



Gestión de Procesos en Linux

LIN - Curso 2016-2017





El Proceso (I)

- Una de las abstracciones más importantes de Unix/Linux
 - **Conceptualmente:** programa (código objeto almacenado en algún medio) en ejecución / programa activo.
- El SO proporciona dos abstracciones fundamentales:
 - 1 Procesador Virtual (gracias al planificador)
 - Proporciona la ilusión al proceso de que monopoliza el sistema
 - 2 Memoria Virtual
 - Los procesos de usuario y el propio kernel (el procesador) generan direcciones de memoria virtuales, no físicas
 - El HW (MMU) realiza la traducción de direcciones (d. virtual → d. física) bajo la gestión del SO
 - El proceso puede utilizar la memoria como si fuera el “propietario” de toda la memoria del sistema





El Proceso (II)

- Programa en ejecución: no sólo texto, incluye otros “recursos”:
 - Ficheros abiertos
 - Señales pendientes
 - Espacio de direcciones (Mapa de memoria)
 - Uno o más Hilos/Threads de ejecución
 - Cada hilo: PC único, pila, registros del procesador y estado
 - Recursos de sincronización (semáforos, mótexes,...)
 - Datos internos del kernel
- El SO “anota” los recursos asociados a un proceso en el *Bloque de Control de Proceso* (BCP)
 - Estructura que mantiene el SO para cada proceso

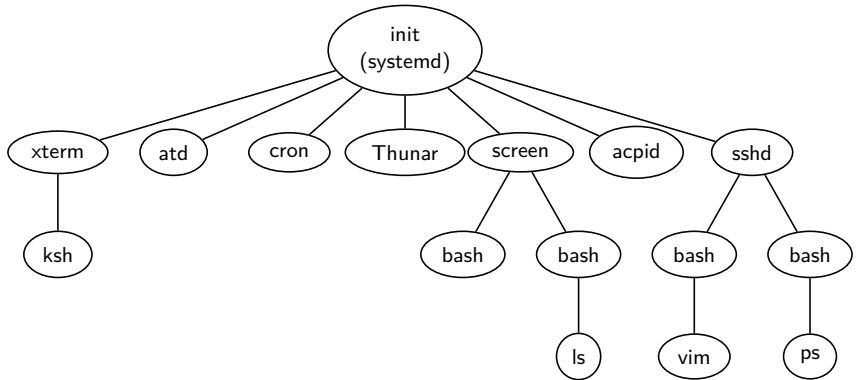


El Proceso (III)

Llamadas al sistema

- `fork()`: La creación de un nuevo proceso se realiza mediante una copia del proceso actual (difiere en el PID, PPID, BCP y algunos recursos y estadísticas).
 - Todos los procesos tienen a `init` (`pid=1`) como ancestro común.
 - En Linux se implementa vía la llamada `clone()`
- `exec*()`: Crear un nuevo espacio de direcciones y cargar un programa en él.
- `exit()`: Termina el proceso y libera todos los recursos asignados a él.
- `wait*()`: Permite que un proceso espere la terminación de un proceso hijo

Árbol de procesos en UNIX/Linux



El árbol de procesos se puede consultar con `pstree`





El Proceso (IV)

- Cuando se crea un proceso mediante `fork()`:
 - Es “casi” idéntico a su padre
 - Recibe una copia (lógica) del espacio de direcciones del padre
 - Ejecuta el mismo código que el padre (comienza en la siguiente instrucción a `fork()`)
- Aunque padre e hijo pueden compartir ciertas páginas (texto/código), tienen copias separadas de stack, bss, ...
 - Los cambios realizados por el padre en stack, bss,... son invisibles al hijo y viceversa





Concepto de hilo o *thread*

- Unidad mínima planificable en una CPU
 - Proceso: Contenedor de recursos (ficheros, señales, memoria, ...)
- Cada hilo:
 - PC único
 - Pila
 - Registros del procesador
 - Estado (bloqueado, listo para ejecutar, en ejecución, ...)
- Los hilos de un mismo proceso comparten un mismo espacio de direcciones (ED)
 - Los “cambios” realizados en el ED por un hilo son visibles a otros hilos



