Diseño de Sistemas Operativos

EXEC

Linux Kernel 2.6.28.7

Eduardo Jorge Carrasco Hernández

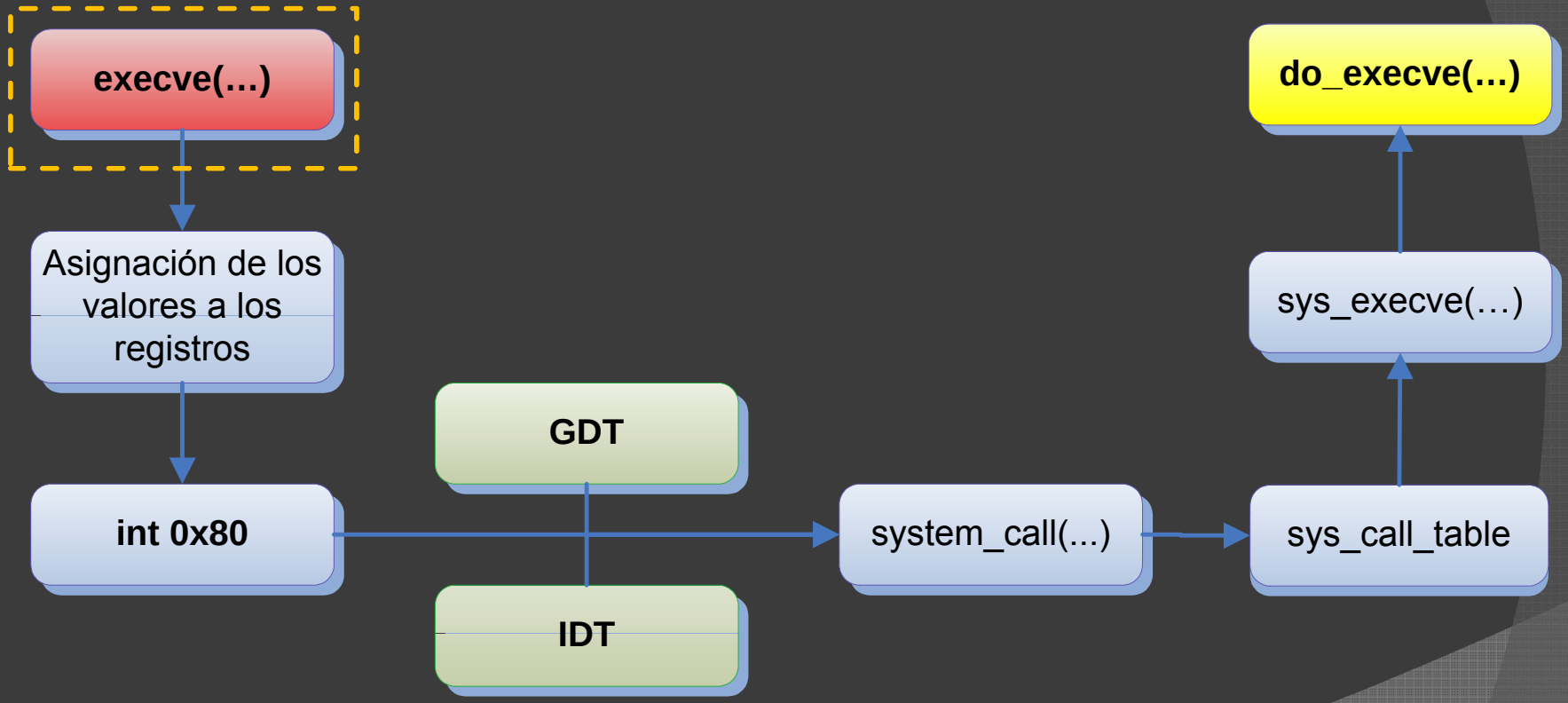
Emilio Macías Conde

¿Dónde estamos?

- Planificador de procesos
- Creación de hilos y subprocesos
- Llamadas al sistema que permiten esperar por la terminación de un hilo y notificarla
- Ejecución de programas desde ficheros ejecutables

Uso de execve

Execve utiliza el mismo flujo de funciones que muchas de las llamadas al sistema vistas hasta ahora



Uso de execve

¿Qué hace el execve?:

- Reemplaza al proceso invocador
- Carga en su espacio de memoria un nuevo proceso
- Si todo va bien, nunca retorna
- Si se producen errores, puede volver o no

La familia exec*

- Funciones
  - execl `execl("/bin/ls", "/bin/ls", "-l", (const char *)NULL);`
  - execlp `execlp("ls", "ls" , "-l", (const char *)NULL);`
  - execle `execle("/bin/ls","/bin/ls","-l",(const char *)NULL, env);`
  - execv `execv("/bin/ls", argv);`
  - execvp `execv("ls", argv);`
- Implementan distintas interfaces para execve
- Facilitan el uso de execve
- Su primer parámetro siempre es el fichero ejecutable
- Sus nombres indican qué parámetros requieren

l	*Los parámetros se pasan como ristras separadas por comas*
v	*Los parámetros se pasan en un vector*
p	*Si no encuentra el ejecutable, lo busca*
e	*Las variables de entorno se pasan en un vector*

La familia exec*

- En las funciones “l”, el primer parámetro a pasar al programa es el ejecutable:
- La lista de parámetros debe terminar en NULL

```
execl("/bin/ls","/bin/ls","-l",NULL)
```

- Los vectores deben tener su última posición a NULL

```
char *argv[] = {"-l", (const char*)NULL};
execv("/bin/ls", argv);
```

- Si no se requieren variables de entorno, se debe declarar la variable *environ*:

```
extern const char* environ[];
```

Formatos ejecutables

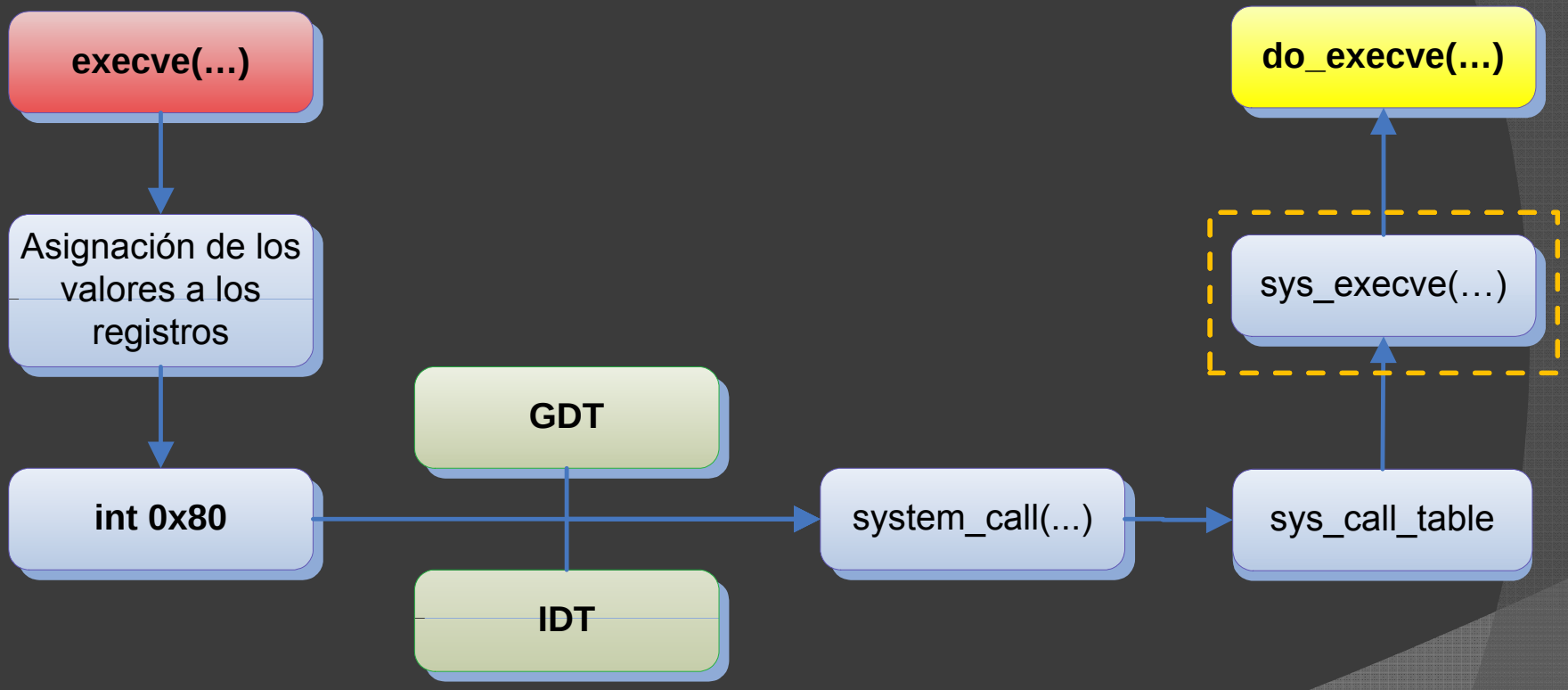
- Linux permite varios formatos ejecutables
- Hay un manejador binario por cada formato ejecutable
- Todos los manejadores binarios se definen como módulos externos al kernel
- Permite al S.O. abstraerse del modo en que se cargan
- Linux permite agregar nuevos formatos ejecutables
  - Se define el “número mágico” a encontrar
  - Se asocia un manejador binario
- No es necesario recompilar el núcleo

Formatos ejecutables

- Mecanismos de detección
  - Qué ficheros son ejecutables y cuáles no
  - Los 128 primeros bytes contienen el “número mágico”
    - Código que permite saber si el fichero es ejecutable
  - El Sistema Operativo identifica de qué formato se trata
  - Ejecuta funciones del manejador binario asociado
  - Manejador, es el encargado de alojar las páginas en memoria

Implementación de execve

Execve utiliza el mismo flujo de funciones que muchas de las llamadas al sistema vistas hasta ahora

