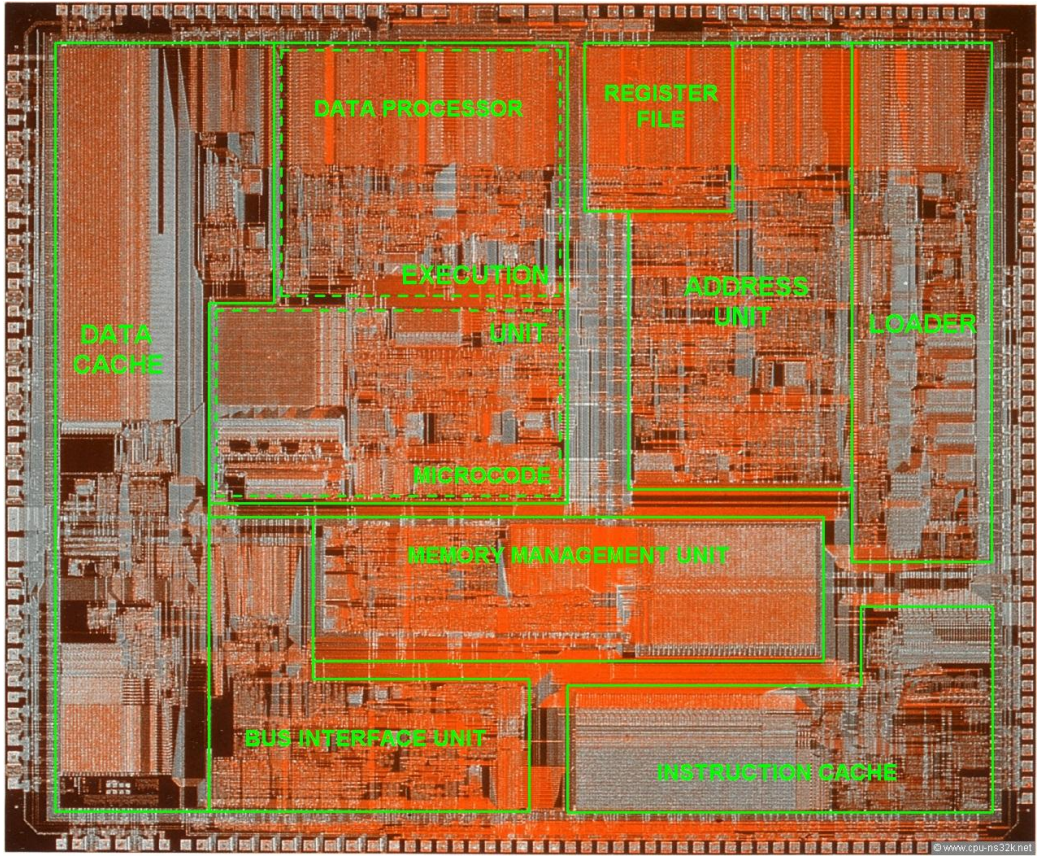
Virtualización de Memoria

Traducción de Direcciones

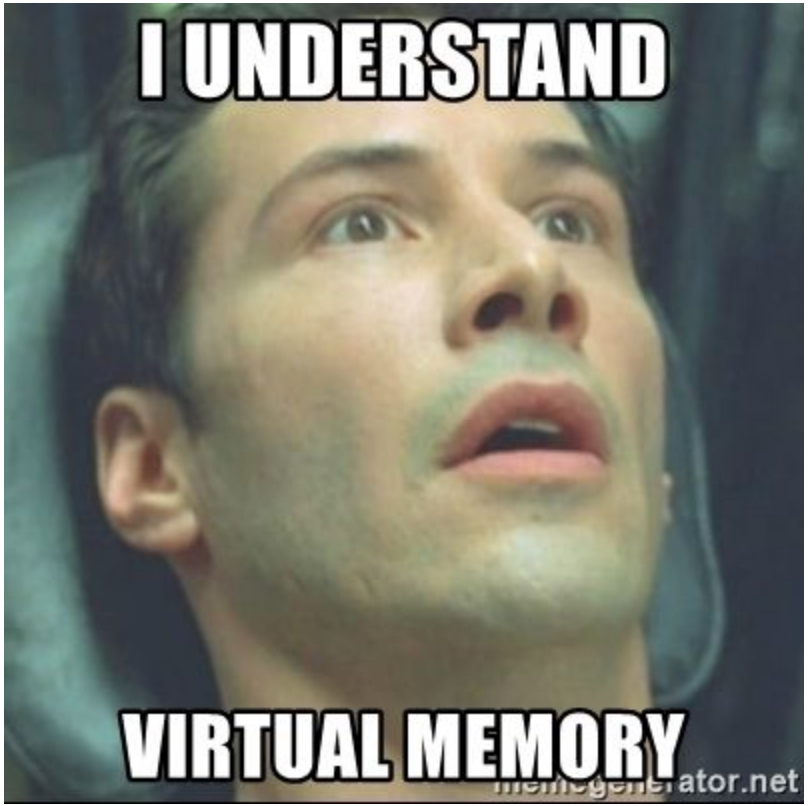
Se traduce una Dirección Virtual (emitida por la CPU) en una Dirección Física (la memoria). Este mapeo se realiza por hardware, más específicamente por Memory Management Unit (MMU).



