TEMA 2. PROCESOS E HILOS

Nos basamos el kernel 2.6 de linux.

Podemos descargar los fuentes de www.kernel.org.

Nomenclatura usada para las rutas de archivos : aludiremos los archivos fuente expresando su ruta relativa desde el directorio donde hemos descargado todos los fuentes.

Bibliografía:

- [Sta05] W. Stallings, Sistemas Operativos. Aspectos Internos y Principios de Diseño (5/e), Prentice Hall, 2005.
- [Car07] J. Carretero et al., Sistemas Operativos (2ª Edición), McGraw-Hill, 2007.
- [Lov10] R. Love, *Linux Kernel Development (3/e)*, Addison-Wesley Professional, 2010.
- [Lov03] R. Love, *The Linux Process Scheduler,*
- *http://www.informit.com/articles/printerfriendly.aspx?p=101760*
- [Mau08] W. Mauerer, Professional Linux Kernel Architecture, Wiley, 2008.

1. GENERALIDADES SOBRE PROCESOS, HILOS Y PLANIFICACIÓN

Nos situamos en la teoría general sobre procesos y planificación, justo tras lo visto en la asignatura de Fundamentos del Software; no incluimos por tanto lecturas sobre conceptos vistos allí.

LECTURAS PARA CUBRIR LOS CONTENIDOS DE ESTE PUNTO:

Niveles de planificación; planificador a largo, medio y corto plazo: [Sta05 9.1]

Sobre la planificación a corto plazo, incluir de [Sta05 9.2] los siguientes puntos:

> Criterios de la planificación a corto plazo; uso de prioridades; planificación apropiativa y no apropiativa; algoritmo FCFS (Primero En Llegar, Primero en Servirse); algoritmo round robin; algoritmo el más corto primero; algoritmo de el menor tiempo restante; colas múltiples con retroalimentación; planificación de contribución justa (Fair-Share Scheduling).

Operaciones genéricas de creación y destrucción de procesos [Sta05 pags 114-116 y 137] (Solo para los conceptos de "Creación de procesos" y "Terminación de procesos")

Operaciones sobre procesos en Unix :

> Llamada al sistema fork: [Car07 pags 118-119]; Llamada al sistema exec [Car07 pags 121-123]
>
> Llamada al sistema exit: [Car07 pags 125-126]; Llamada al sistema wait: [Car07 pags 126-127]

2. DISEÑO E IMPLEMENTACION DE PROCESOS EN LINUX

Nota sobre términos:

En el entorno Linux otro nombre para proceso es tarea (task);