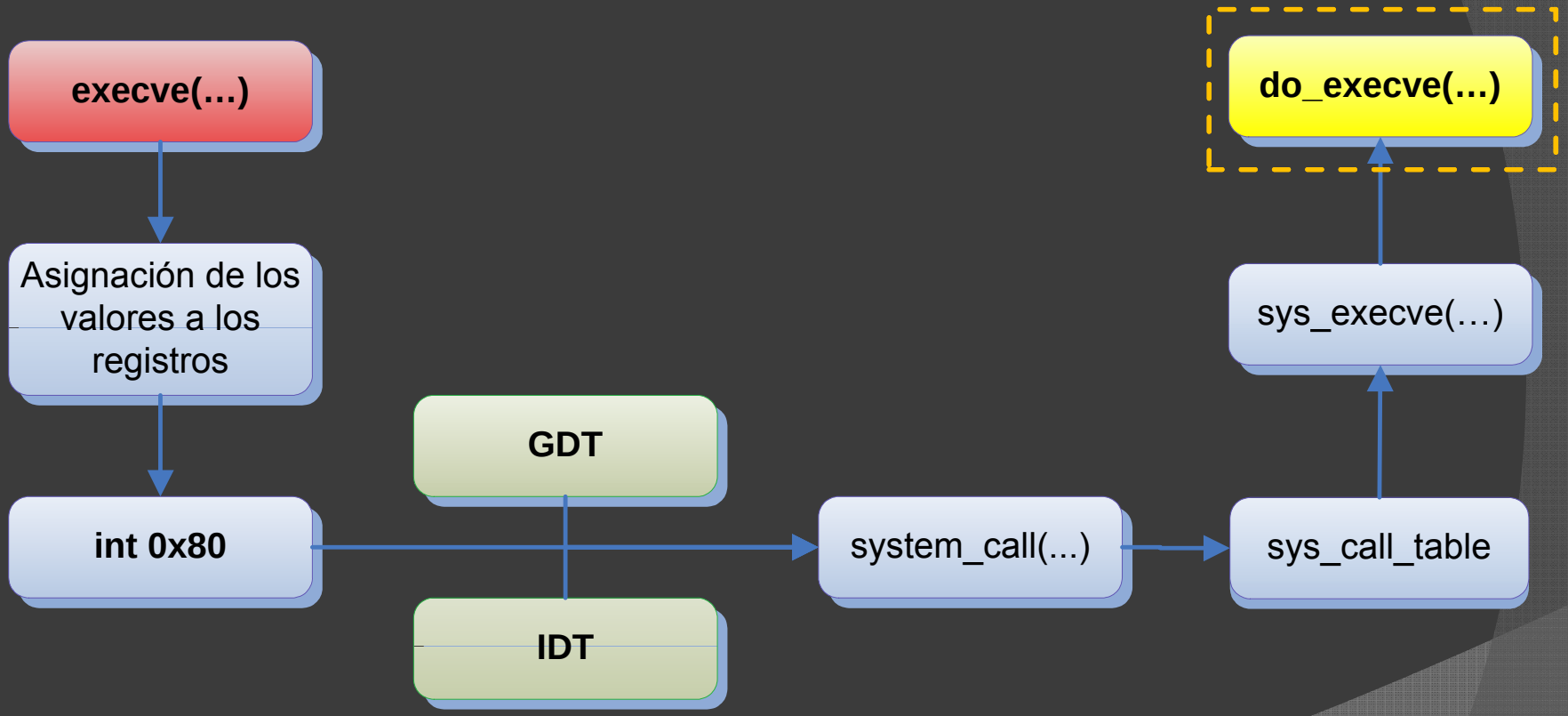


Llamada al sistema sys_execve

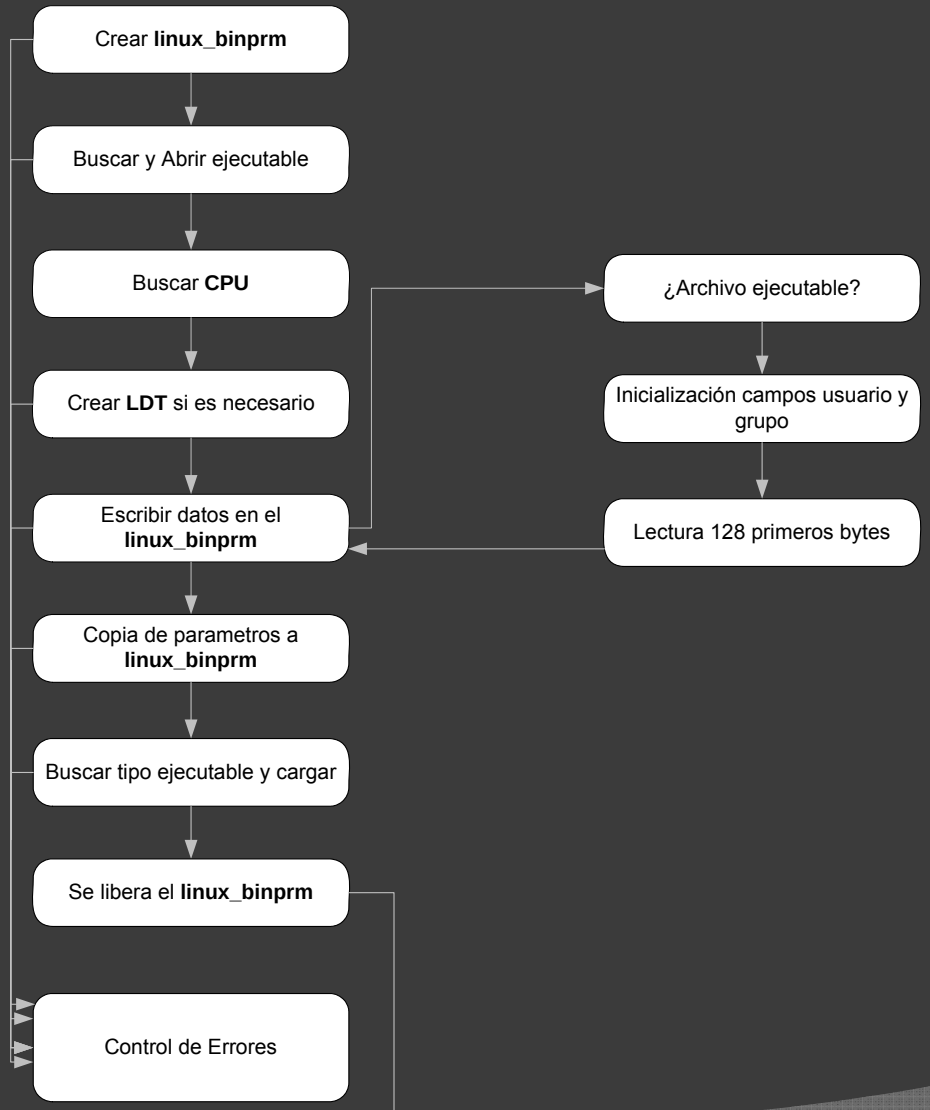
- Requiere 3 parámetros:
  - Dirección donde se encuentra la string del pathname del archivo ejecutable.
  - Array de punteros a strings. El final del array lo determinará un puntero a NULL dentro del array. Cada cadena representa un argumento de la línea de comandos.
  - Otro array de punteros a string, pero cada cadena representa una variable de entorno
- Esta llamada al sistema invoca a do_execve()

Invocación a do_execve()

Execve utiliza el mismo flujo de funciones que muchas de las llamadas al sistema vistas hasta ahora



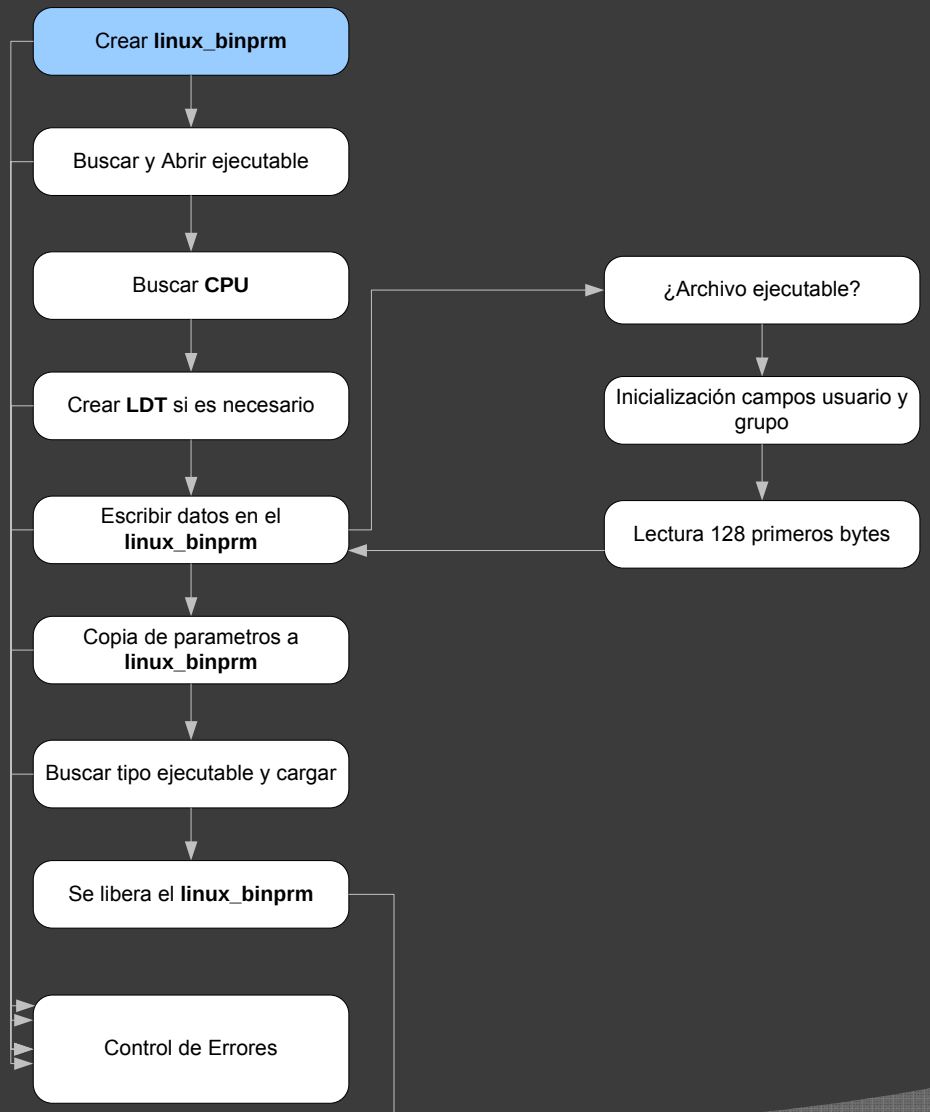
Ejecución de do_execve()



- Ejecución de do_execve()
- Control de errores centralizado
  - Se deshace lo hecho hasta ese punto
- Casi todos los pasos modulados en funciones

Se encuentra en:
/fs/exec.c#L1279

Ejecución de do_execve()



- Crear *linux_binprm*
- Se inicializa la estructura a lo largo de la ejecución

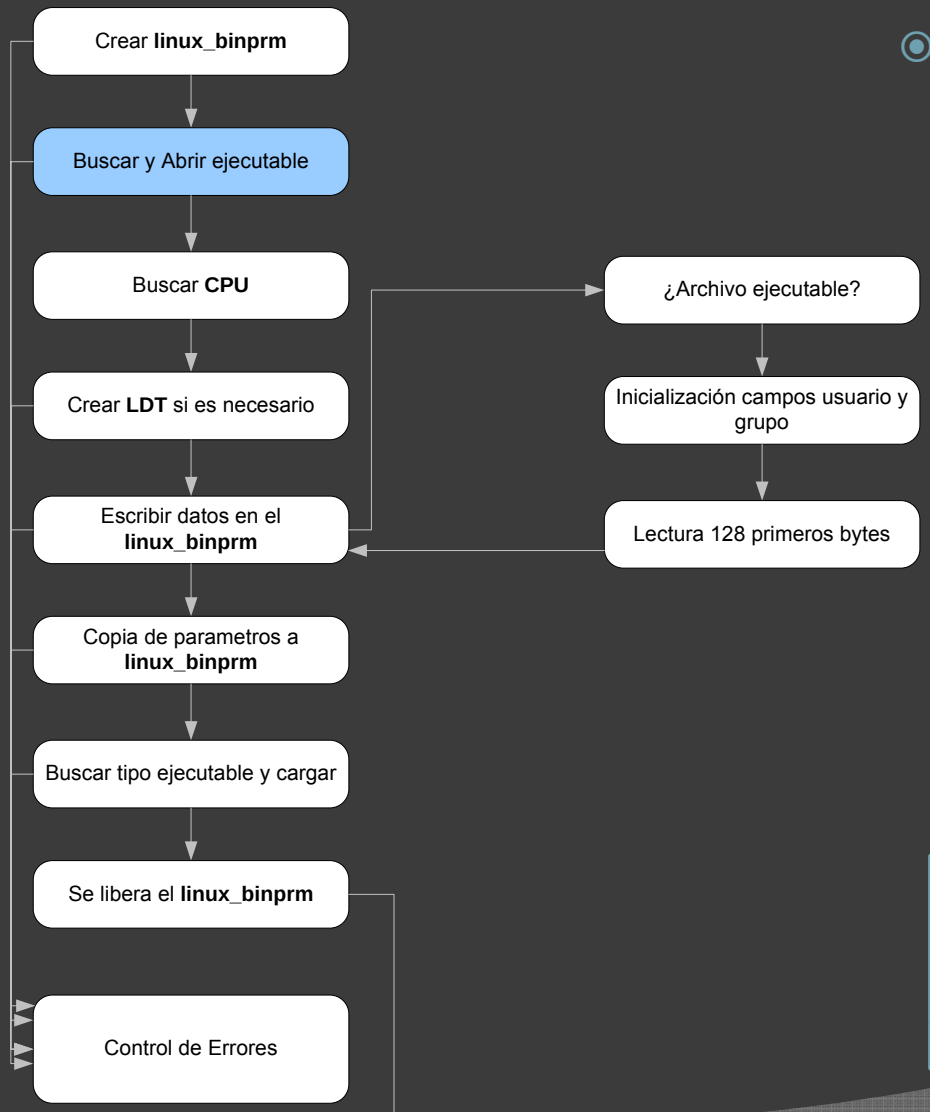
```
...
1284        struct linux_binprm *bprm;
...
1294        bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
...


...
1305        bprm->file = file;
1306        bprm->filename = filename;
1307        bprm->interp = filename;
...
```

linux_binprm se encuentra en:
/include/linux/binfmts.h#L27

Estructura linux_binprm

`char buf[BINPRM_BUF_SIZE]`	Se almacenarán los primeros 128 bytes del archivo ejecutable
`struct page *page[MAX_ARG_PAGES]`	Tabla de páginas del proceso Máximo 32 páginas
`struct mm_struct *mm`	Mapa de memoria del proceso
`unsigned long p`	Cantidad de memoria actualmente utilizada
`struct file * file`	Descriptor del Fichero ejecutable
`int e_uid, e_gid`	Permisos de ejecución del ejecutable
`kernel_cap_t cap_inheritable, cap_permitted, cap_effective`	Definen un conjunto de privilegios, privilegios heredados, permitidos y efectivos
`void *security`	Guarda información de seguridad
`int argc, envc`	Número de argumentos y de variables de entorno
`char * filename`	Nombre del ejecutable
`char * interp`	Nombre del binario realmente ejecutado La mayor parte del tiempo es lo mismo que *filename*, pero puede ser diferente en ciertas situaciones

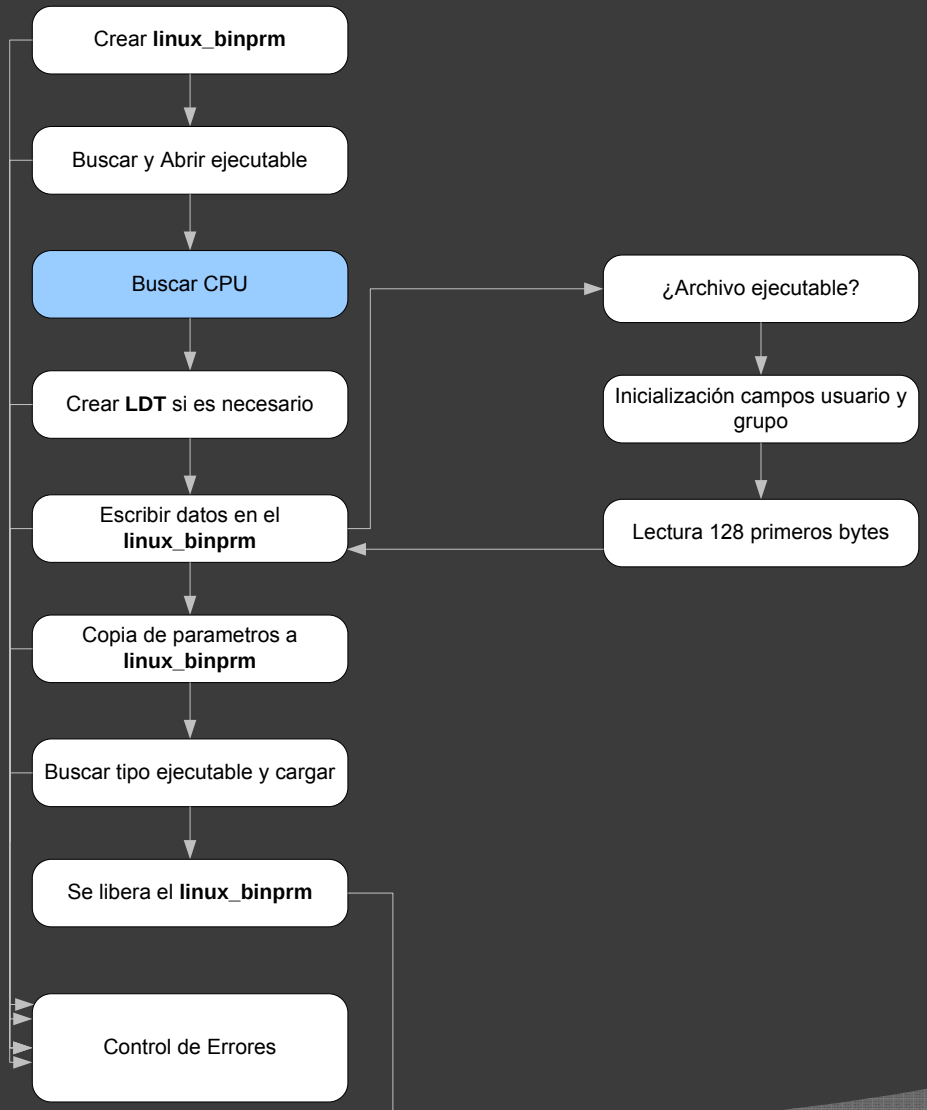
Ejecución de do_execve()