Commodore 64

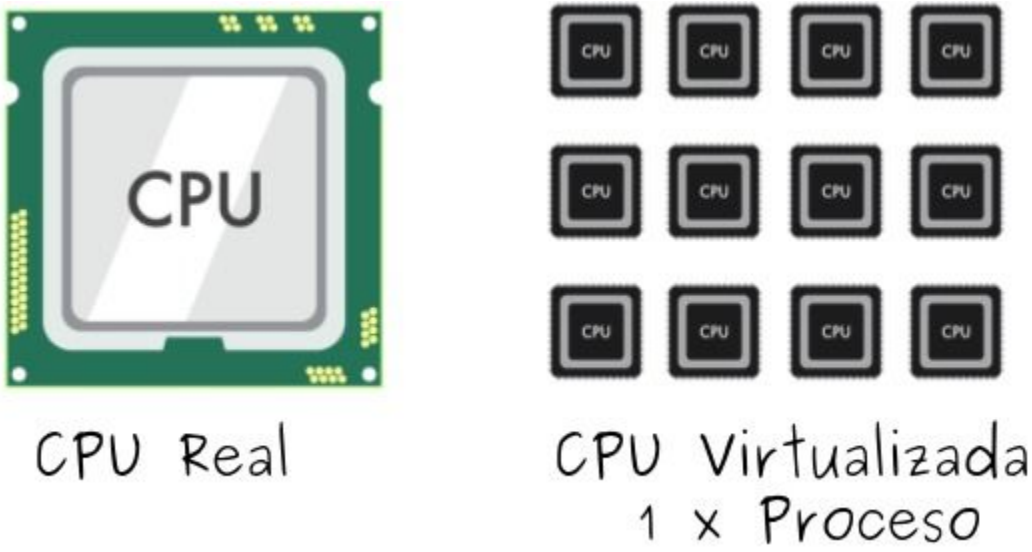


La MMU Dentro del Procesador

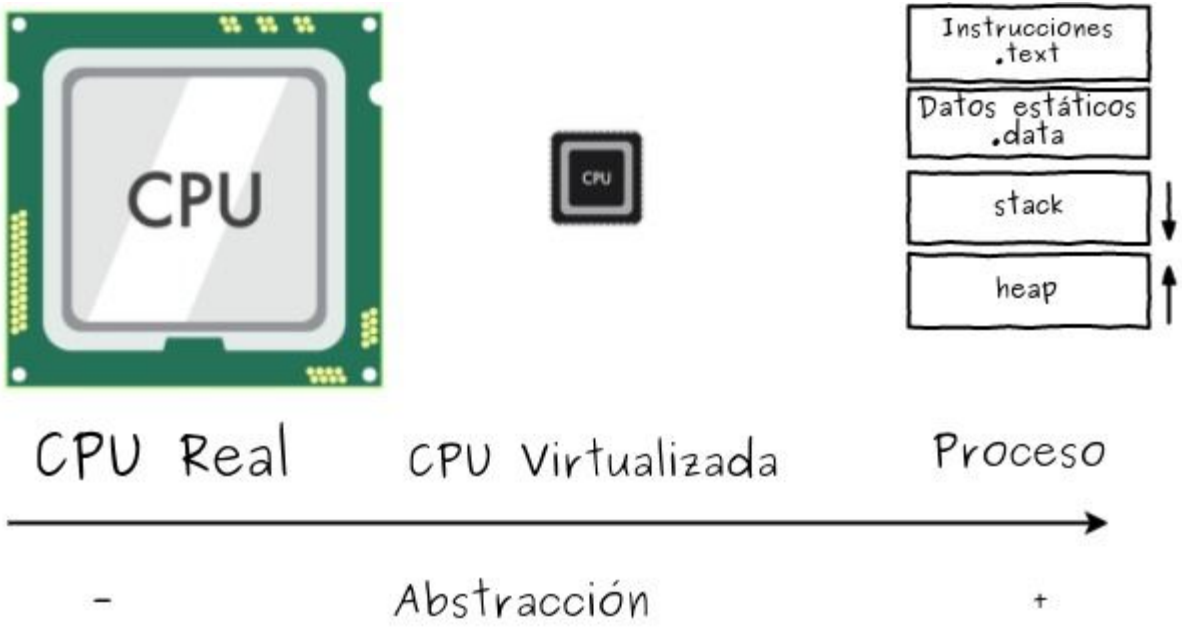

I UNDERSTAND
VIRTUAL MEMORY

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.**

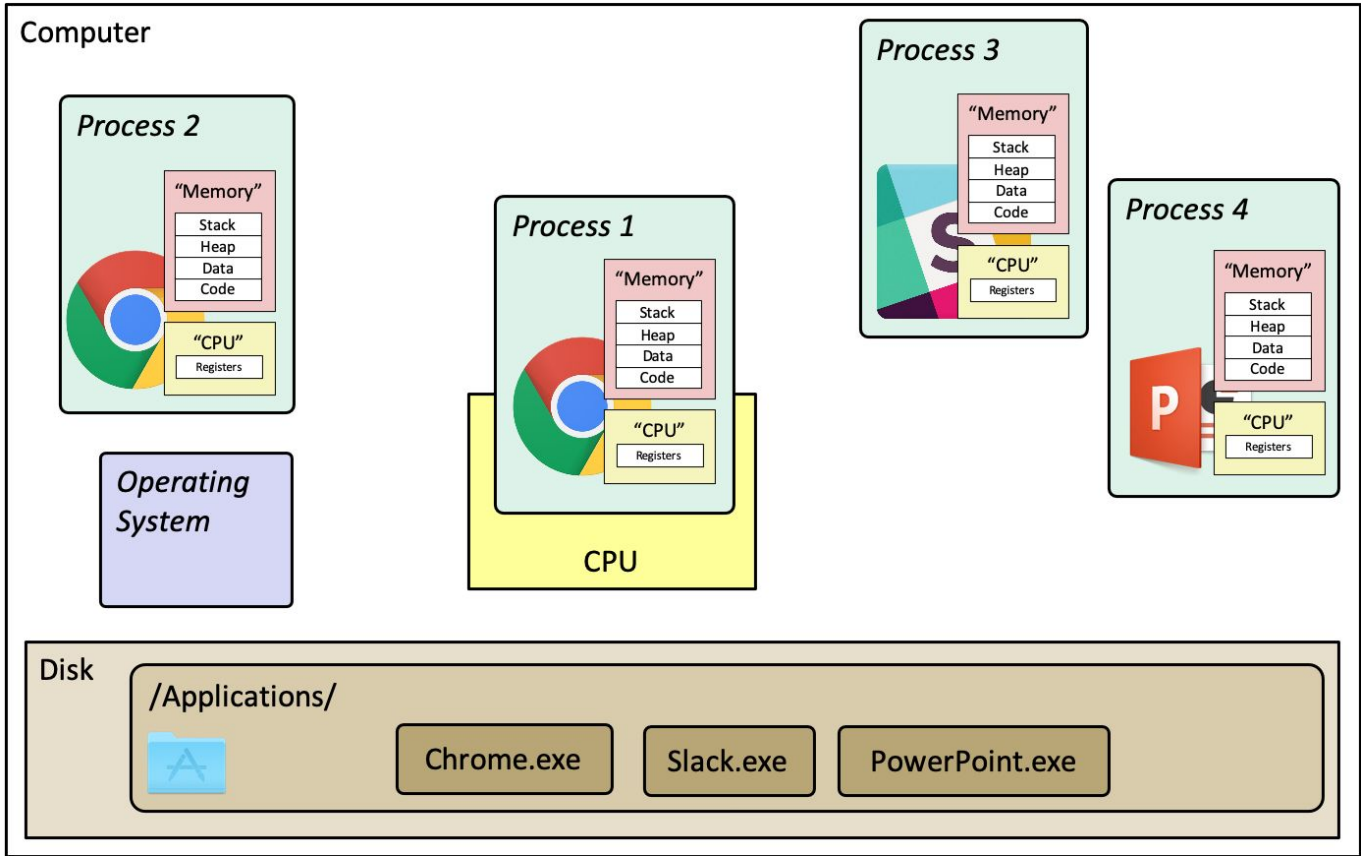


El SO crea esta ilusión mediante la virtualización de la CPU a través del kernel.