Creación de hilos

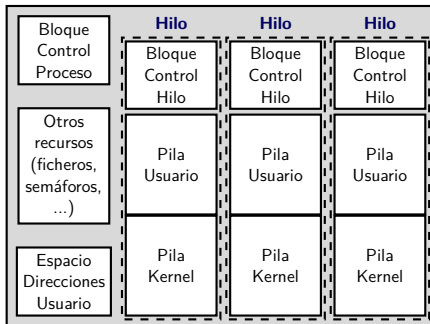
- En Linux es posible crear hilos mediante la biblioteca POSIX Threads
 - `pthread_create()`
 - `pthread_join()`
 - `pthread_exit()`
 - ...
- `pthread_create()` (función de biblioteca) se implementa usando llamada al sistema `clone()`
 - `clone()` permite especificar de forma precisa qué recursos se comparten entre hilo creador y creado
 - `clone()` también permite crear procesos



El Proceso (VI)



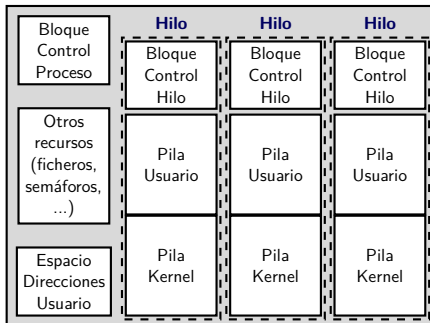
Modelo de proceso multihilo



El Proceso (VI)



Modelo de proceso multihilo



- El kernel de muchos SSOO derivados de UNIX, como Solaris, representa el BCP y el BCH mediante estructuras C distintas
- Ejemplo: `proc_t` y `kthread_t` en Solaris





El Proceso (VII)

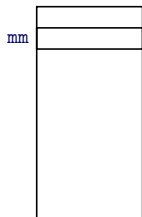
- En el kernel Linux tanto el BCP como el BCH están “descritos” mediante una estructura `task_struct`
 - Hay tantos `task_struct` como hilos a nivel de kernel (KLTs) tenga el proceso
 - Cada `task_struct` tiene un valor distinto para campo `pid`
 - Para que hilos de un mismo proceso compartan recursos, algunos campos tipo puntero tienen el mismo valor en todos los `task_struct` (ej: `mm` apunta a la misma dirección)
 - Para procesos con un solo hilo, el proceso tiene asociado un único `task_struct`, que se comporta como BCP+BCH
- La estructura `task_struct` se emplea también para describir *kernel threads* (hilos especiales del SO)
 - **Ojo:** *kernel threads* != hilos a nivel de kernel (KLTs)



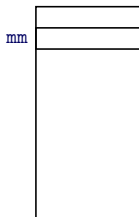
Compartición de recursos entre *threads*