- Llamada a *open_exec()*
  - Abre el fichero, invocando a *path_lookup_open()*
    - Comprueba si es ejecutable por el proceso actual
  - Bloqueo de accesos de escritura con deny_write_access
  - Devuelve estructura file
    - File
      - Datos del ejecutable
      - ID del fichero abierto

```
1298        file = open_exec(filename);
1299        retval = PTR_ERR(file);
1300        if (IS_ERR(file))
1301                goto out_kfree;
```

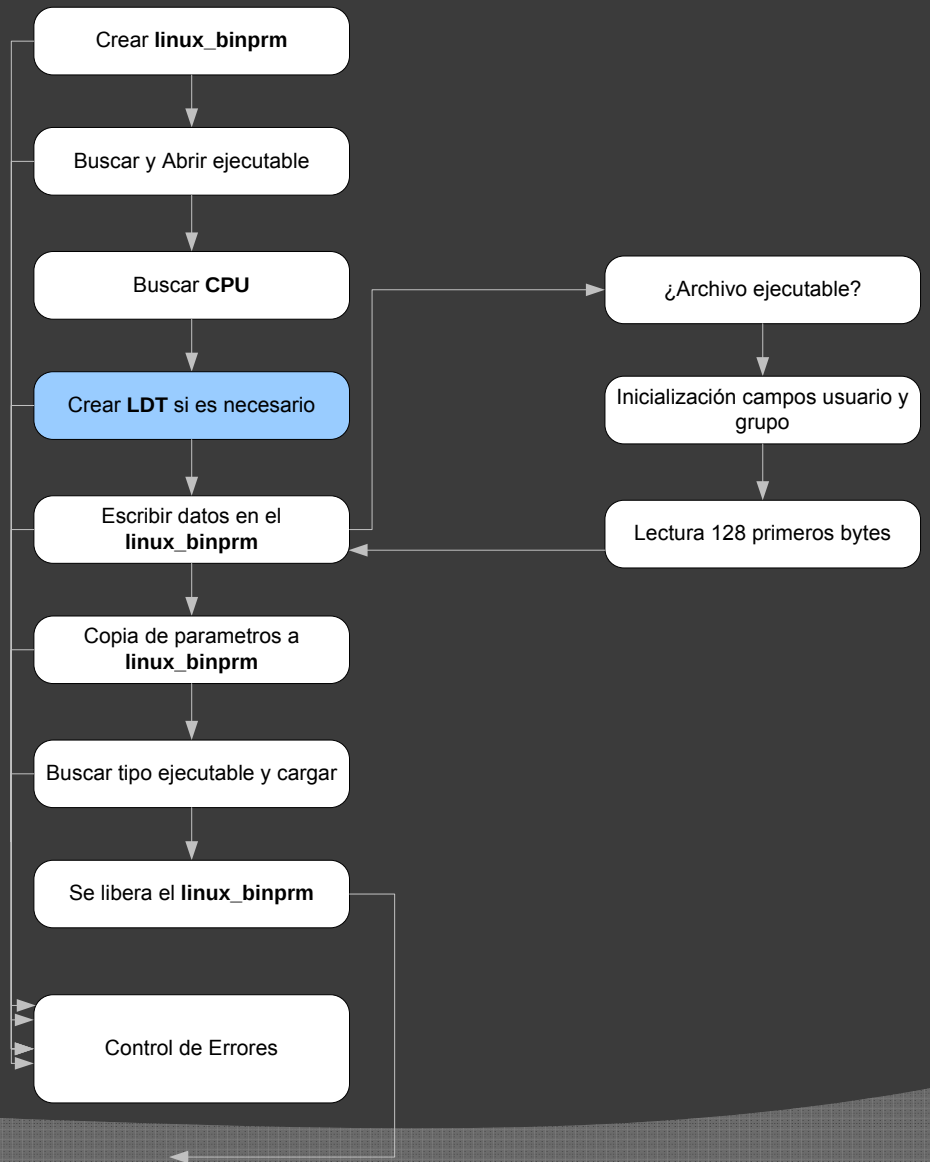
Ejecución de do_execve()



- Llamada a *sched_exec()*
- Útil para sistemas multiprocesador
- Busca CPU más descargada
- Traslada el proceso a esa CPU

```
...
1303        sched_exec();
...
```

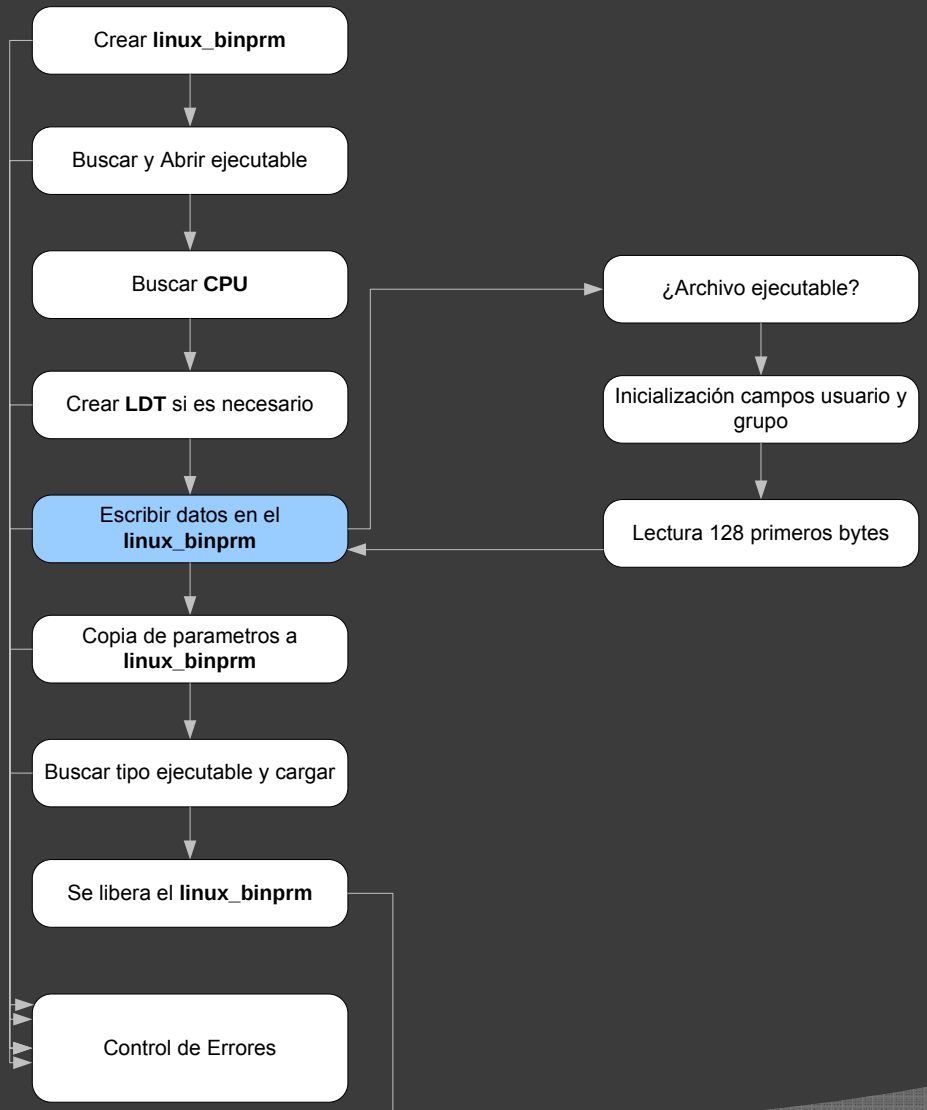
Ejecución de do_execve()



- Llamada a *init_new_context()* dentro de *bprm_mm_init(...)*
- Comprobar si el proceso actual tiene LDT personalizada
  - Si es así, se crea una nueva LDT y se rellena para ser usada por el nuevo programa

```
...
363       err = init_new_context(current, mm);
364       if (er)
365             goto err;
...
1309    retval = bprm_mm_init(bprm);
1310    if (retval)
1311          goto out_file;
...
```

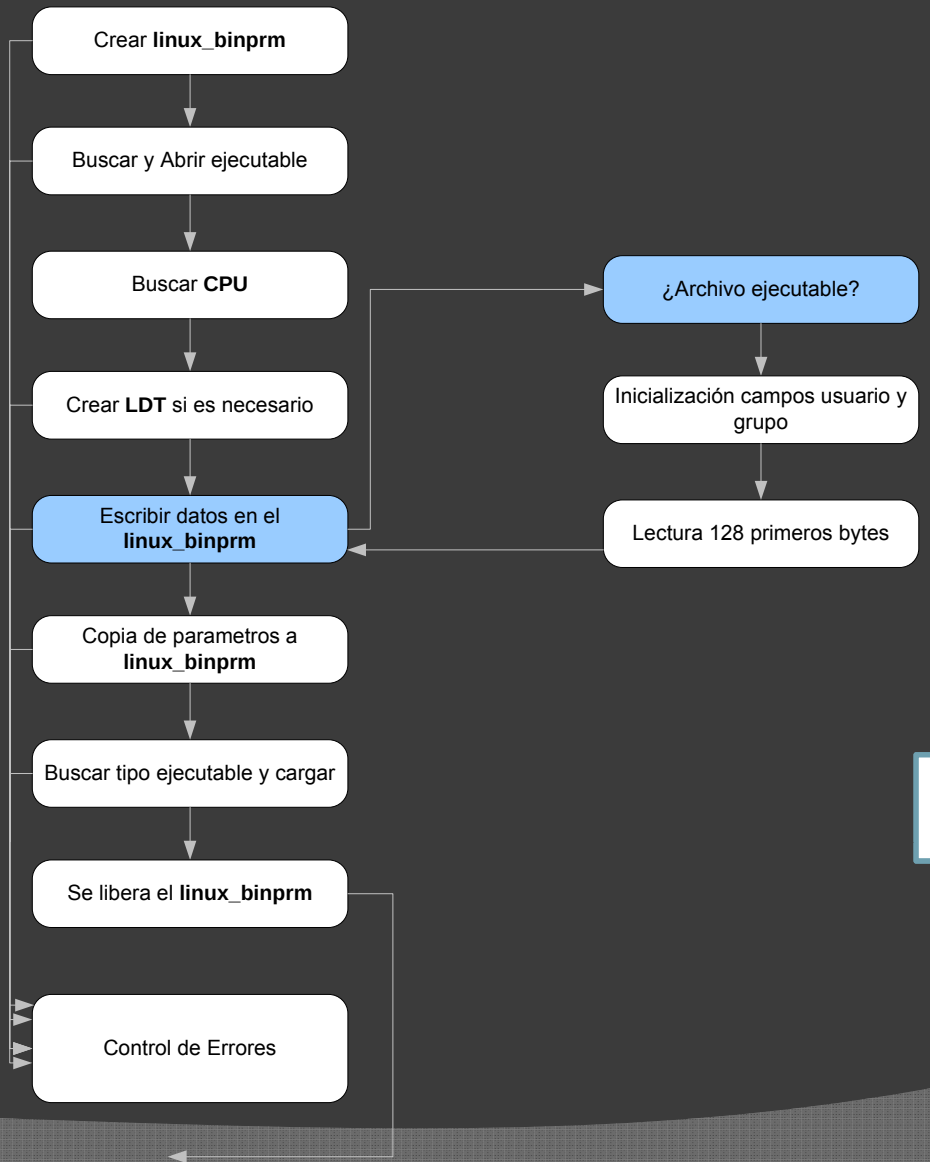
Ejecución de do_execve()



- Llamada a *prepare_binprm()*
- Rellenado de algunos campos de la estructura *linux_binprm*

```
...
1325        retval = prepare_binprm(bprm);
1326        if (retval < 0)
1327                goto out;
...
```