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

Proceso: El Contexto

- El **contexto** de un proceso **es la información necesaria para describir al proceso**.
- Cada proceso posee un contexto.
- Según bach:" el contexto de un proceso comprende, el contenido del address space, el contenido de los registros de hardware y las estructuras de datos que pertenecen al kernel relacionadas con el proceso" .
- El contexto de un proceso es la unión de:
    - User-level context
    - Register context
    - System-level context

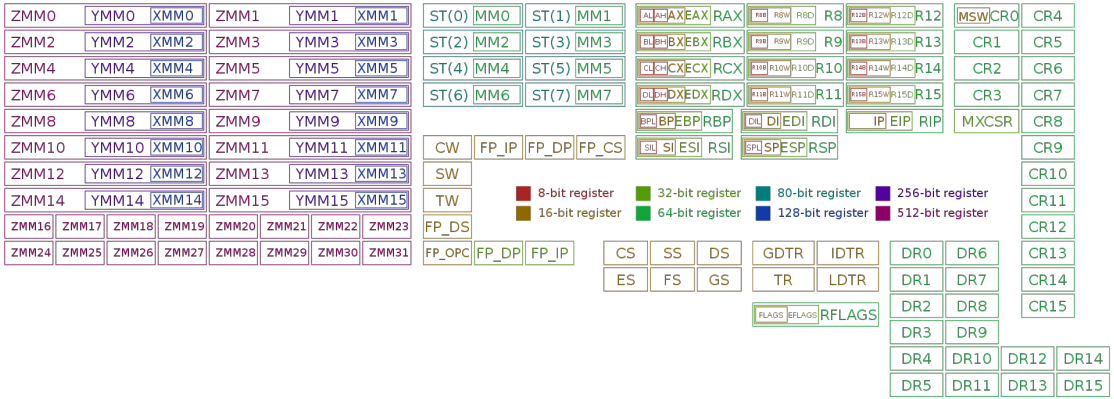

Computer
Process 2
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
Operating System
Process 1
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
CPU
Process 3
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
Process 4
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
Disk
/Applications/
Chrome.exe
Slack.exe
PowerPoint.exe

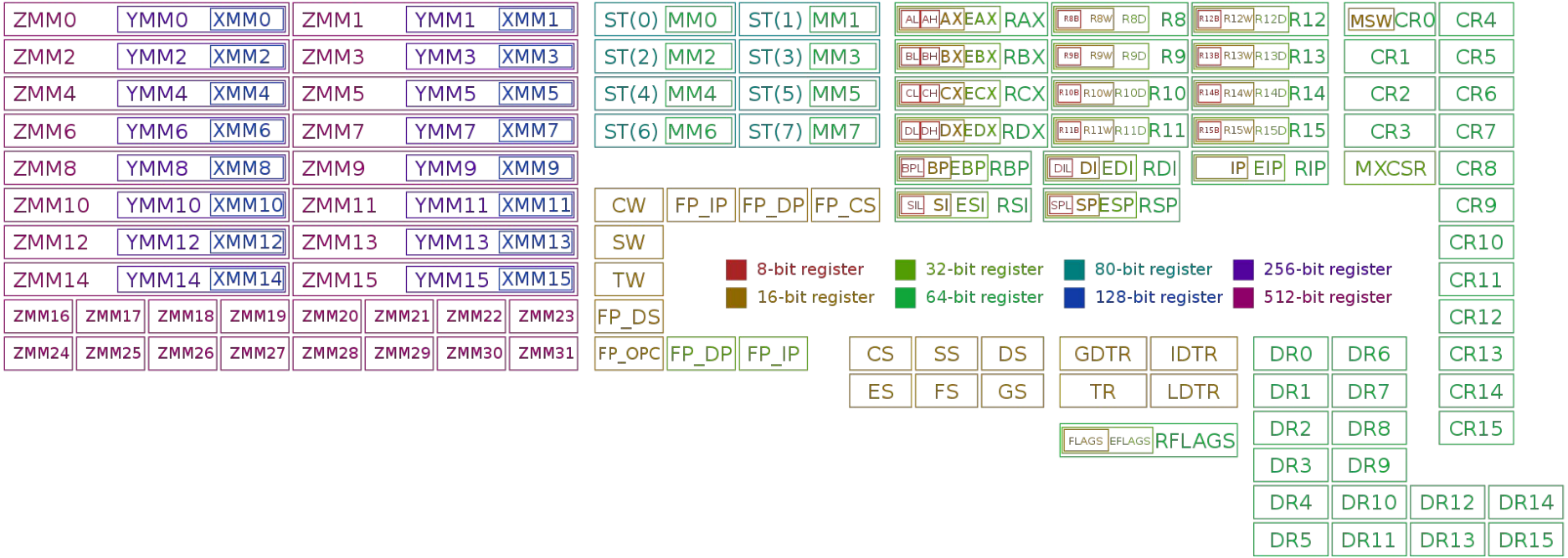
Proceso: El Contexto User-level

- Consiste en las Secciones
  - Text
  - Data
  - Stack
  - Heap

Proceso: El Contexto Register

- Program Counter Register
- Processor Status Register
- Stack Pointer Register
- General Purpose Registers
  - En x86





Proceso: El Contexto Register

https://upload.wikimedia.org/wikipedia/commons/thumb/1/15/Table_of_x86_Registers_svg.svg/1920px-Table_of_x86_Registers_svg.svg.png

Proceso: El Contexto System-level

- La entrada en la **Process Table Entry**
- La **u area**
- La Process Region Entry, Region Table y Page Table que definen el mapeo de la memoria virtual vs memoria física del proceso.
- El Stack del Kernel, contiene los stack frames de las llamada al kernel hechas por el proceso.