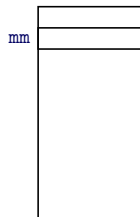
task_struct



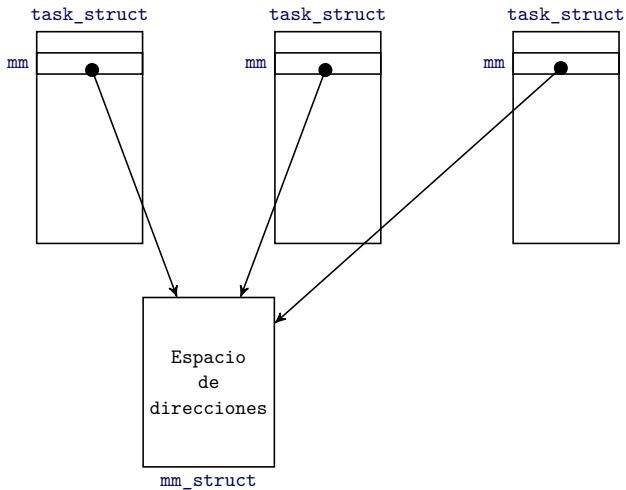
task_struct



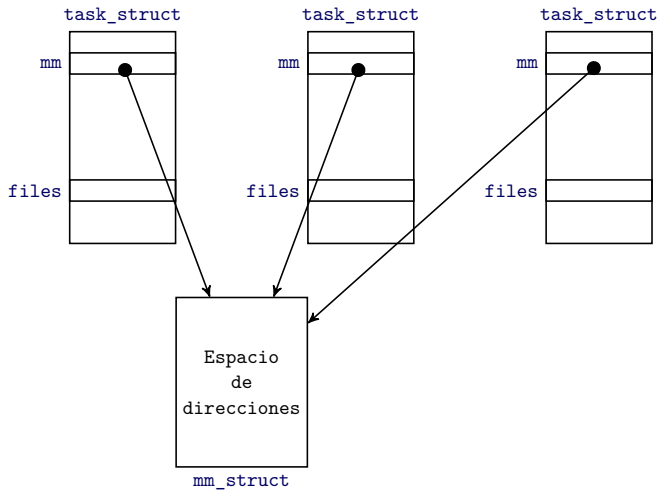
task_struct



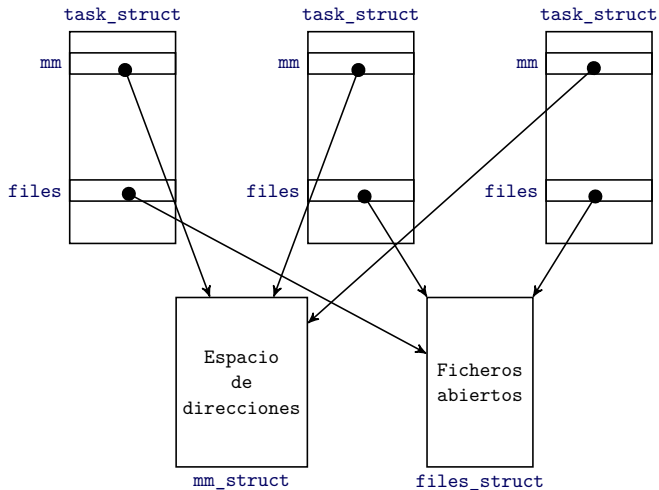
Compartición de recursos entre *threads*



Compartición de recursos entre *threads*



Compartición de recursos entre *threads*





Task Structure (I)

struct task_struct (<linux/sched.h>)

```
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags;
    /* per process flags, defined below */
    ...
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    ...
    pid_t pid;
    ...
}
```





Task Structure (II)

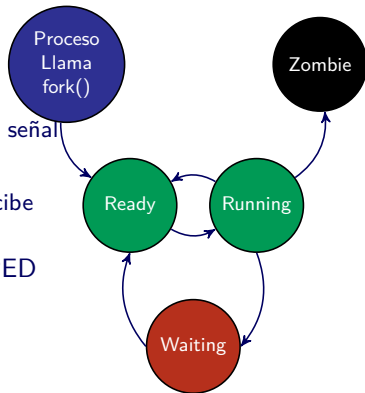
- Información de Estado/Ejecución
 - Estado (state, exit_state)
 - Información de *scheduling* (prioridades)
 - Información temporal (CPU time,..)
 - pid, tgid
 - Punteros a padres, hijos, hermanos, ...
 - ...
- Credenciales
 - Usuario / usuario efectivo ...
- Información sobre recursos
 - Memoria virtual asignada
 - Ficheros abiertos por el proceso
 - Recursos IPC (*Inter-Process Communication*)
 - Señales (Manejadores, señales pendientes, ...)



Estado

■ Estados (state, exit_state):

- TASK_RUNNING (Ready, Running)
- TASK_INTERRUPTIBLE (Waiting)
 - Espera evento. Se “despierta” si recibe señal
- TASK_UNINTERRUPTIBLE (Waiting)
 - Espera evento. No se “despierta” si recibe señal
- __TASK_TRACED / __TASK_STOPPED (Waiting)
 - Se ha detenido la ejecución
- EXIT_ZOMBIE (Zombie)
 - El proceso ha terminado pero no wait4()
- EXIT_DEAD
 - Después de wait(), antes de eliminación completa





__TASK_STOPPED

- La ejecución del proceso ha sido detenida:
 - Señal **SIGSTOP**: para la ejecución del proceso (no se puede capturar ni ignorar)
 - Señal **SIGTSTP**: parado desde el terminal (tty) ^Z (no se puede capturar ni ignorar)
 - Señal **SIGTTIN**: proceso en background requiere entrada
 - Señal **SIGTTOU**: proceso en background requiere salida
- Para que un proceso en TASK_STOPPED continúe requiere la señal **SIGCONT**
 - Es ignorada por los procesos que ya están en TASK_RUNNING
 - Se puede capturar (acción especial)





Identificación Procesos (I)

- En el kernel, la mayor parte de las referencias se hacen vía puntero a `task_struct`
- Por ejemplo, la macro `current` permite obtener puntero al `task_struct` del proceso/hilo que se está ejecutando en la CPU donde se invoca la macro
 - La implementación de esta macro depende de la arquitectura

```
struct task_struct* p=current;  
printk(KERN_INFO "PID of current process=%d\n",p->pid);
```