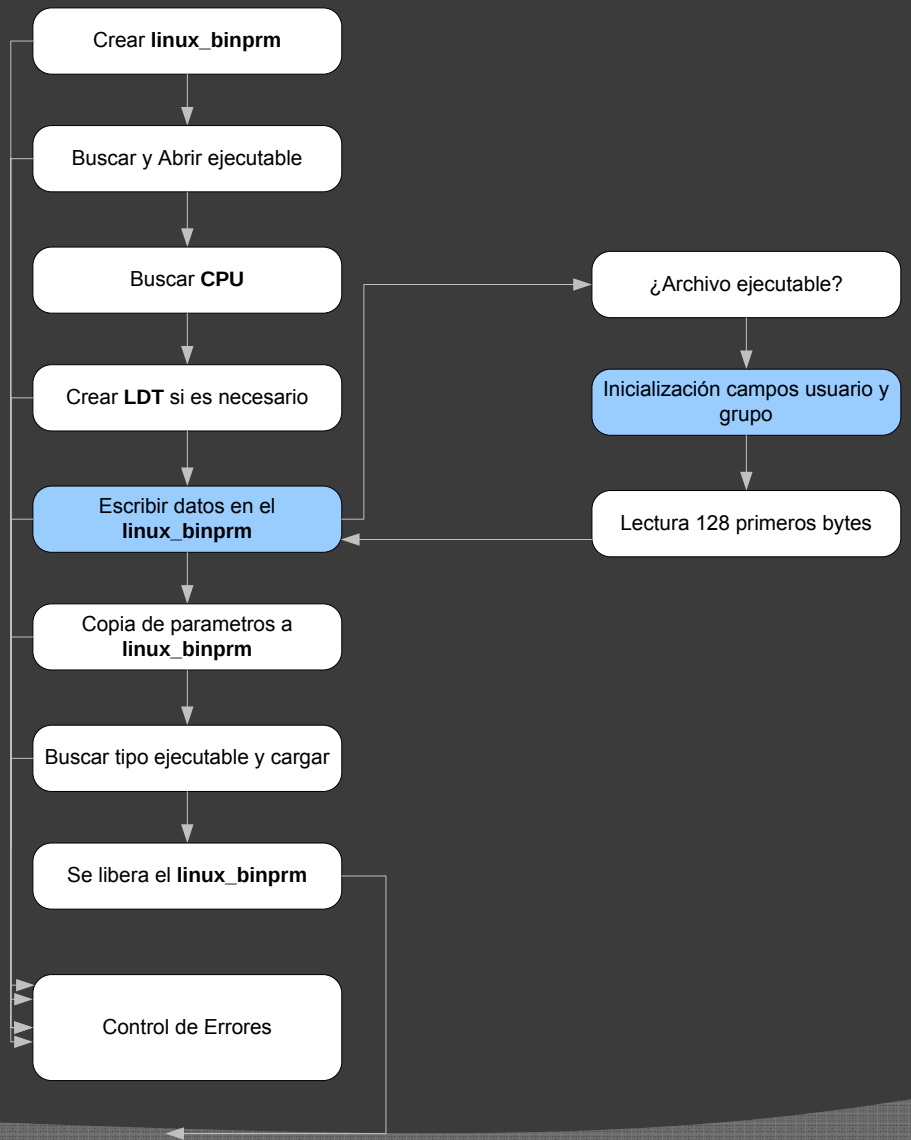
Ejecución de do_execve()



- ¿Archivo ejecutable por el proceso actual?

```
1047    if (bprm->file->f_op == NULL)
1048            return -EACCES;
```

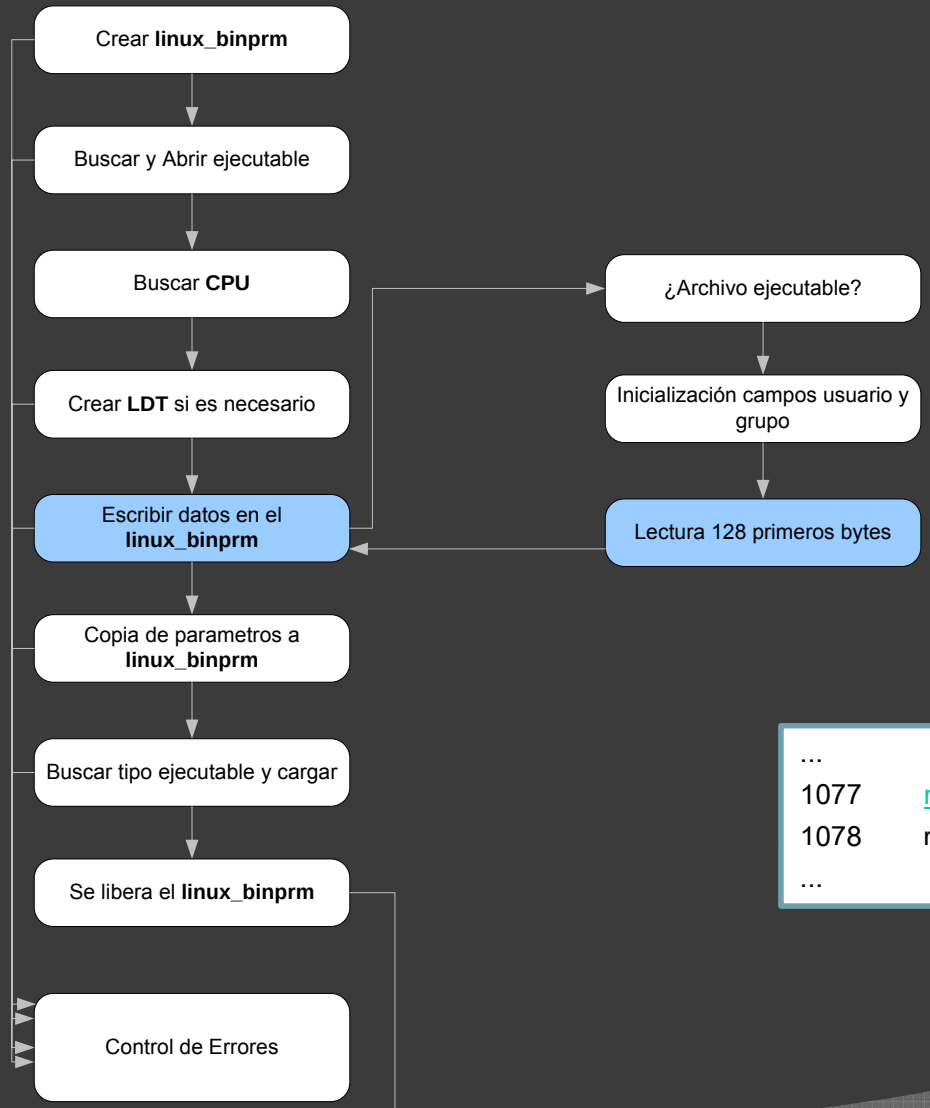
Ejecución de do_execve()



- Inicialización campos *e_uid* y *e_gid*
- Se tienen en cuenta los permisos del fichero ejecutable

```
...
1050    bprm->e_uid = current->euid;
1051    bprm->e_gid = current->egid;
…
1053    if(!(bprm->file->f_vfsmnt->mnt_flags & MNT_NOSUID)) {
1055        if (mode & S_ISUID) {
1056                current->personality &= ~PER_CLEAR_ON_SETID;
1057                bprm->e_uid = inode->i_uid;
1058        }
1066        if ((mode & (S_ISGID | S_IXGRP)) == (S_ISGID | S_IXGRP)) {
1067                current->personality &= ~PER_CLEAR_ON_SETID;
1068                bprm->e_gid = inode->i_gid;
1069        }
1070    }
…
1073    retval = security_bprm_set(bprm);
...
```

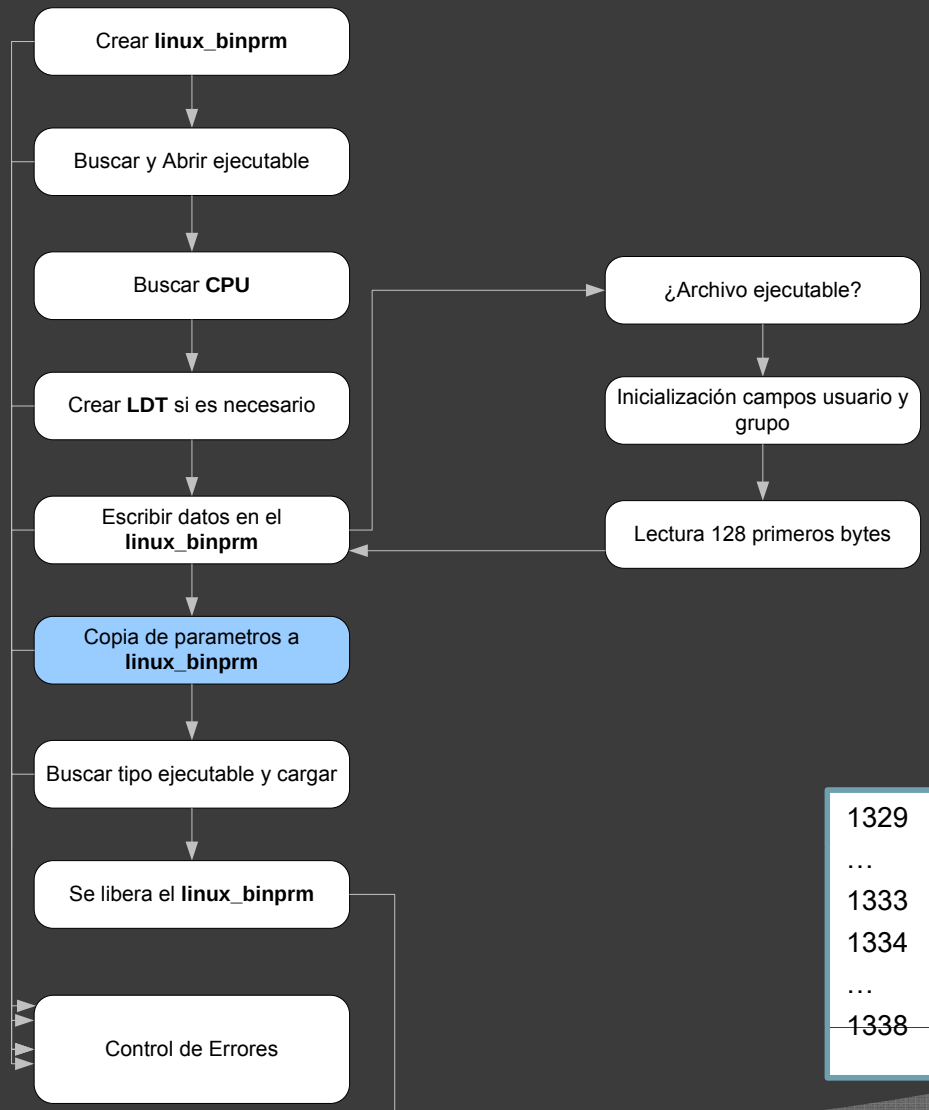
Ejecución de do_execve()



- Inicializa con ceros *buf* en la estructura *linux_bprm*
- Rellena *buf* con los primeros 128 bytes del archivo ejecutable
- Útil para identificación del tipo ejecutable
  - Número mágico

```
...
1077    memset(bprm->buf,0,BINPRM_BUF_SIZE);
1078    return kernel_read(bprm->file,0,bprm->buf,BINPRM_BUF_SIZE);
...
```

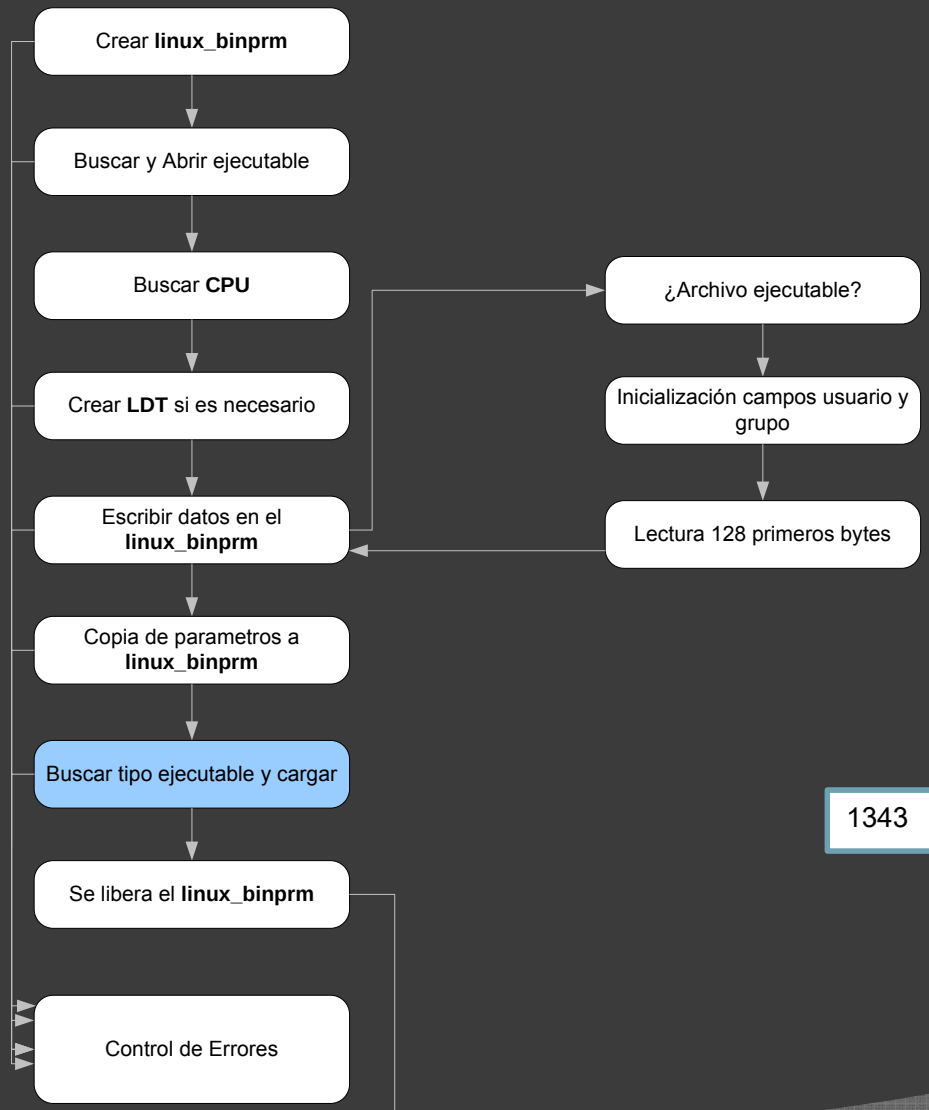
Ejecución de do_execve()