Proceso: El Contexto System-level

- La entrada en la **Process Table Entry**
- La **u area**

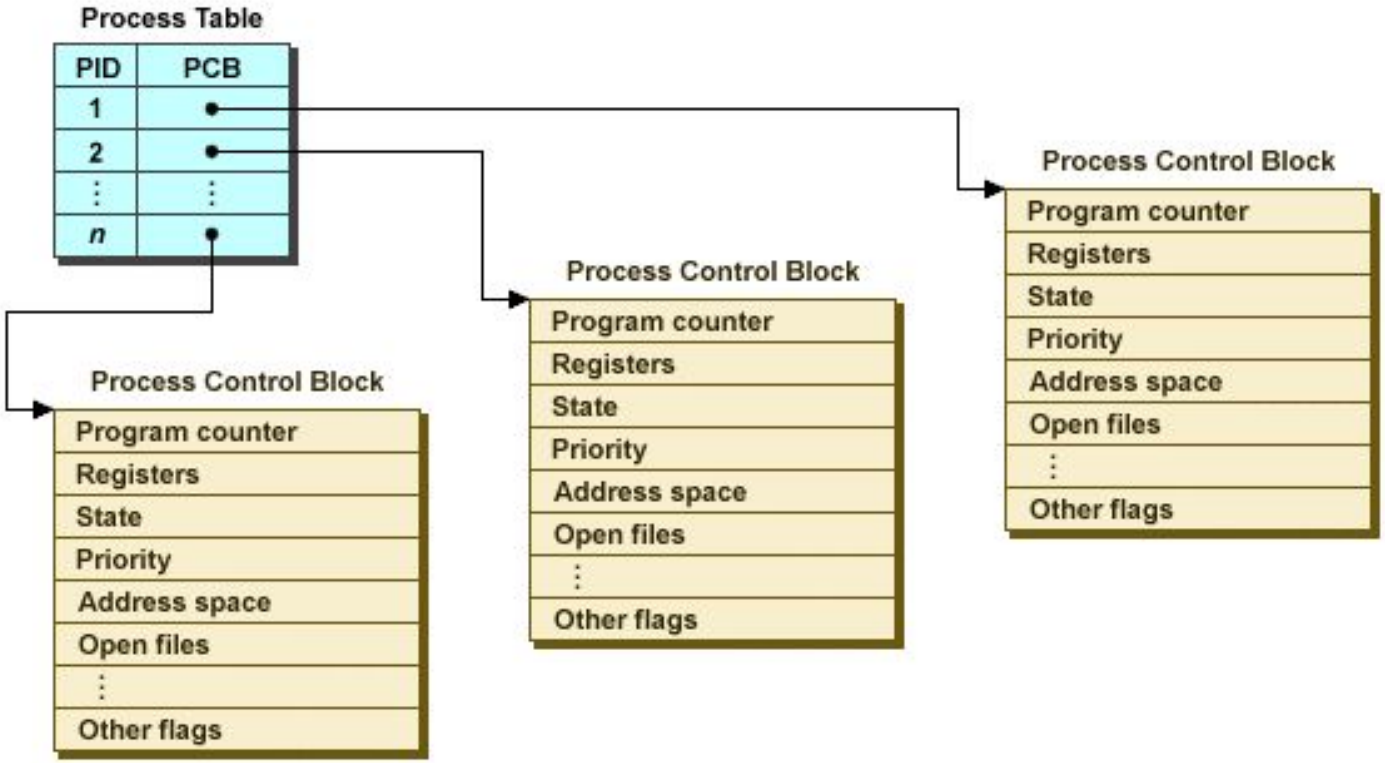
Son dos estructuras que pertenecen al Kernel:

- la **process table** contiene información que siempre tiene que estar disponible para el kernel.

- La **u área** contiene campos que sólo deben estar disponibles cuando el proceso está corriendo.

Proceso: El Contexto System-level: Process Table

- dentificación: 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 Kerner 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.



Proceso: El Contexto System-level: Process Table

Contenido de la user area <arch/x86/include/asm/user.h>:

- Un puntero a la proc structure del proceso
- El UID y GID real
- Argumentos para, y valores de retorno o errores hacia, la system call actual
- Manejadores de Señales
- Información sobre las áreas de memoria text,data, stack, heap y otra información.
- La tabla de descriptores de archivos abiertos (Open File descriptor Table).
- Un puntero al directorio actual
- Datos estadísticos del uso de la cpu, información de perfilado, uso de disco y límites de recursos.

Proceso: El Contexto System-level: U Area


SEEMS A LITTLE CRAZY!

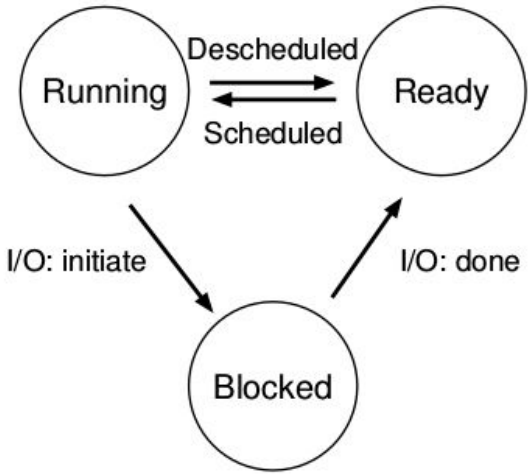
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process