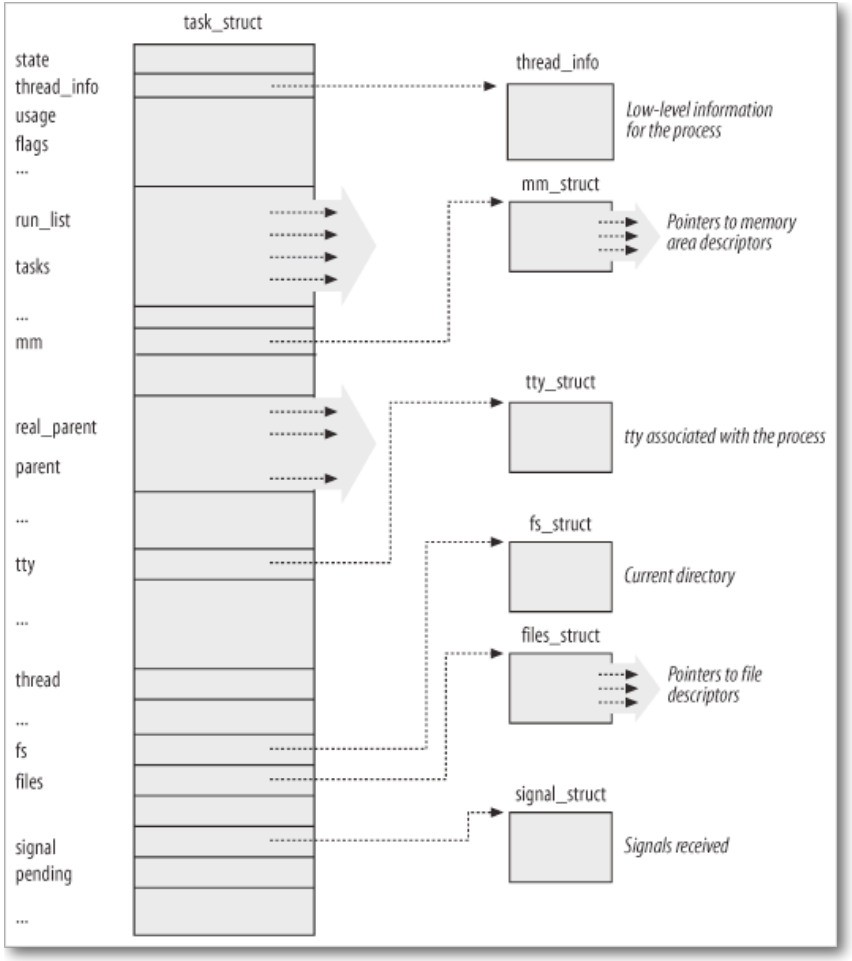
Usaremos como equivalentes los términos proceso y tarea.

Internamente, el kernel de Linux usa la palabra tarea para referirse a un proceso.

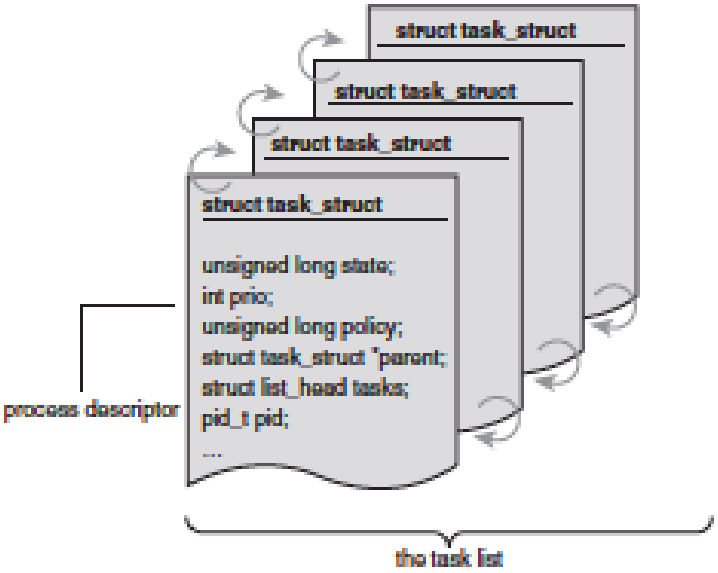
Cada proceso puede tener varias hebras (threads).

2.1 Representación de los procesos [Mau08 2.3]

En Linux, un proceso es representado por estas dos estructuras: el PCB que es una estructura del tipo `struct task_struct` y una estructura del tipo `struct thread_info`.


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

El kernel almacena la lista de procesos como una lista circular doblemente enlazada llamada lista de tareas (task list):