- Copia de parámetros a *linux_binprm*
  - *Pathname*
  - *Argumentos línea de comandos*
  - *Variables de entorno*

```
1329        retval = copy_strings_kernel(1, &bprm->filename, bprm);
…
1333        bprm->exec = bprm->p;
1334        retval = copy_strings(bprm->envc, envp, bprm);
…
1338        retval = copy_strings(bprm->argc, argv, bprm);
```

Ejecución de do_execve()



- Llamada a *search_binary_handler()*
- Búsqueda del tipo de ejecutable
- Carga del ejecutable
  - *load_binary()*

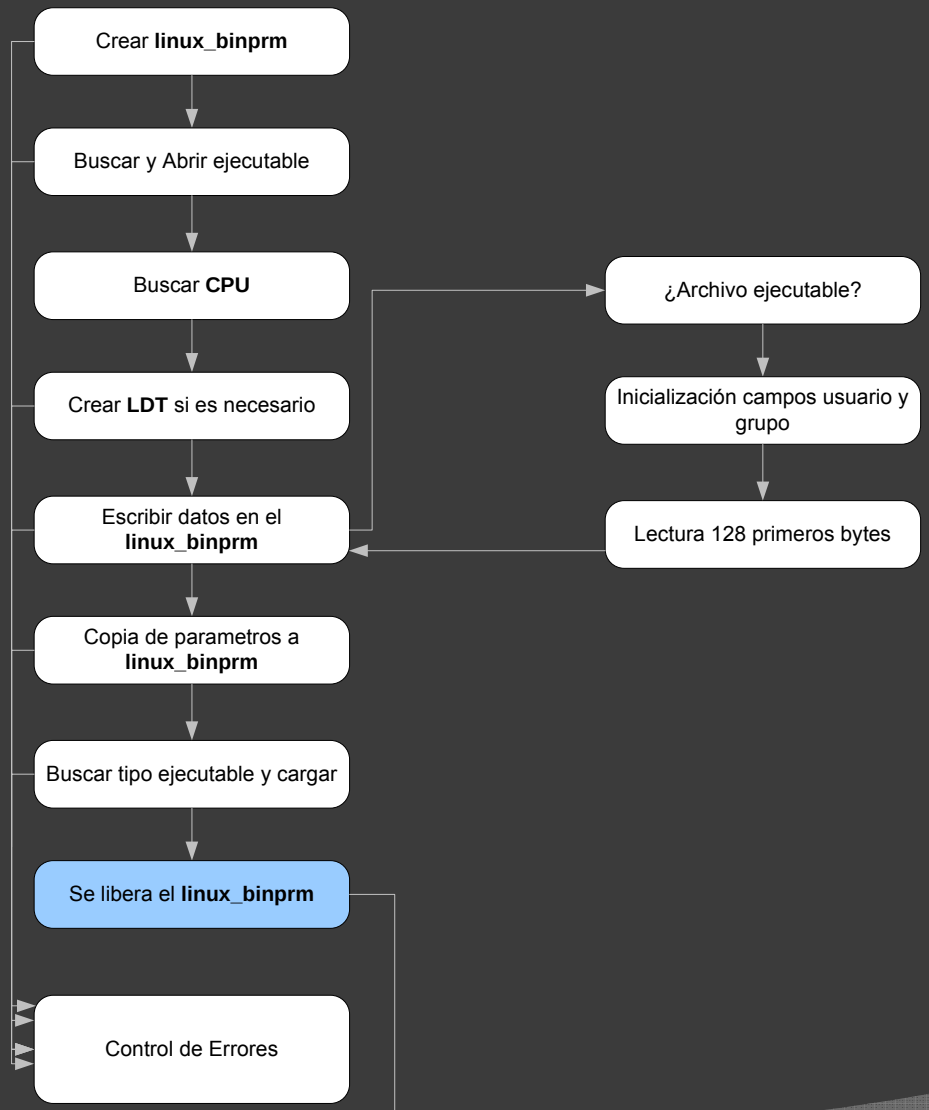
```
1343        retval = search_binary_handler(bprm,regs);
```

Ejecución de do_execve()

- *search_binary_handler()*
- Para cada formato se llama a *load_binary* para buscar un manejador adecuado
- Comprueba que el fichero puede ser ejectuado
- Si no se puede traer el módulo se pasa al siguitente formato

```
1213   for (try=0; try<2; try++) {
1214       read_lock(&binfmt_lock);
1215       list_for_each_entry {
1216           int (*fn)(struct linux_binprm *, struct
                       pt_regs *) = fmt->load_binary;
1217           if (!fn)
1218               continue;
1219           if (!try_module_get(fmt->module))
1220               continue;
1221           read_unlock(&binfmt_lock);
…
1263       }
1264   }
```

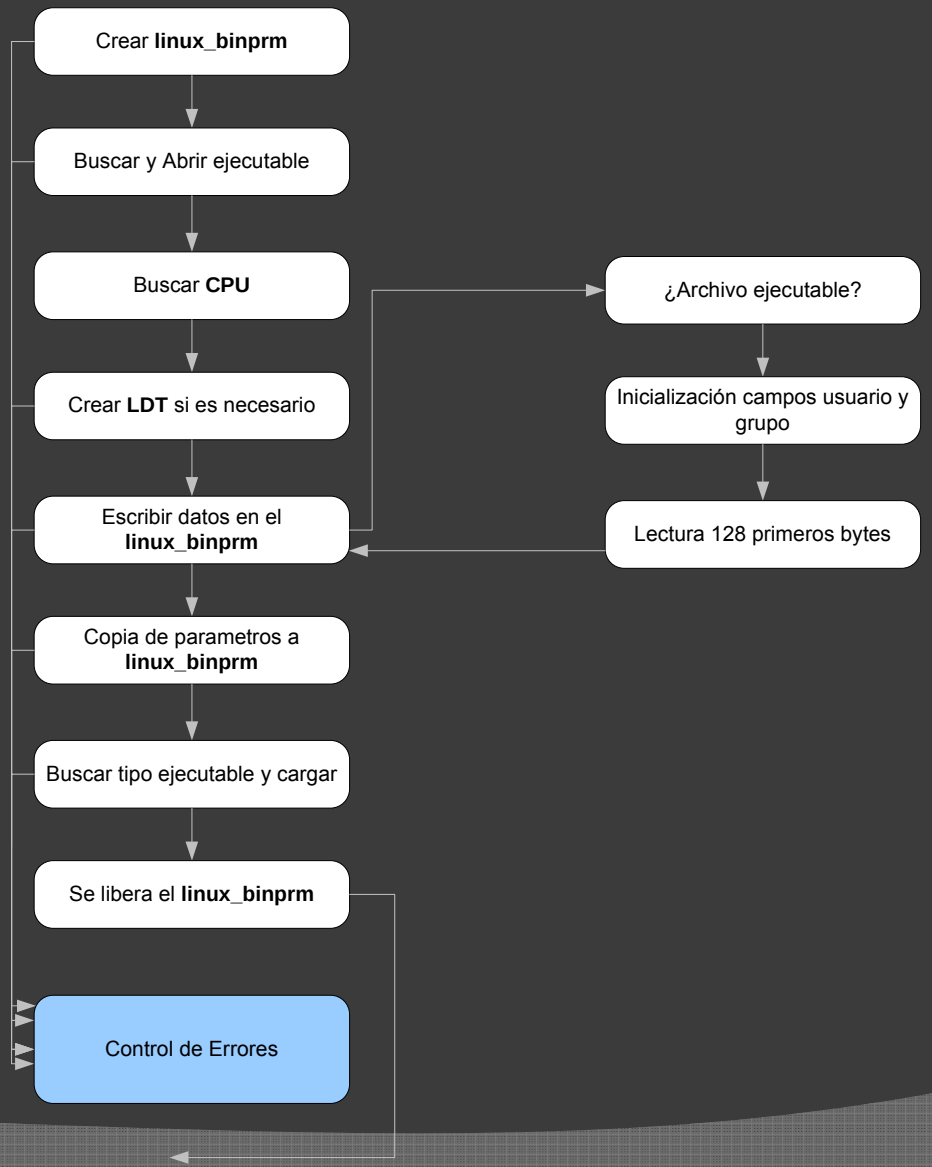
Ejecución de do_execve()



- Fin de *do_exec()*
- Recuperación en caso de retorno

```
1344    if (retval >= 0) {
1345            /* execve success */
1346            security_bprm_free(bprm);
1347            acct_update_integrals(current);
1348            free_bprm(bprm);
1349            if (displaced)
1350                    put_files_struct(displaced);
1351            return retval;
1352    }
```

Ejecución de do_execve()



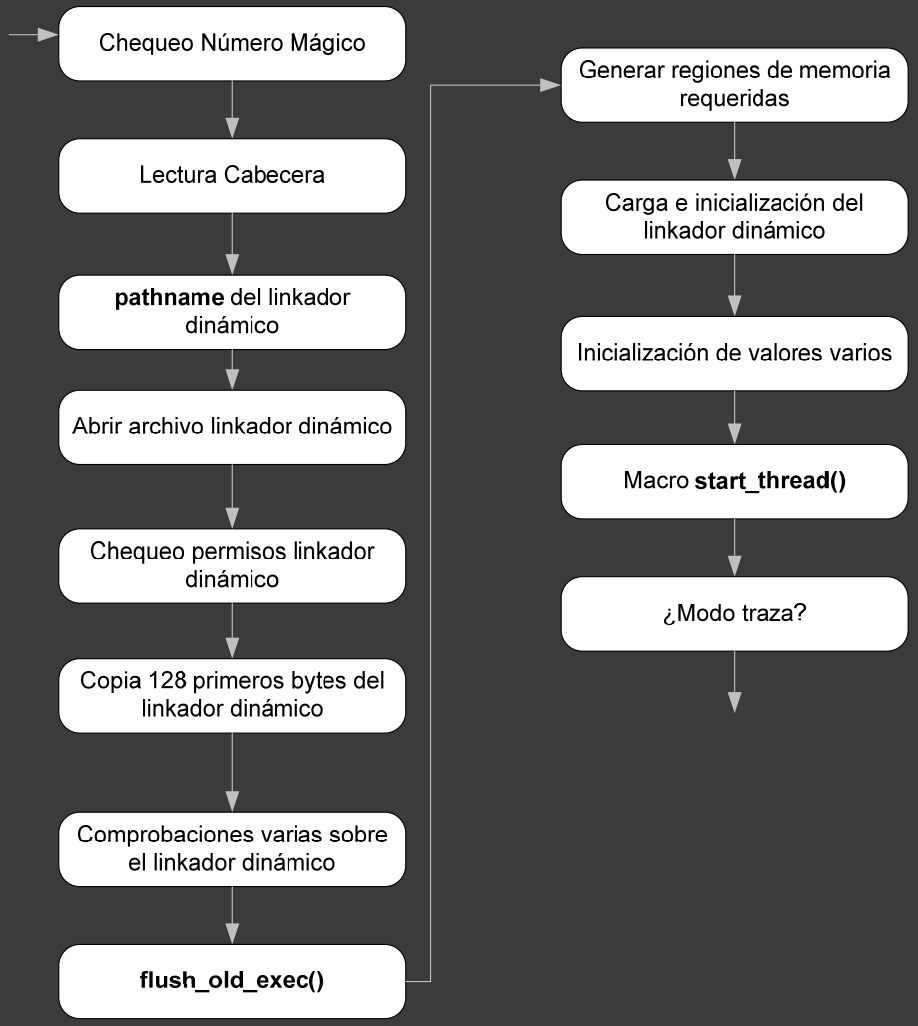
- Control de errores
- Distintos puntos de entrada

```
1354 out:
1355      if (bprm->security)
1356            security_bprm_free(bprm);
1357
1358 out_mm:
1359      if (bprm->mm)
1360            mmput (bprm->mm);
1361
1362 out_file:
1363      if (bprm->file) {
1364            allow_write_access(bprm->file);
1365            fput(bprm->file);
1366      }
1367 out_kfree:
1368      free_bprm(bprm);
1369
1370 out_files:
1371      if (displaced)
1372            reset_files_struct(displaced);
1373 out_ret:
1374      return retval;
1375 }
```

Ejecución de do_execve()