struct context
{
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                  // Start of process memory
    uint sz;                    // Size of process memory
    char *kstack;               // Bottom of kernel stack
                            // for this process
    enum proc_state state;      // Process state
    int pid;                    // Process ID
    struct proc *parent;        // Parent process
    void *chan;                 // If non-zero, sleeping on chan
    int killed;                 // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;          // Current directory
    struct context context;     // Switch here to run process
    struct trapframe *tf;       // Trap frame for the
                            // current interrupt
};
```

Estados de un Proceso 1

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.

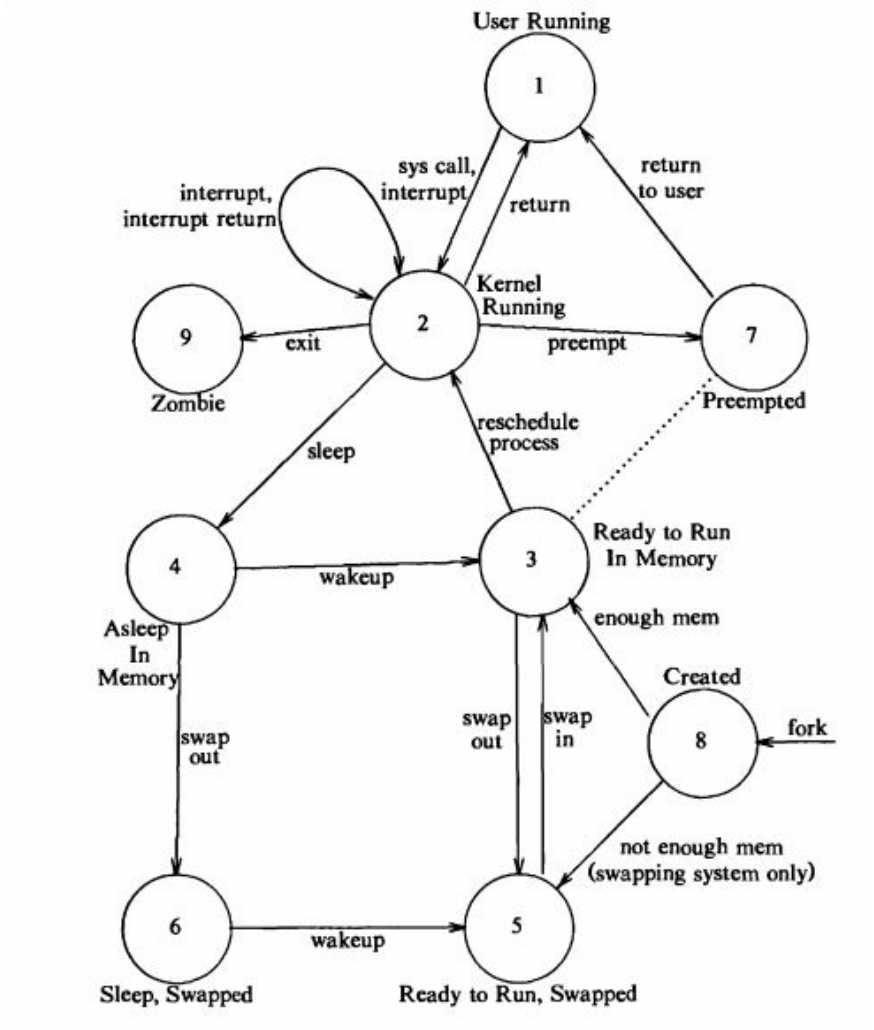
Bloqueado (Blocked): en este estado el proceso ha ejecutado algún tipo de operación que hace que éste no esté listo para ejecutarse hasta que algún evento suceda.



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.

Estados de un Proceso 2: System V

1-Running User Mode
2-Running Kernel Mode
3-Ready to Run on Memory
4-Asleep In Memory
5-Ready to Run but swapped
6-Asleep Swapped
7-Preempt
8-Created
9-Zombie



El API resumida

fork(): Crea un proceso y devuelve su id.
exit(): Termina el proceso actual.
wait(): Espera por un proceso hijo.
kill(pid): Termina el proceso cuyo pid es el parámetro.
getpid(): Devuelve el pid del proceso actual.
exec(filename, argv): Carga un archivo y lo ejecuta.
sbrk(n): Crece la memoria del proceso en n bytes.

Creación de un Proceso

La única forma de que un usuario cree un proceso en el sistema operativo UNIX es llamando a la system call fork.
El proceso que invoca a fork es llamado proceso padre, el nuevo proceso creado es llamado hijo.

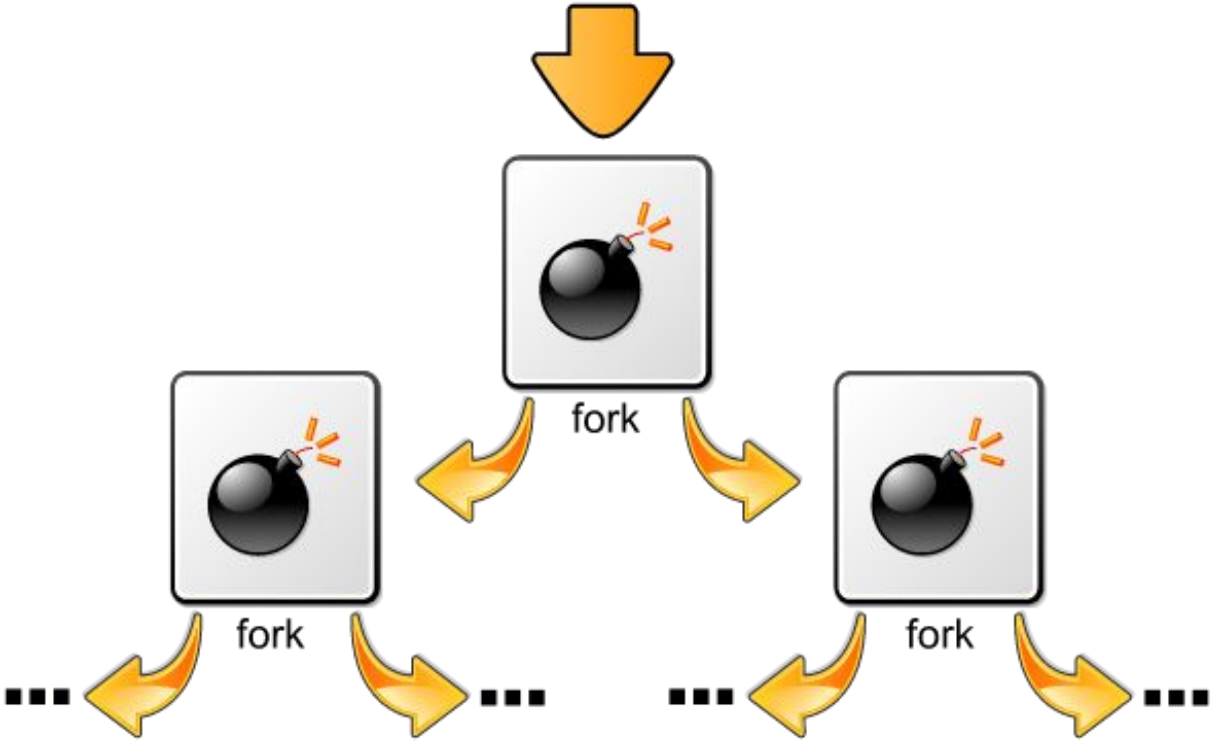
¿Que hace fork?:
Crea y asigna una nueva entrada en la Process Table para el nuevo proceso.
Asigna un número de ID único al proceso hijo.
Crea una copia lógica del contexto del proceso padre, algunas de esas partes pueden ser compartidas como la sección text
Realiza ciertas operaciones de I/O.
Devuelve el número de ID del hijo al proceso padre, y un 0 al proceso hijo

La implementación de esta system call no es para nada trivial ya que cuando el proceso hijo inicia inicia a ejecutarse parece hacerlo casi en el aire:

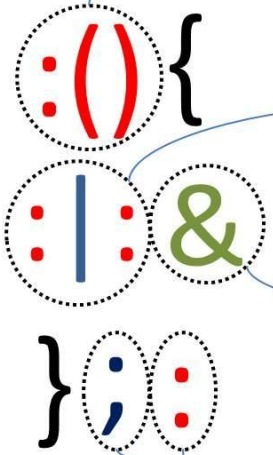
- chequear que haya recursos en el kernel;
- obtener una entrada libre de la Process Table, como un PID único;
- chequear que el usuario no esté ejecutando demasiados procesos;
- macar al proceso hijo en estado “siendo creado”;
- copiar los datos de la entrada en la Process Table del padre a la del hijo;
- incrementar el contador del current directoty inode;
- incrementar el contador de archivos abiertos en la File Table;
- hacer una copia del contexto del padre en memoria;
- crear un contexto a nivel sistema falso para el hijo;
    - el contexto falso contiene datos para que el hijo se reconozca a sí mismo
    - y para que tenga un punto de inicio cuando el planificador lo haga ejecutarse;