Cada elemento en la task list es un descriptor de proceso de tipo struct task_struct (definido en `</include/linux/sched.h>`):

```
struct task_struct { /// del kernel 2.6.24
	volatile long state;	/* -1 unrunnable, 0 runnable, >0 stopped */
	void *stack;
	atomic_t usage;
	unsigned int flags;	/* per process flags, defined below */
	unsigned int ptrace;

	int lock_depth;		/* BKL lock depth */

#ifdef CONFIG_SMP
#ifdef __ARCH_WANT_UNLOCKED_CTXSW
	int oncpu;
#endif
#endif

	int prio, static_prio, normal_prio;
	struct list_head run_list;
	const struct sched_class *sched_class;
	struct sched_entity se;

#ifdef CONFIG_PREEMPT_NOTIFIERS
	/* list of struct preempt_notifier: */
	struct hlist_head preempt_notifiers;
#endif

	unsigned short ioprio;
	/*
	 * fpu_counter contains the number of consecutive context switches
	 * that the FPU is used. If this is over a threshold, the lazy fpu
	 * saving becomes unlazy to save the trap. This is an unsigned char
	 * so that after 256 times the counter wraps and the behavior turns
	 * lazy again; this to deal with bursty apps that only use FPU for
```

```
         * a short time
         */
        unsigned char fpu_counter;
        s8 oomkilladj; /* OOM kill score adjustment (bit shift). */
#ifdef CONFIG_BLK_DEV_IO_TRACE
        unsigned int btrace_seq;
#endif
        unsigned int policy;
        cpumask_t cpus_allowed;
        unsigned int time_slice;

#if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
        struct sched_info sched_info;
#endif
        struct list_head tasks;
        /*
         * ptrace_list/ptrace_children forms the list of my children
         * that were stolen by a ptracer.
         */
        struct list_head ptrace_children;
        struct list_head ptrace_list;

        struct mm_struct *mm, *active_mm;

/* task state */
        struct linux_binfmt *binfmt;
        int exit_state;
        int exit_code, exit_signal;
        int pdeath_signal;  /*  The signal sent when the parent dies  */
        /* ??? */
        unsigned int personality;
        unsigned did_exec:1;
```

```
        pid_t pid;
        pid_t tgid;

#ifdef CONFIG_CC_STACKPROTECTOR
        /* Canary value for the -fstack-protector gcc feature */
        unsigned long stack_canary;
#endif
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->parent->pid)
         */
        struct task_struct *real_parent; /* real parent process (when being debugged) */
        struct task_struct *parent;  /* parent process */
        /*
         * children/sibling forms the list of my children plus the
         * tasks I'm ptracing.
         */
        struct list_head children;   /* list of my children */
        struct list_head sibling;    /* linkage in my parent's children list */
        struct task_struct *group_leader;  /* threadgroup leader */

        /* PID/PID hash table linkage. */
        struct pid_link pids[PIDTYPE_MAX];
        struct list_head thread_group;

        struct completion *vfork_done;          /* for vfork() */
        int __user *set_child_tid;          /* CLONE_CHILD_SETTID */
        int __user *clear_child_tid;        /* CLONE_CHILD_CLEARTID */

        unsigned int rt_priority;
        cputime_t utime, stime, utimescaled, stimescaled;
```

```
        cputime_t gtime;
        cputime_t prev_utime, prev_stime;
        unsigned long nvcsw, nivcsw; /* context switch counts */
        struct timespec start_time;         /* monotonic time */
        struct timespec real_start_time;    /* boot based time */
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-
specific */
        unsigned long min_flt, maj_flt;

        cputime_t it_prof_expires, it_virt_expires;
        unsigned long long it_sched_expires;
        struct list_head cpu_timers[3];

/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        struct group_info *group_info;
        kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
        unsigned keep_capabilities:1;
        struct user_struct *user;
#ifdef CONFIG_KEYS
        struct key *request_key_auth;       /* assumed request_key authority */
        struct key *thread_keyring;  /* keyring private to this thread */
        unsigned char jit_keyring;   /* default keyring to attach requested keys to */
#endif
        char comm[TASK_COMM_LEN]; /* executable name excluding path
                                  - access with [gs]et_task_comm (which lock
                                    it with task_lock())
                                  - initialized normally by flush_old_exec */
/* file system info */
        int link_count, total_link_count;
#ifdef CONFIG_SYSVIPC
```

```
/* ipc stuff */
        struct sysv_sem sysvsem;
#endif
/* CPU-specific state of this task */
        struct thread_struct thread;
/* filesystem information */
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
/* namespaces */
        struct nsproxy *nsproxy;
/* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;

        sigset_t blocked, real_blocked;
        sigset_t saved_sigmask;            /* To be restored with TIF_RESTORE_SIGMASK */
        struct sigpending pending;

        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;
#ifdef CONFIG_SECURITY
        void *security;
#endif
        struct audit_context *audit_context;
        seccomp_t seccomp;

/* Thread group tracking */
        u32 parent_exec_id;
```

```
        u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
        spinlock_t alloc_lock;

        /* Protection of the PI data structures: */
        spinlock_t pi_lock;

#ifdef CONFIG_RT_MUTEXES
        /* PI waiters blocked on a rt_mutex held by this task */
        struct plist_head pi_waiters;
        /* Deadlock detection and priority inheritance handling */
        struct rt_mutex_waiter *pi_blocked_on;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
        /* mutex deadlock detection */
        struct mutex_waiter *blocked_on;
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
        unsigned int irq_events;
        int hardirqs_enabled;
        unsigned long hardirq_enable_ip;
        unsigned int hardirq_enable_event;
        unsigned long hardirq_disable_ip;
        unsigned int hardirq_disable_event;
        int softirqs_enabled;
        unsigned long softirq_disable_ip;
        unsigned int softirq_disable_event;
        unsigned long softirq_enable_ip;
        unsigned int softirq_enable_event;
        int hardirq_context;
        int softirq_context;
```

```
#endif
#ifdef CONFIG_LOCKDEP
# define MAX_LOCK_DEPTH 30UL
        u64 curr_chain_key;
        int lockdep_depth;
        struct held_lock held_locks[MAX_LOCK_DEPTH];
        unsigned int lockdep_recursion;
#endif

/* journalling filesystem info */
        void *journal_info;

/* stacked block device info */
        struct bio *bio_list, **bio_tail;

/* VM state */
        struct reclaim_state *reclaim_state;

        struct backing_dev_info *backing_dev_info;

        struct io_context *io_context;

        unsigned long ptrace_message;
        siginfo_t *last_siginfo; /* For ptrace use.  */
...}
```