Identificación Procesos (II)

- Los usuarios suelen identificar los procesos vía PID
 - Se numeran de forma secuencial (habitualmente $\text{nuevoPID} = \text{PID_previo} + 1$)
 - En UNIX se usaba un `short int` (16 bits) para representar el PID en el BCP
 - Límite superior `/proc/sys/kernel/pid_max` (32767 por defecto)
 - Si se supera el límite se reciclan (`pidmap` array)





Grupos de Threads (I)

- El estándar POSIX define el concepto de grupos de *threads*
- En Linux cada hilo se representa mediante un `task_struct`
 - Para identificar el grupo, todos los *threads* de un mismo grupo comparten el mismo valor en el campo `tgid`
 - `tgid`= PID del “group leader”
 - `pid` es el ID del `task_struct`
- En Linux `getpid()` devuelve el valor de `tgid`





Grupos de Threads (II)

- Por defecto, al crear un proceso con `fork()` se crea un grupo de hilos nuevo formado por un único *thread*:
 - `tgid = pid`
- Si creamos una aplicación multi-hilo con POSIX threads en LINUX, todos los hilos creados pertenecen al mismo grupo de *threads* que el hilo *main*
 - `getpid()` devuelve el mismo valor para todos (mismo `tgid`)





Grupos de Threads (III)

- El valor del campo `pid` de todos los hilos de un proceso puede consultarse listando el directorio `/proc/<pid_proceso>/task`
 - Una entrada (subdirectorio) por cada hilo del proceso
 - El nombre del subdirectorio es el valor del campo `pid` del *task struct* correspondiente

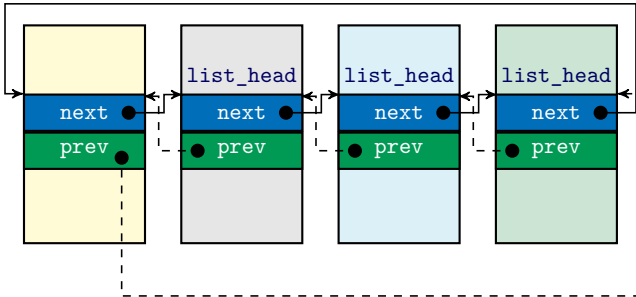
terminal

```
kernel@debian:~$ ps -ef | grep blast
bench      31985 31952 70 16:32 pts/2      00:00:13 ../bin/blastall
bench      31996 31942  0 16:32 pts/1      00:00:00 grep blast
bench@pineta:~$ ls /proc/31985/task/
31985 31991 31992 31993 31994
kernel@debian:~$
```



Implementación task_list (I)

- El kernel mantiene una lista doblemente enlazada con la descripción de todos los “procesos” del sistema - task list -
 - Cada elemento de task list es de tipo struct task_struct
 - task_struct incluye struct list_head tasks
 - list_head: implementación genérica lista doblemente enlazada





Implementación task_list (II)

- Primer elemento: descriptor de tarea init
 - Definido estáticamente en init/init_task.c:
 - `struct task_struct init_task = INIT_TASK(init_task);`

```
/*  
 * INIT_TASK is used to set up the first task table, touch at  
 * your own risk!. Base=0, limit=0x1fffff (=2MB)  
 */  
#define INIT_TASK(tsk) \  
{  
    .state      = 0,  
    .stack      = &init_thread_info,  
    .usage      = ATOMIC_INIT(2),  
    .flags      = PF_KTHREAD,  
    .prio       = MAX_PRIO-20,  
    .static_prio = MAX_PRIO-20,  
    .normal_prio = MAX_PRIO-20,  
    .policy     = SCHED_NORMAL,  
    .cpus_allowed = CPU_MASK_ALL,  
    ...  
}
```





Implementación task_list (III)

```
#define for_each_process(p) \
for (p=&init_task; (p=list_entry((p)->tasks.next, \
struct task_struct, tasks)) \
!= &init_task; )
```

```
struct task_struct *task;
int counter=1; /* for init_task */
for_each_process(task){
    if (task->state==TASK_RUNNING)
        ++counter;
}
```





Parentescos (I)

- task_struct incluye también

```
/*
 * pointers to (original) parent process, youngest child,
 *   younger sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->real_parent->pid)
 */
struct task_struct *real_parent; /* real parent process */
struct task_struct *parent; /* recipient of SIGCHLD, wait4()
    reports*/

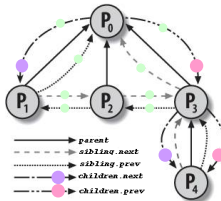
/*
 * children/sibling forms the list of my natural children
 */
struct list_head children; /* list of my children */
struct list_head sibling; /* linkage in my parent's children
    list */
```