```
if(el proceso en ejecución es el padre){
cambiar el estado del hijo a "ready to run";
return ( ID del hijo);}
else /* se esta ejecutando el hijo */{
inicializar algunas cosas;
return 0;}
```

¿Qué hace fork(), el algoritmo?



Forkbomb



It is a function name. It accepts no arguments at all.

Function ':' calls itself recursively and pipes the output to another call. This way its calling itself twice.

Puts the function call in the background so child cannot die at all and start eating system resources.

Terminates the function definition.

Calls the function (Sets the fork bomb).

Forkbomb

System Call _exit()

Generalmente un proceso tiene dos formas de terminar:

La anormal: a través de recibir una señal cuya acción por defecto es terminar el programa.

La normal: a través de invocar a la system call exit()

Esta system call generalmente no es utilizada, en su lugar se utiliza exit() de la biblioteca estándar de C.

System Call _exit()

Que hace exit() , el algoritmo:

- Ignora todas las signals.
- Cierra todos los archivos abiertos
- En consecuencia se liberan todos los locks mantenidos por este proceso sobre esos archivos
- Libera el directorio actual
- Los segmentos de memoria compartida del procesos se separan
- los contadores de los semáforos son actualizados
- Libera todas las secciones y memoria asociada al proceso
- Registra información sobre el proceso (accounting record)
- Pone el estado del proceso en “zombie”
- Le asigna el parent PID de los procesos hijos al PID de init
- le manda una signal o señal de muerte al proceso padre
- context switch

System Call _exit()