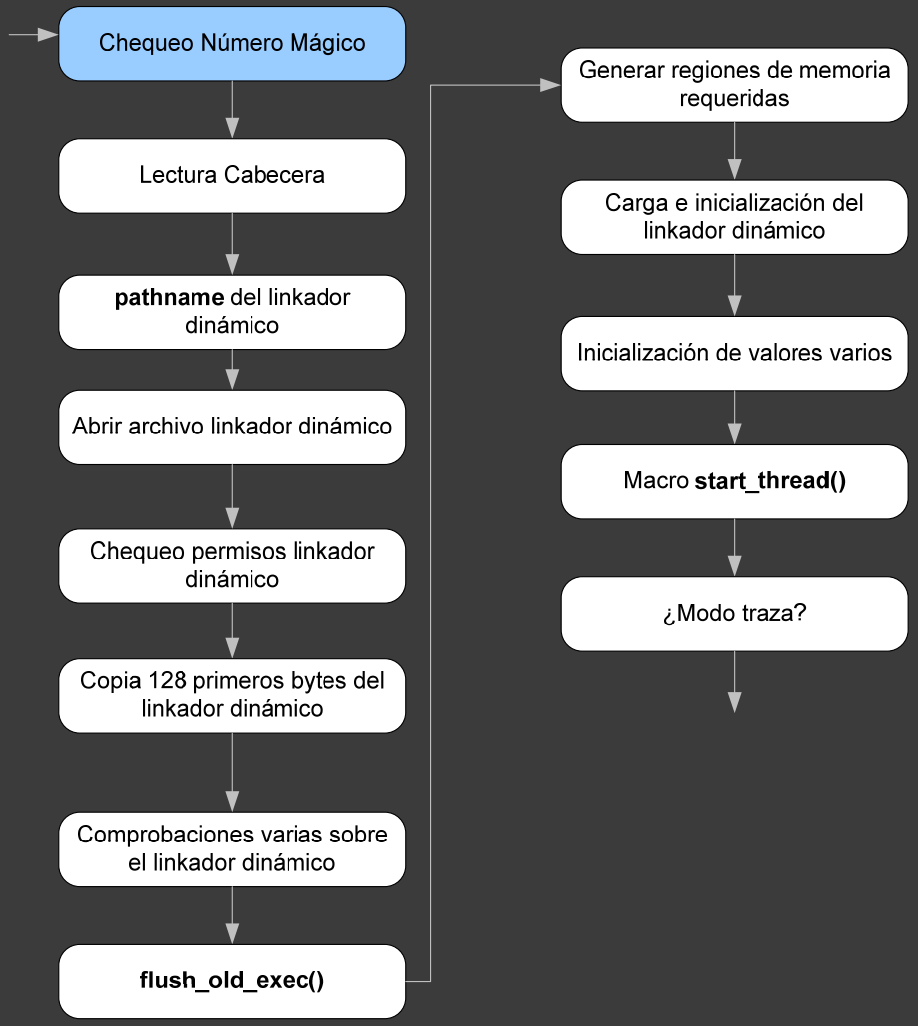
LOAD_BINARY()

load_binary()



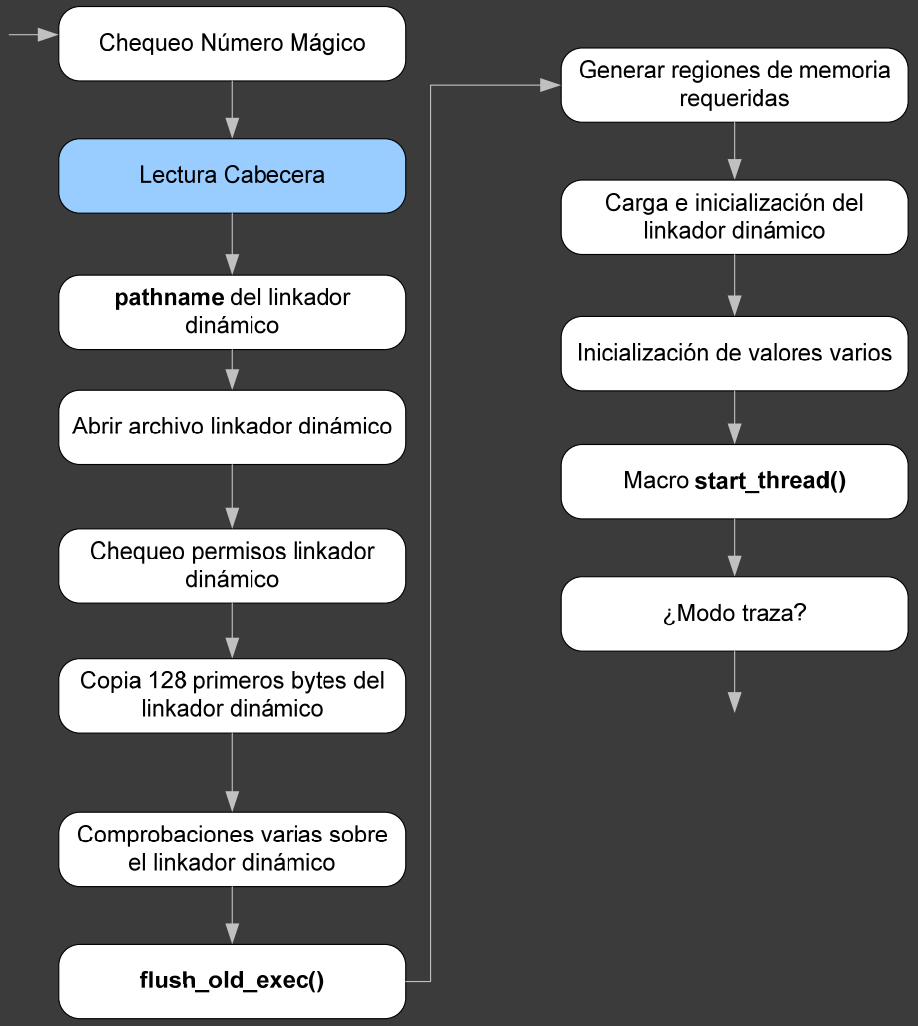
- load_binary()
- Dependiente del formato
- Veremos su comportamiento general

load_binary()



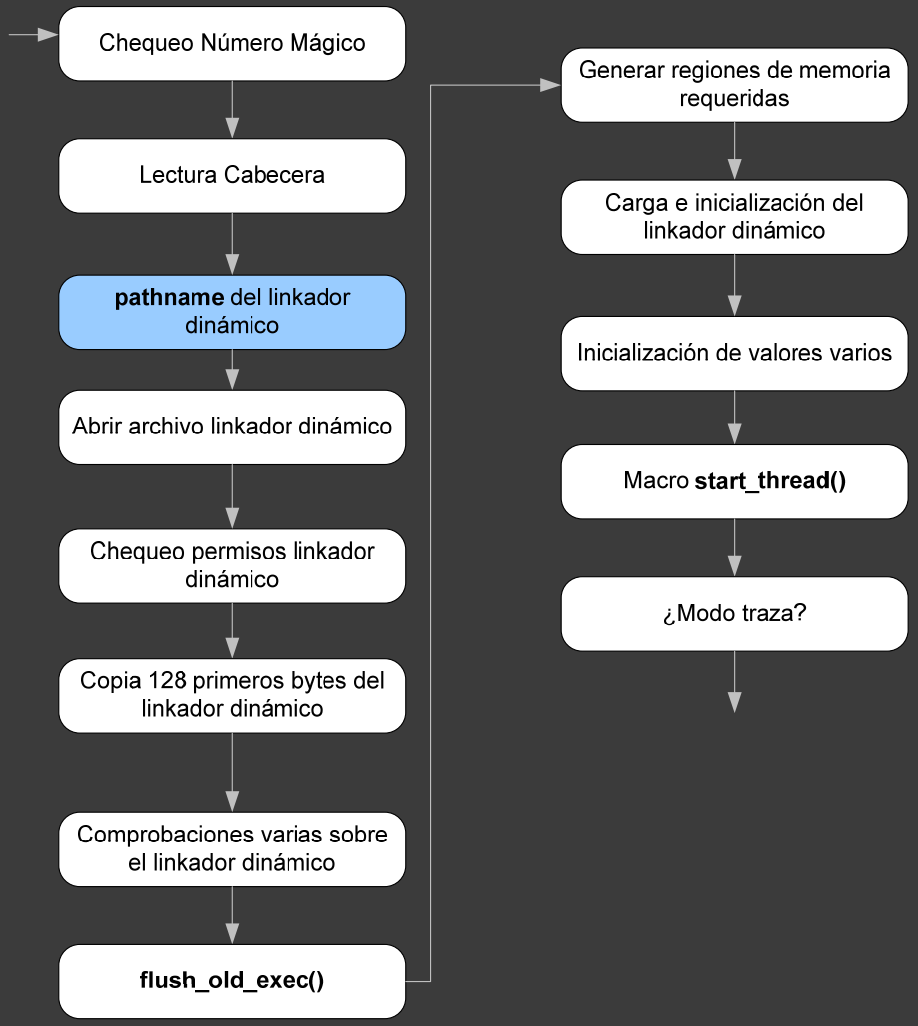
- Comprueba el número mágico
- Particular de cada formato
- Dentro de los 128 bytes
- Si no corresponde:
  - *-ENOEXEC*

load_binary()



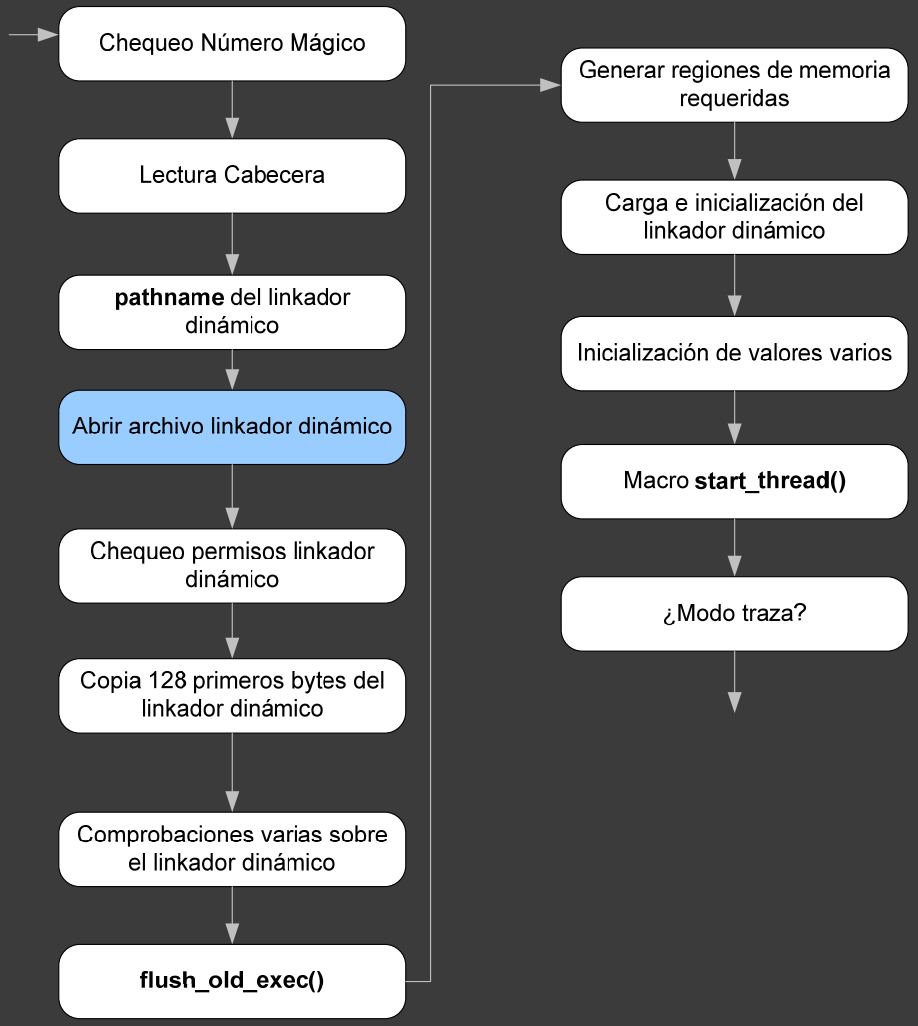
- Lectura de la cabecera del ejecutable
- Datos sobre:
  - *Segmentos*
  - *Librerías requeridas*

load_binary()



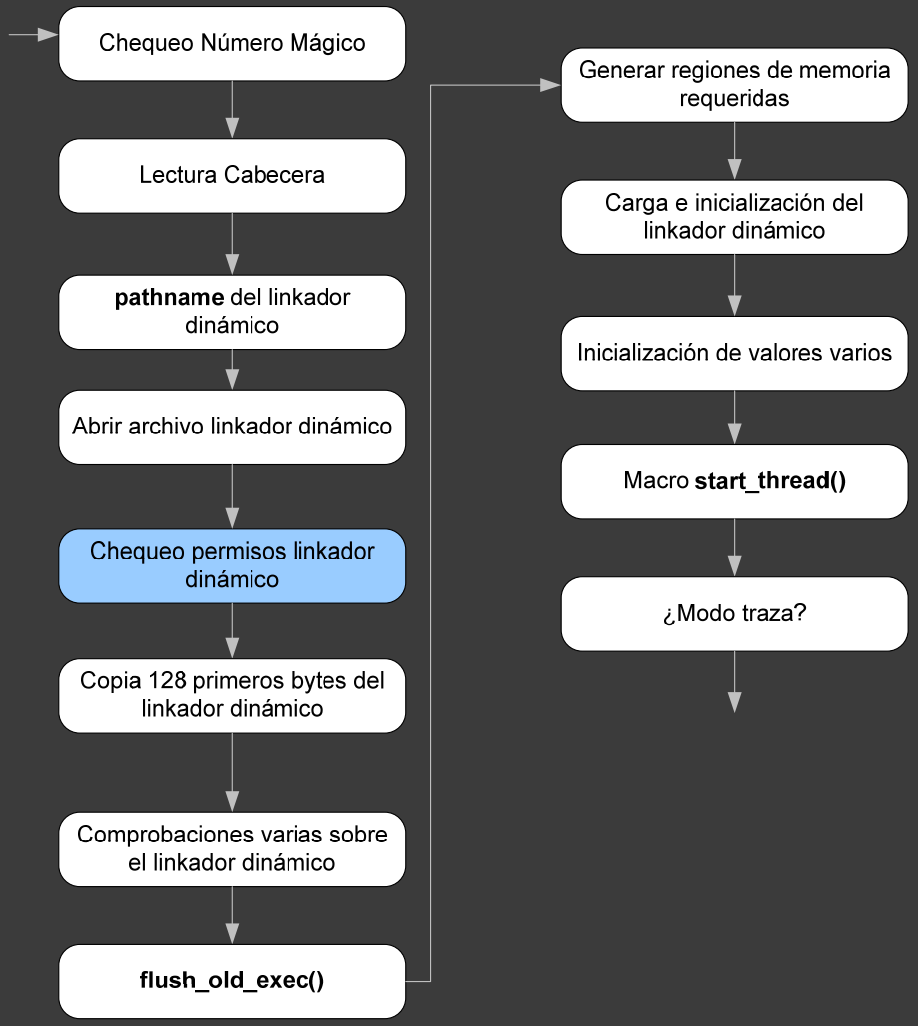
- Se obtiene el pathname del linkador dinámico
- Encargado de buscar y cargar las librerías
- Información extraída de la cabecera

load_binary()