Parentescos (II)

- Para iterar por los hijos de un proceso:

```
struct task_struct *task;
struct list_head *list;
list_for_each(list, &current->children)
{
    task = list_entry(list, struct task_struct, sibling);
    /* task now points to one of current's children */
}
```



Creación Procesos



- Unix: Creación de procesos desacoplada combinando `fork()` y `exec()`
 - `spawn()` en otros sistemas
- Potencial sobrecarga/penalización debido a la duplicación del padre. **¿Cómo acelerar la creación?**



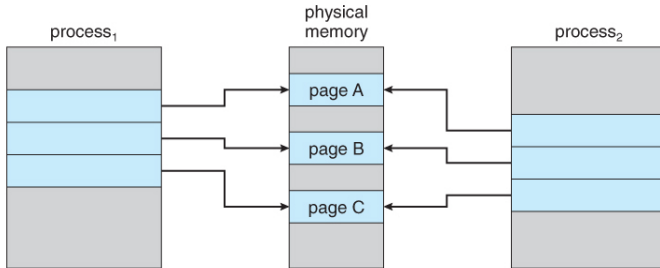


Creación Procesos

- Unix: Creación de procesos desacoplada combinando `fork()` y `exec()`
 - `spawn()` en otros sistemas
- Potencial sobrecarga/penalización debido a la duplicación del padre. **¿Cómo acelerar la creación?**
- **`fork()` con COW – Copy on Write – Pages**
 - En lugar de duplicar inmediatamente las páginas de memoria, se marcan como COW. Sólo se producirá la duplicación en caso de escritura por alguno de los dos procesos (padre/hijo)
 - Sobrecarga inicial:
 - Creación de un nuevo `task_struct`
 - Copia de las tablas de páginas

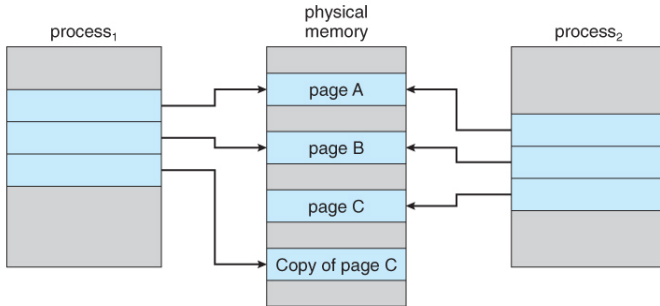


Creación Procesos: Copy-on-Write



Mientras las páginas sean accedidas por padre o hijo en modo lectura, ambos procesos compartirán las páginas

Creación Procesos: Copy-on-Write



Cuando un proceso modifica una página, el SO solicita una nueva página privada para ese proceso



Creación Procesos

■ 3BSD: `vfork()`

- Mismo efecto que `fork()` + COW, aunque adicionalmente:
 - No se copian las tablas de páginas.
 - El padre se bloquea hasta que el hijo `exit()` or `exec()`
- Uso desaconsejado actualmente

■ Linux: `clone()`

- Permite especificar de forma precisa qué recursos se comparten





Creación Procesos

- `fork()`, `vfork()` y `clone()` se implementan vía `do_fork()`
 - `kernel/fork.c`

```
/*  
 * Ok, this is the main fork-routine.  
 *  
 * It copies the process, and if successful kick-starts  
 * it and waits for it to finish using the VM if required.  
 */  
long do_fork(unsigned long clone_flags,  
             unsigned long stack_start,  
             struct pt_regs *regs,  
             unsigned long stack_size,  
             int __user *parent_tidptr,  
             int __user *child_tidptr)
```





Creación Procesos

■ Flags (Propiedades de Duplicación)

Clone Flags

Señal

- Byte Bajo: Señal que se enviará al proceso padre cuando el hijo termine
 - Generalmente **SIGCHLD**
- Bytes Altos (clone flags): Especificación recursos compartidos
 - **CLONE_FILES** → Ficheros abiertos
 - **CLONE_FS** → Información sistema de ficheros
 - **CLONE_IDLETASK** → pid=0 solo para la creación del proceso IDLE
 - **CLONE_NEWNS** → Un nuevo namespace para el hijo
 - **CLONE_PARENT** → Mismo padre
 - **CLONE_SIGHAND** → Manejadores de señal y señales bloqueadas
 - **CLONE_THREAD** → Mismo Thread Group
 - **CLONE_VFORK** → Implementación `vfork()`. El padre bloqueado hasta que el hijo le despierte
 - ...