```
#include <stdlib.h>
void exit(int status);
```

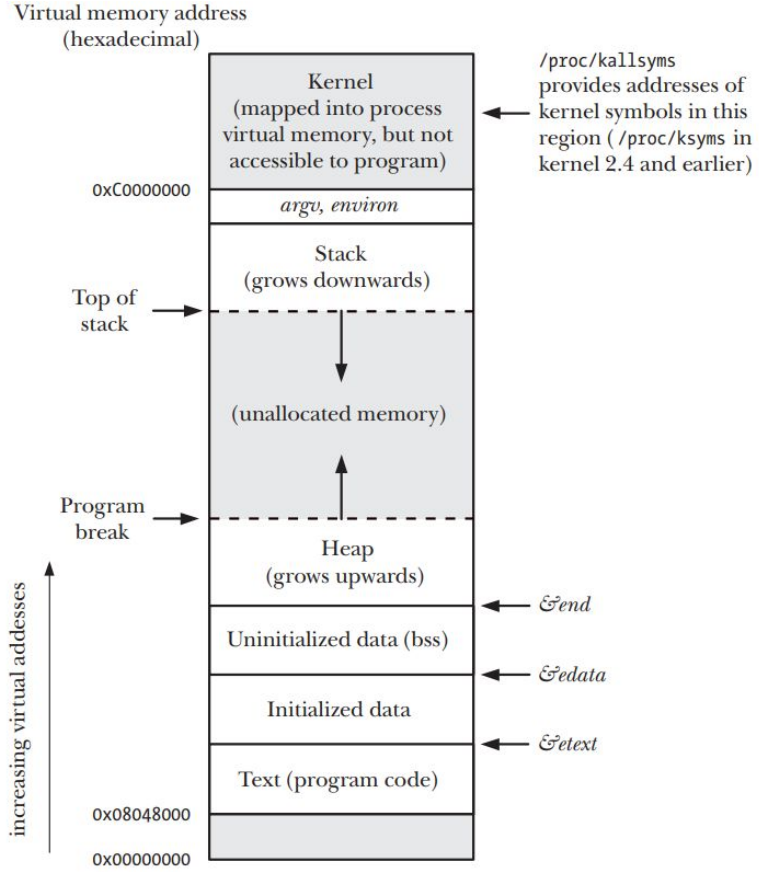
Esta función de la biblioteca estándar de C :

- llama a los Exit Handler que son dos funciones llamadas on_exit() y atexit()
- los streams de stdio son flusheados (buffer−>disco)
- se llama a la system call _exit().

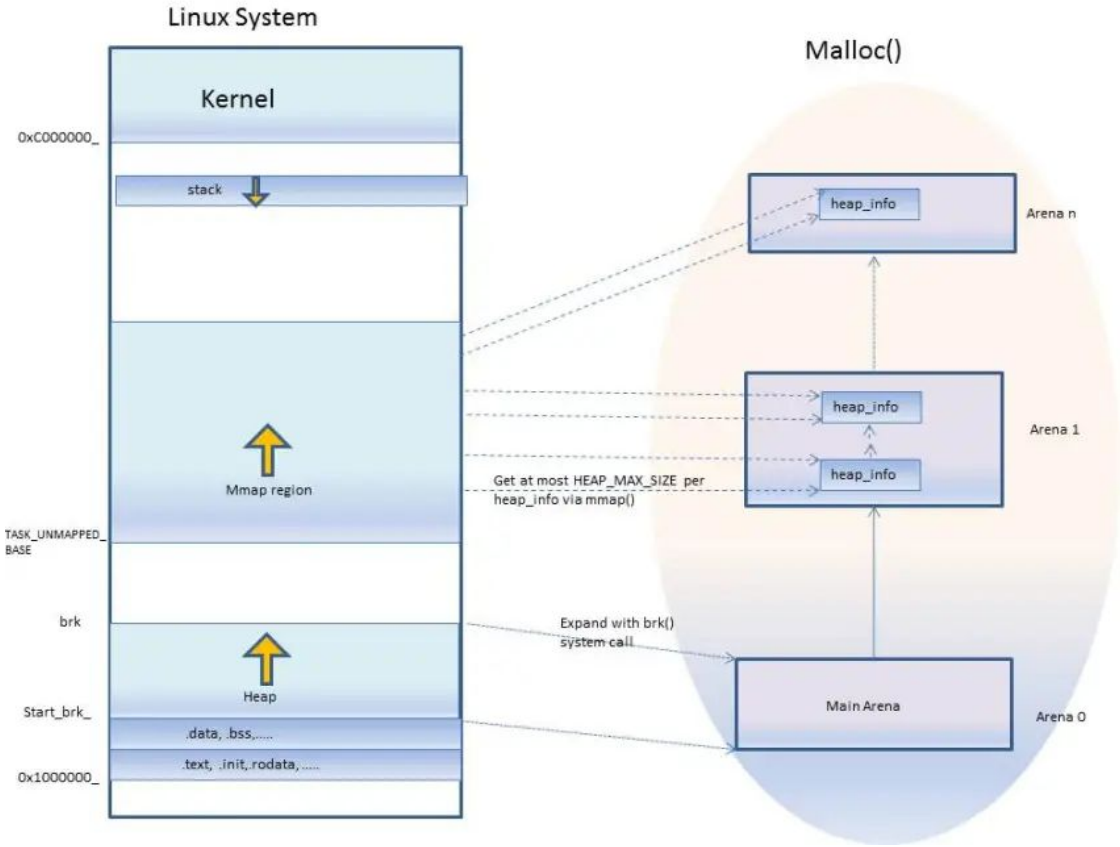
Inicialmente el break del programa está ubicado justo en el final de datos no inicializados.

Despues 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.



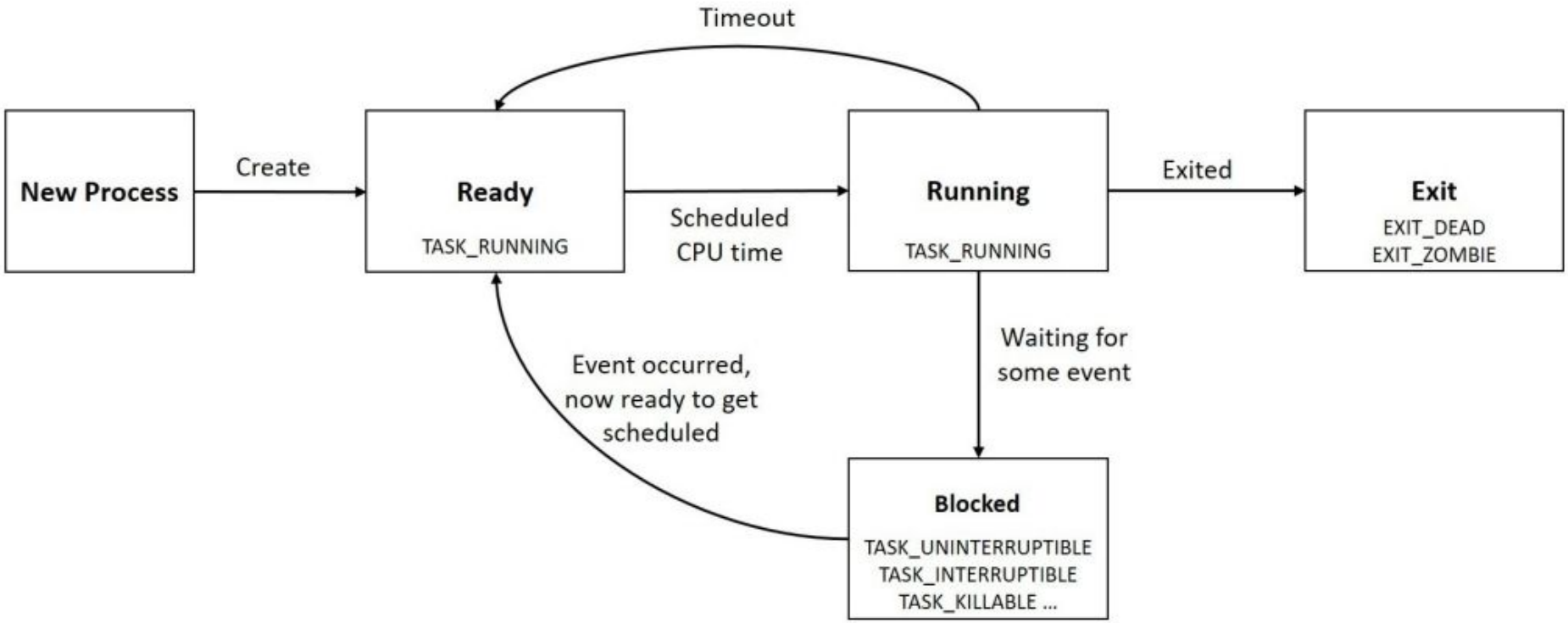
System call brk()



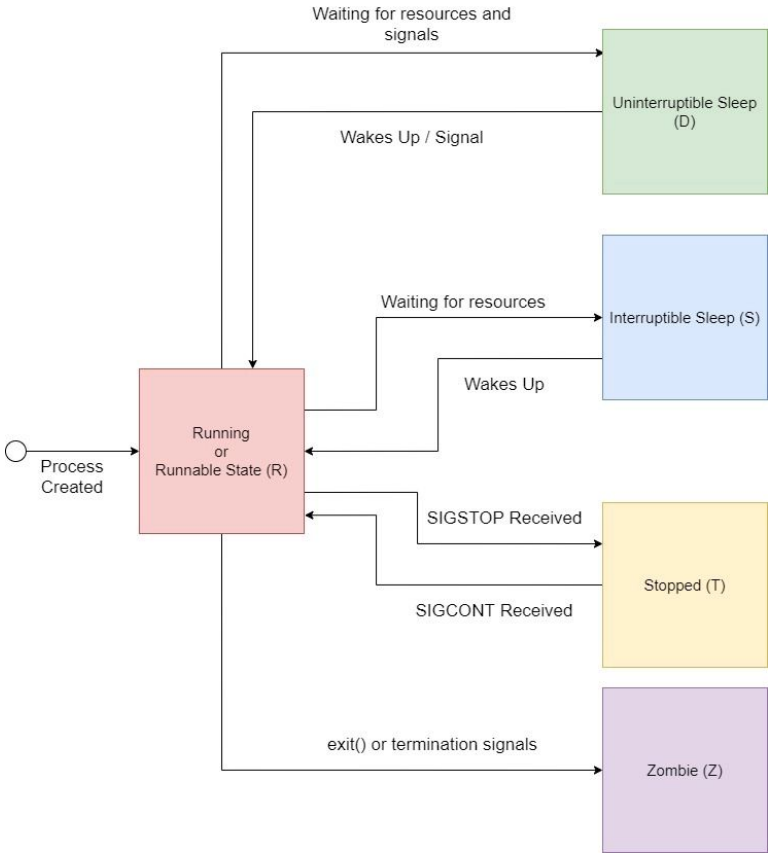
Brk vs. malloc() y free()

TASK_RUNNING : El proceso está o ejecutándose o peleando por CPU en la cola de run del planificador.
TASK_INTERRUPTIBLE : El proceso se encuentra en un estado de espera interrumpible; este queda en este estado hasta que la condición de espera eventualmente sea verdadera, por ejemplo un dispositivo de I/O está listo para ser utilizado, comienza de su time slice, etc. Mientras el proceso está en este estado, cualquier señal (signal) generada para el proceso es entregada al mismo, causando que este se despierte antes que la condición de espera se cumpla.
TASK_KILLABLE: este estado es similar al TASK_INTERRUPTIBLE, con la excepción que las interrupciones pueden ocurrir en fatal signals.