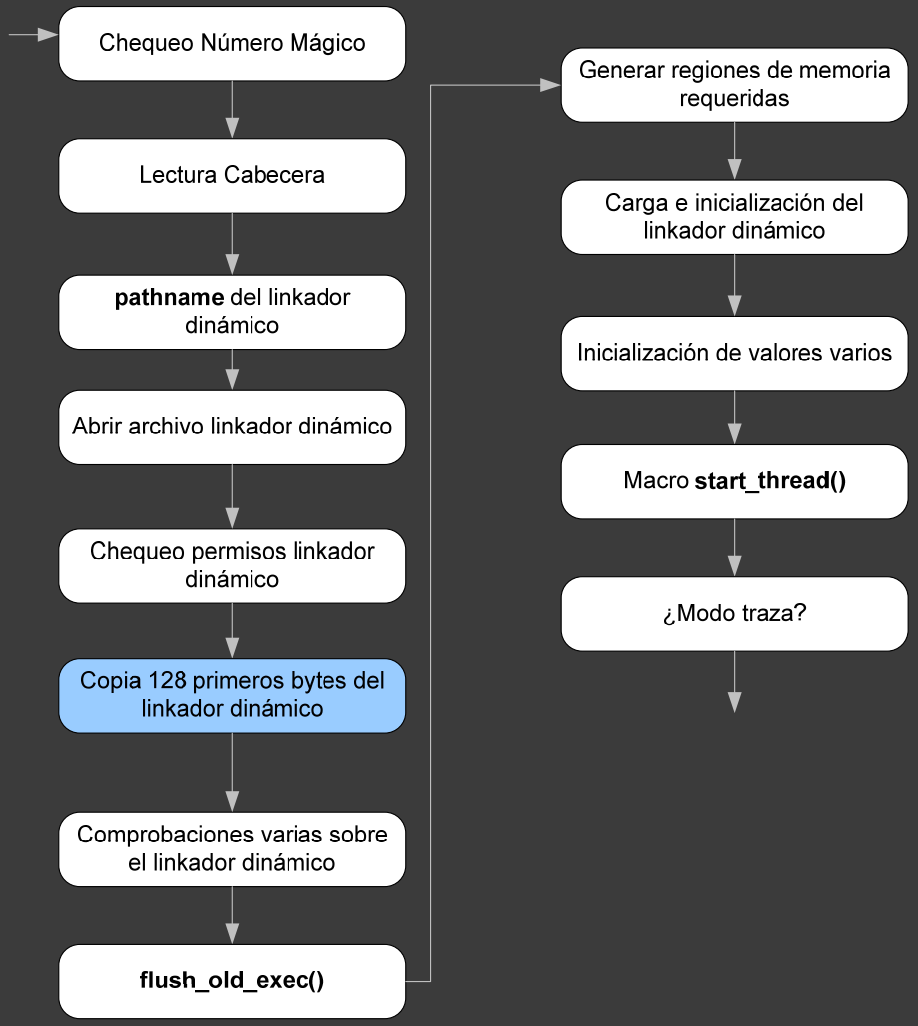
- Se abre el fichero del linkador dinámico
- Similar al *open_exec()*

load_binary()



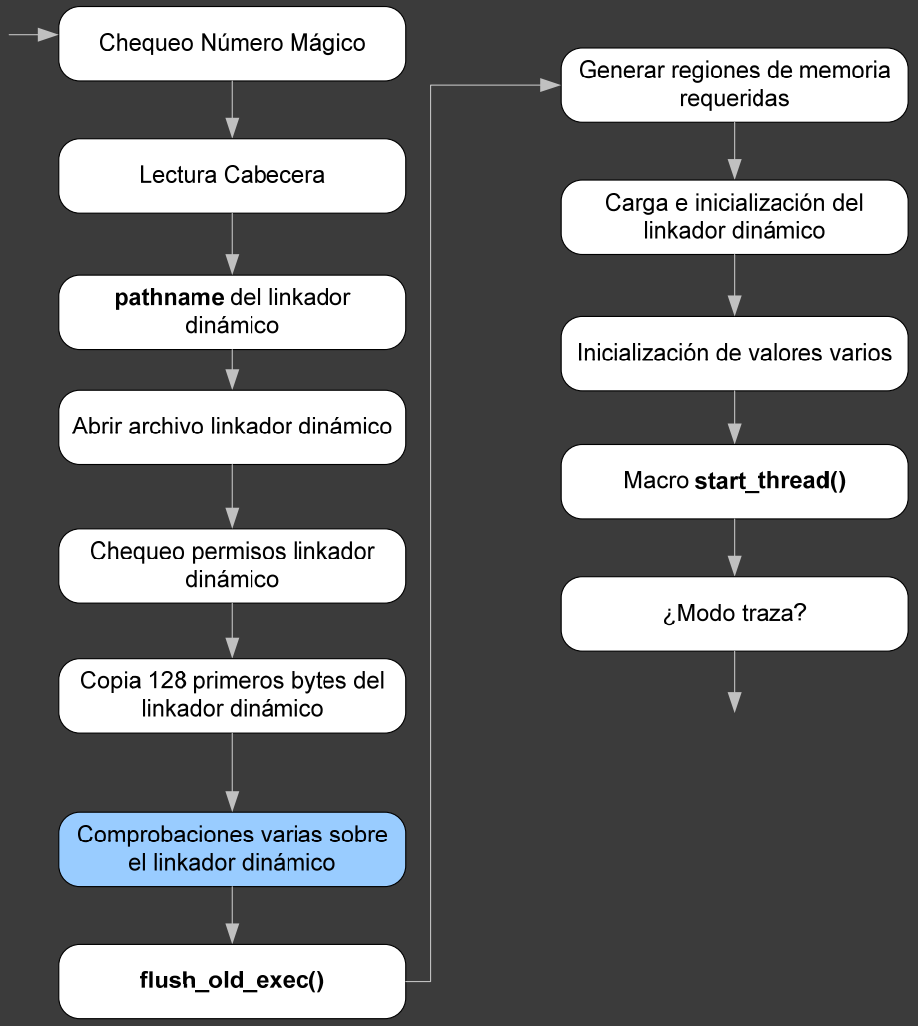
- Chequeo de permisos
- Comprueba si el proceso actual puede ejecutar el linkador dinámico

load_binary()



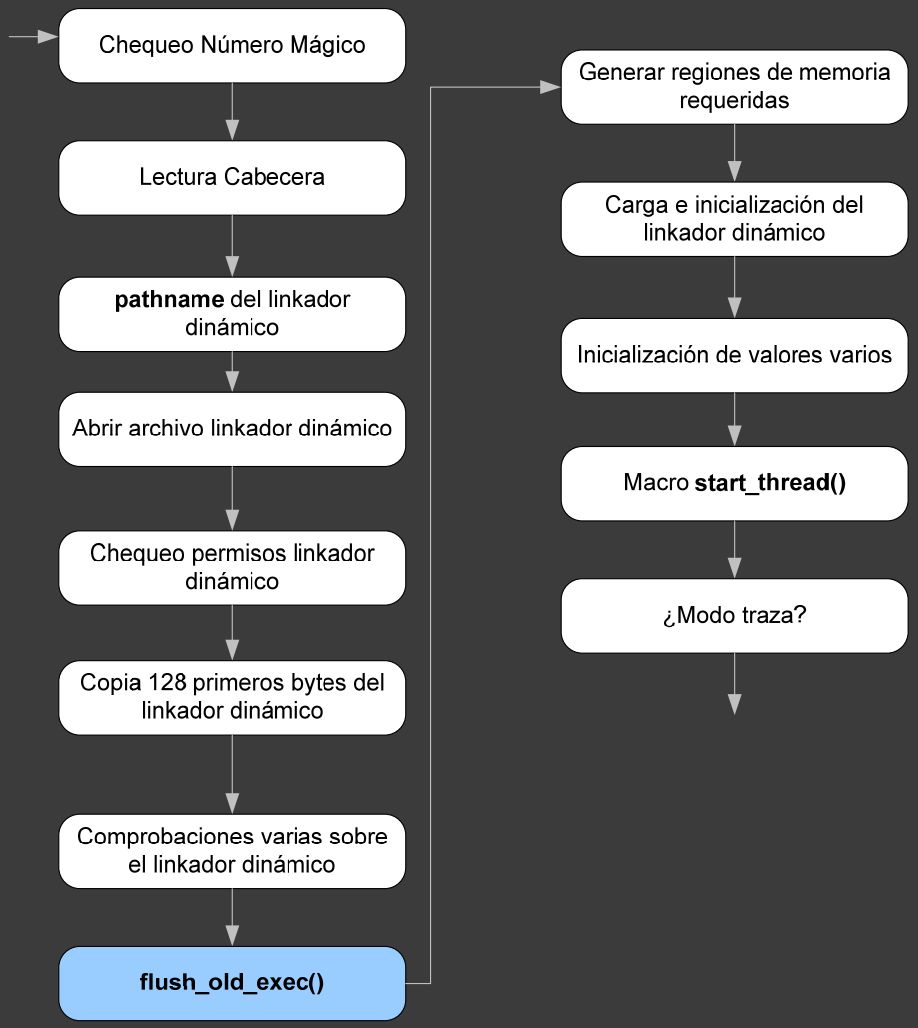
- Lectura primeros 128 bytes del linkador dinámico

load_binary()



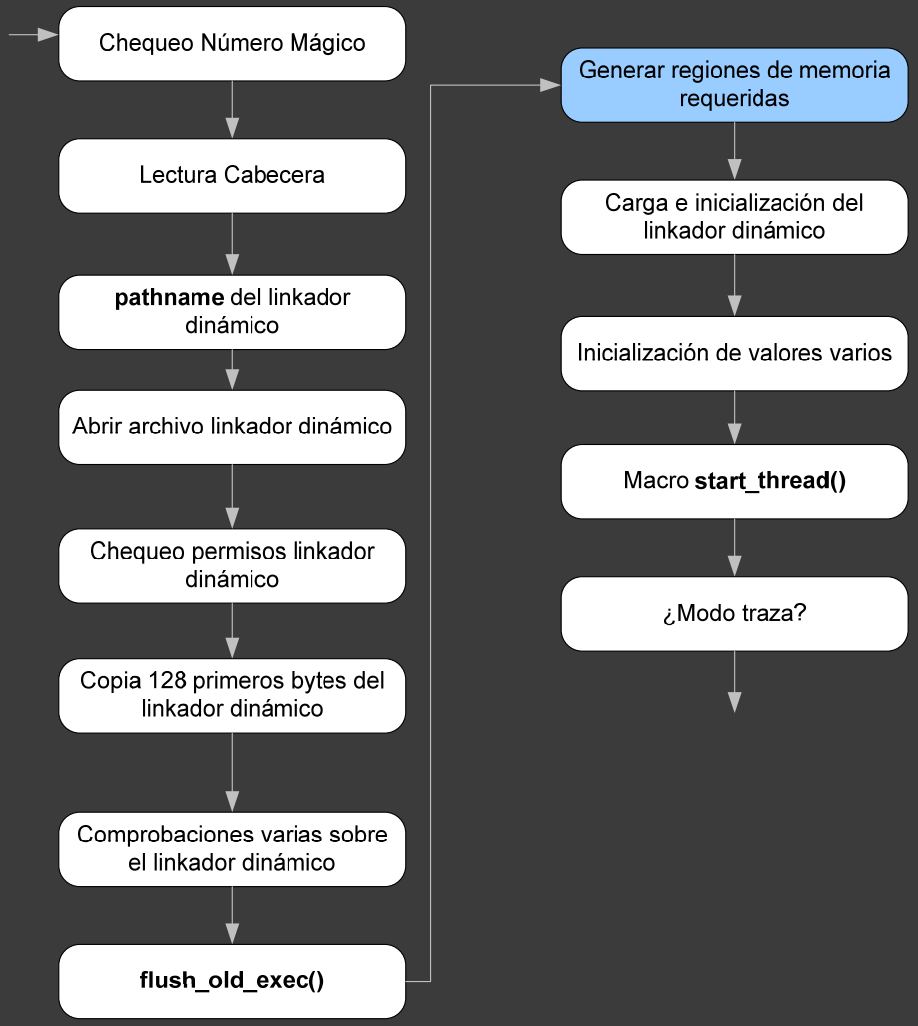
- Comprobaciones varias sobre el linkador dinámico
  - *Formato reconocible*
  - *…*

load_binary()