Creación Procesos



■ Flags (Propiedades de Duplicación)

Clone Flags

Señal

- Byte Bajo: Señal que se enviará al proceso padre cuando el hijo termine
 - Generalmente **SIGCHLD**
- Bytes Altos (clone flags): Especificación recursos compartidos
 - ...
 - **CLONE_PTRACE** → El hijo también se monitoriza
 - **CLONE_STOP** → El proceso comienza en el estado **TASK_STOPPED**
 - **CLONE_VM** → Se comparte el mismo espacio de direcciones
 - ...



Creación Procesos



- Implementación de `fork()`: `SIGCHLD`
- Implementación de `vfork()`: `CLONE_VFORK` | `CLONE_VM` | `SIGCHLD`
- Creación de un hilo : `CLONE_VM` | `CLONE_FS` | `CLONE_FILES` | `CLONE_SIGHAND` | `CLONE_THREAD`





Creación Procesos

- `long do_fork()`
 - `copy_process`
 - Realiza la mayor parte del trabajo
 - Gestión PID local. `copy_process` devuelve pid global, pero la gestión de PID es mas compleja si se ha creado un nuevo *PID namespace*
 - Si `CLONE_PTRACE`, se envía la señal `SIGSTOP` para que el depurador pueda examinar proceso
 - `wake_up_new_task`
 - Se añade la nueva tarea a las colas de ejecución del scheduler
 - Si `CLONE_VFORK` `wait_for_vfork_done`





Creación Procesos

- `struct task_struct *copy_process(...)`
 - Comprobar flags (algunas combinaciones no tienen sentido)
 - `dup_task_struct`
 - Nuevo `kernel_stack`, `thread_info` y `task_struct`
 - Comprobar límites recursos (numero de procesos usuario)
 - Inicializar `task_struct`
 - `sched_fork`
 - Se ajustan parámetros de scheduling
 - Copiar/Compartir componentes
 - `copy_semundo`
 - `copy_files`
 - `copy_fs`
 - `copy_sighand`
 - `copy_signal`
 - `copy_nm`
 - `copy_namespaces`
 - `copy_thread`
 - Establecer IDs, relaciones, etc...





Creación Procesos

- `static int copy_files(unsigned long clone_flags, struct task_struct * tsk)`

```
struct files_struct *oldf, *newf;
int error = 0;
oldf = current->files;
if (!oldf)
    goto out;
if (clone_flags & CLONE_FILES) {
    atomic_inc(&oldf->count);
    goto out;
}
newf = dup_fd(oldf, &error);
if (!newf)
    goto out;
tsk->files = newf;
error = 0;
out: return error;
```



Creación Procesos

- `int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)`

```
struct pt_regs regs;
memset(&regs, 0, sizeof(regs));
regs.bx = (unsigned long) fn;
regs.dx = (unsigned long) arg;
regs.ds = __USER_DS;
regs.es = __USER_DS;
regs.fs = __KERNEL_PERCPU;
regs.gs = __KERNEL_STACK_CANARY;
regs.orig_ax = -1;
regs.ip = (unsigned long) kernel_thread_helper;
regs.cs = __KERNEL_CS | get_kernel_rpl();
regs.flags = X86_EFLAGS_IF | X86_EFLAGS_SF | X86_EFLAGS_PF | 0x2;
return do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, &regs, 0, NULL,
              NULL);
```





Referencias

- Linux Kernel Development
 - Cap. 3 *“Process Management”*
- Professional Linux Kernel Architecture
 - Cap. 2 *“Process Management and Scheduling”*
- IBM Developer Works
 - Anatomy of Linux process management





LIN - Gestión de Procesos en Linux Versión 0.5

©J.C. Sáez, M. Prieto

*This work is licensed under the Creative Commons **Attribution-Share Alike 3.0 Spain License**. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/es/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.*

*Esta obra está bajo una licencia **Reconocimiento-Compartir Bajo La Misma Licencia 3.0 España de Creative Commons**. Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-sa/3.0/es/> o envíe una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.*

Este documento (o uno muy similar) está disponible en <https://cv4.ucm.es/moodle/course/view.php?id=75410>