TASK_UNINTERRUTPIBLE (2): El proceso está en un estado de interrupción similar al anterior pero no podrá ser despertado por las señales que le lleguen. Este estado es raramente utilizado.
TASK_ STOPPED : El proceso recibió una señal de STOP. Volverá a running cuando reciba la señal para continuar (SIGCONT).
TASK_TRACED : Un proceso se dice que esta en estado de trace, cuando está siendo revisado probablemente por un debbuger.
EXIT_ZOMBIE): El proceso está terminado, pero sus recursos aún no han sido solicitados.
E**XIT_DEAD** : El proceso Hijo ha terminado y todos los recursos que este mantenía para sí se han liberado, el padre posteriormente obtiene el estado de salida del hijo usando wait.

Procesos en Linux



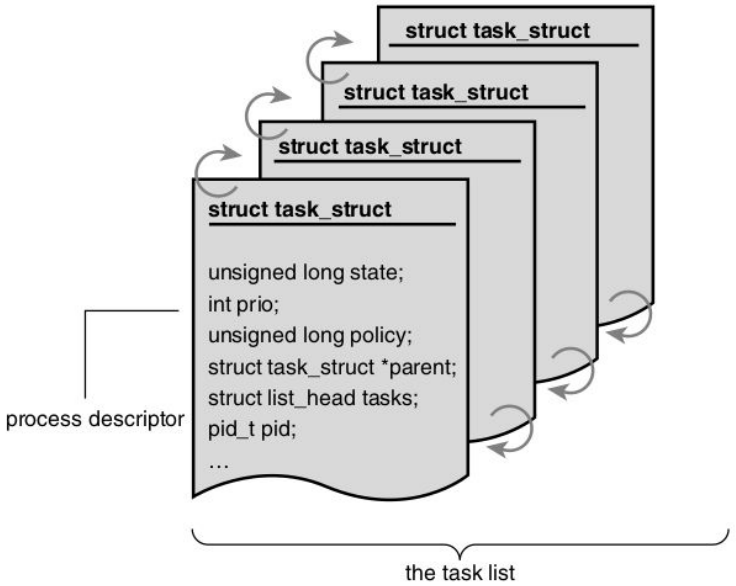
Estados de un proceso en linux

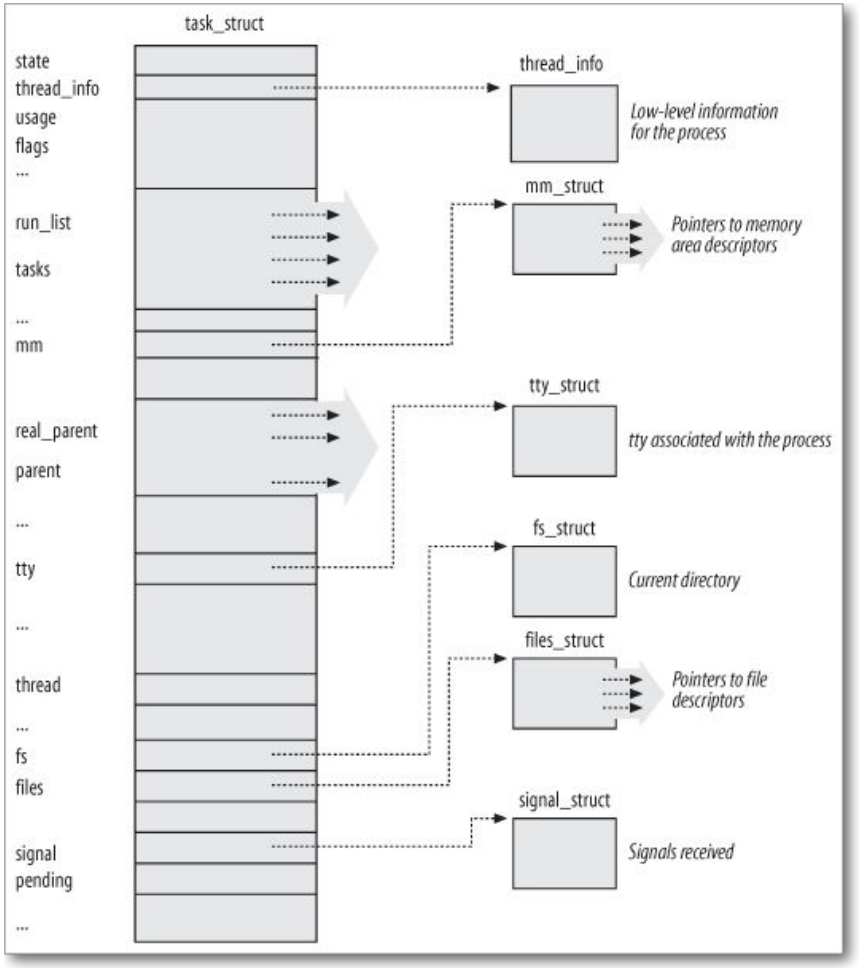


Estados de un proceso en linux

Desde el Kernel de Linux

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




task_struct
state
thread_info
usage
flags
...
run_list
tasks
...
mm
real_parent
parent
...
tty
...
thread
...
fs
files
signal
pending
...
thread_info
Low-level information for the process
mm_struct
Pointers to memory area descriptors
tty_struct
tty associated with the process
fs_struct
Current directory
files_struct
Pointers to file descriptors
signal_struct
Signals received

task_struct