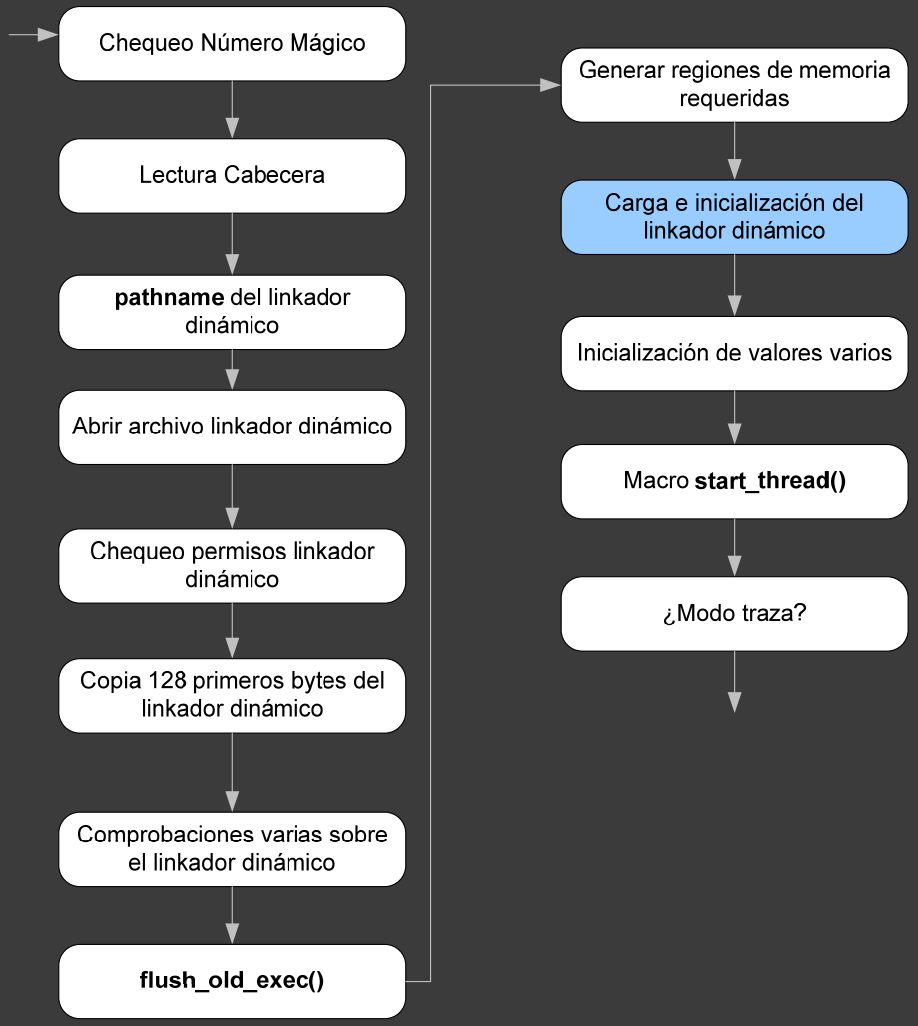
- Llamada a *flush_old_exec()*
- Pone a *default* las señales
  - Si son compartidas:
    - Primero copia
    - Luego se desvincula
- Cerrar ficheros abiertos
  - Si son compartidos:
    - Igual al anterior
- Liberación de memoria y páginas
- Limpieza de registros en punto flotante
- A partir de este momento no hay vuelta atrás:
  - No se puede retornar a la llamada del execve()
  - El código de ese proceso no existe

load_binary()



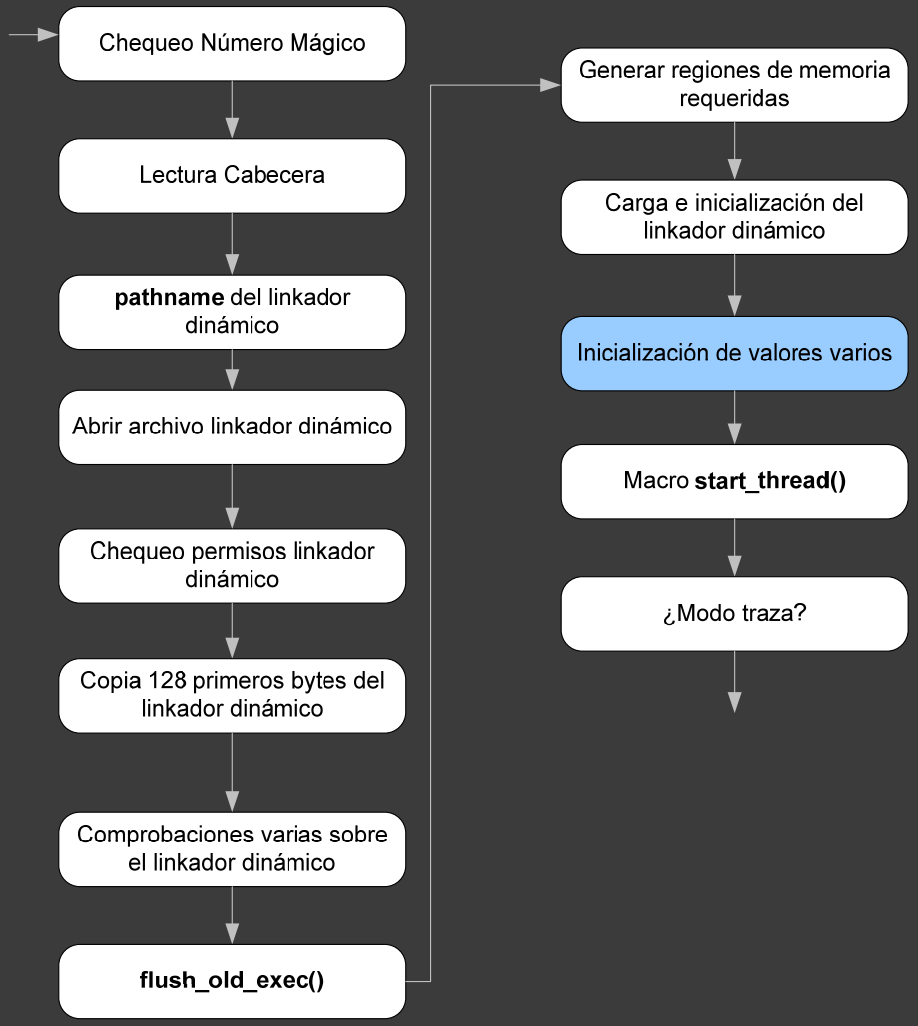
- Generación de regiones de memoria requeridas
- Carga de los segmentos
  - Código
  - Datos
  - ...

load_binary()



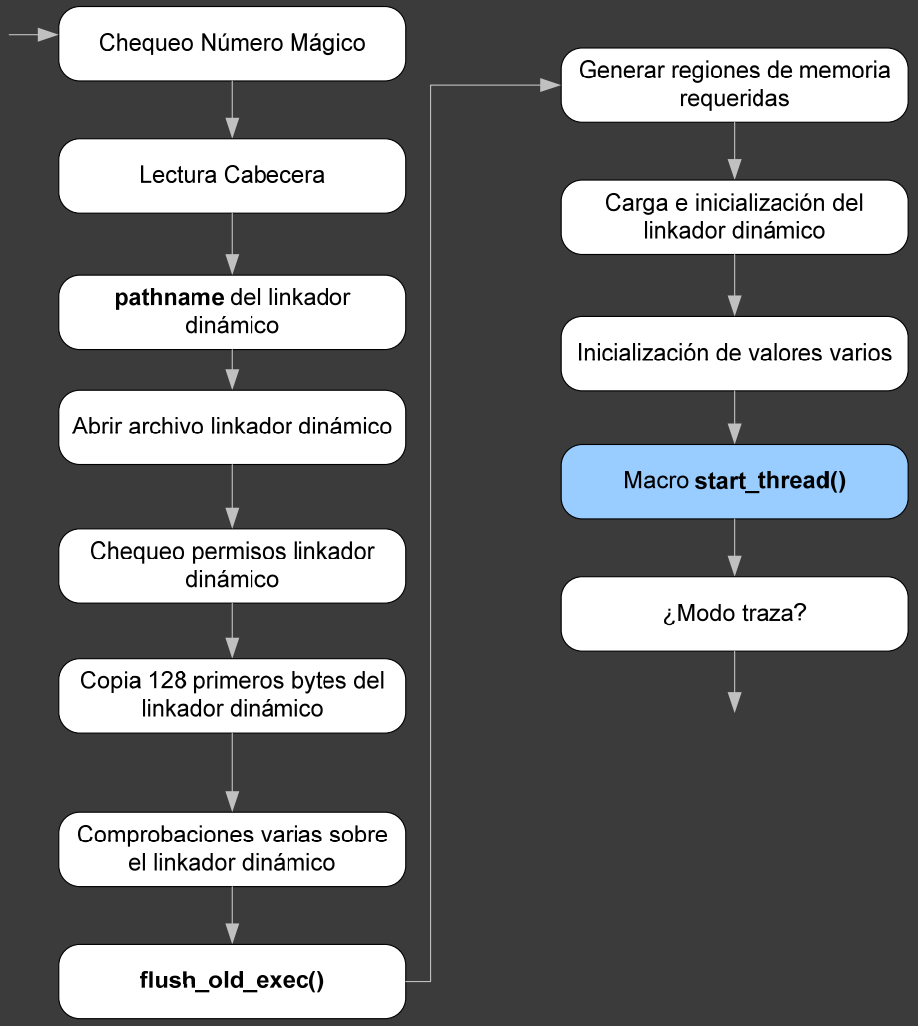
- Carga del linkador dinámico
  - Normalmente en zonas altas de memoria:
    - Evitar colisiones con segmentos del programa
    - Por encima de la 0x40000000
- Inicialización

load_binary()



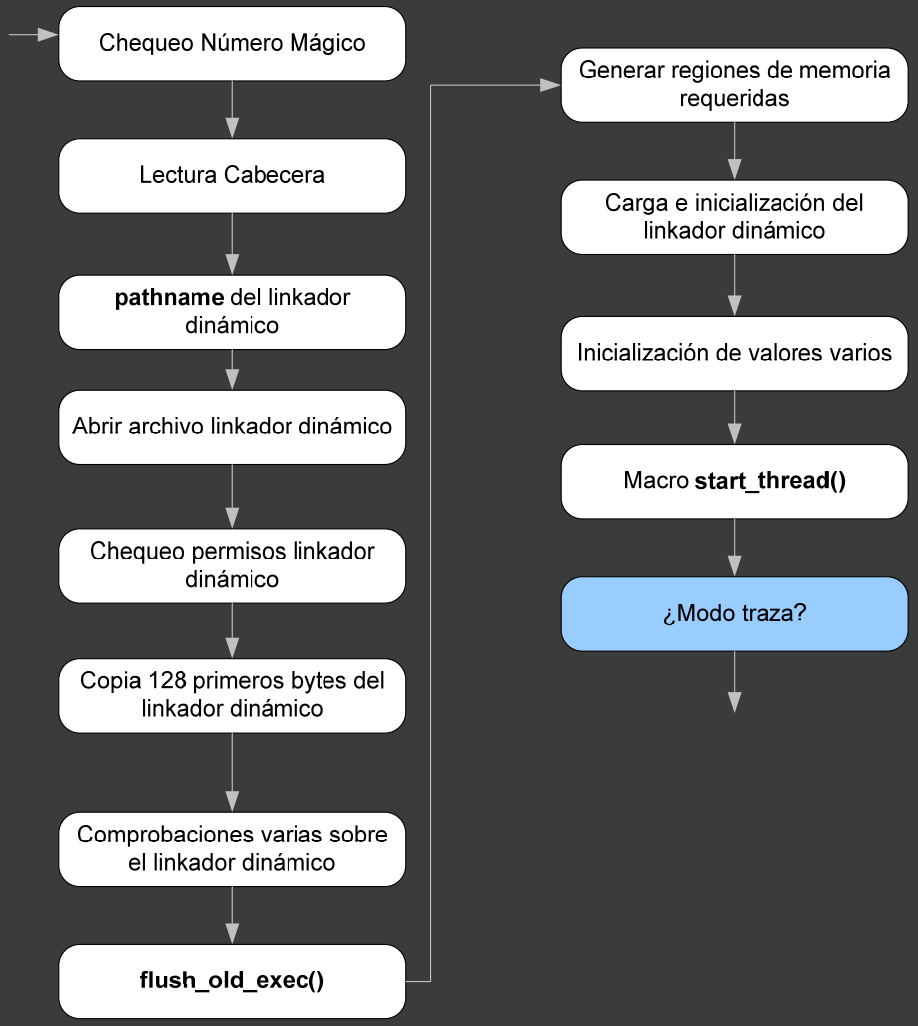
- Inicialización de valores varios
  - start_code y end_code
  - start_data y end_data
  - start_stack
  - …
- Indican dónde se encuentran en memoria las zonas de:
  - Código
  - Datos
  - Pila
  - …

load_binary()



- Llamada a la macro *start_thread()*
- Inicialización de *eip* y *esp*
  - *eip:* Punto de entrada al linkador dinámico
  - esp: Inicio de la pila

load_binary()



- Si estamos en modo traza
  - Avisar al depurador
    - EXEC completado con éxito

Referencias

- Presentación EXEC

  Roberto González Suárez y Néstor Escandell Montesdeoca
  Diseño de Sistemas Operativos

  2007 – 2008

- Understanding the Linux Kernel (3ª Edición)

  Daniel P. Bovet, Marco Cesati

  Ed. O'Reilly

  2005

- Linux Core Kernel Commentary

  Scott Maxwell

  Ed. Coriolis Open Press

  1999

- Linux Cross Reference

  lxr.linux.com