ALGUNOS CONTENIDOS DE task_struct:

* estado e información de ejecución tales como señales pendientes, Pid, puntero al padre y a otros procesos relacionados (se verá más adelante), prioridades, información sobre el tiempo de CPU..
* Información sobre asignación de memoria
* Credenciales del proceso como identificativos de usuario y de grupo
* Archivos usados
* Información sobre comunicación entre procesos
* Manejadores de señales usados por el proceso para responder a las señales

Dentro del kernel las tareas son referenciadas mediante un puntero a su task_struct.

La macro *current* proporciona un puntero al descriptor de proceso de la tarea que se está ejecutando actualmente.

2.2 Estados de un proceso [Mau08 2.3]

La variable *state* de task_struct especifica el estado actual del proceso.

Valores: (son constantes que resuelve el preprocesador definidas en <sched.h>)

TASK_RUNNING: proceso ejecutable, tanto si está actualmente ejecutándose o está esperando a que se le asigne CPU.

Si el proceso se está ejecutando puede estar tanto en ejecución en el espacio de usuario como en el espacio del kernel.

TASK_INTERRUPTIBLE: proceso bloqueado o durmiendo;

la tarea no está lista para ejecutarse porque espera un evento;

cuando el kernel notifique al proceso (mediante una señal) que el evento ha ocurrido se calificará en el estado `TASK_RUNNING` y podrá reanudar su ejecución cuando el planificador le asigne la CPU

TASK_UNINTERRUPTIBLE: estado idéntico al anterior excepto que el proceso no despierta si recibe una señal;

TASK_STOPPED: proceso parado o detenido, no se está ejecutando ni es elegible para ejecutarse. Ocurre cuando la tarea recibe señales como SIGSTOP o ciertas señales de depuración.

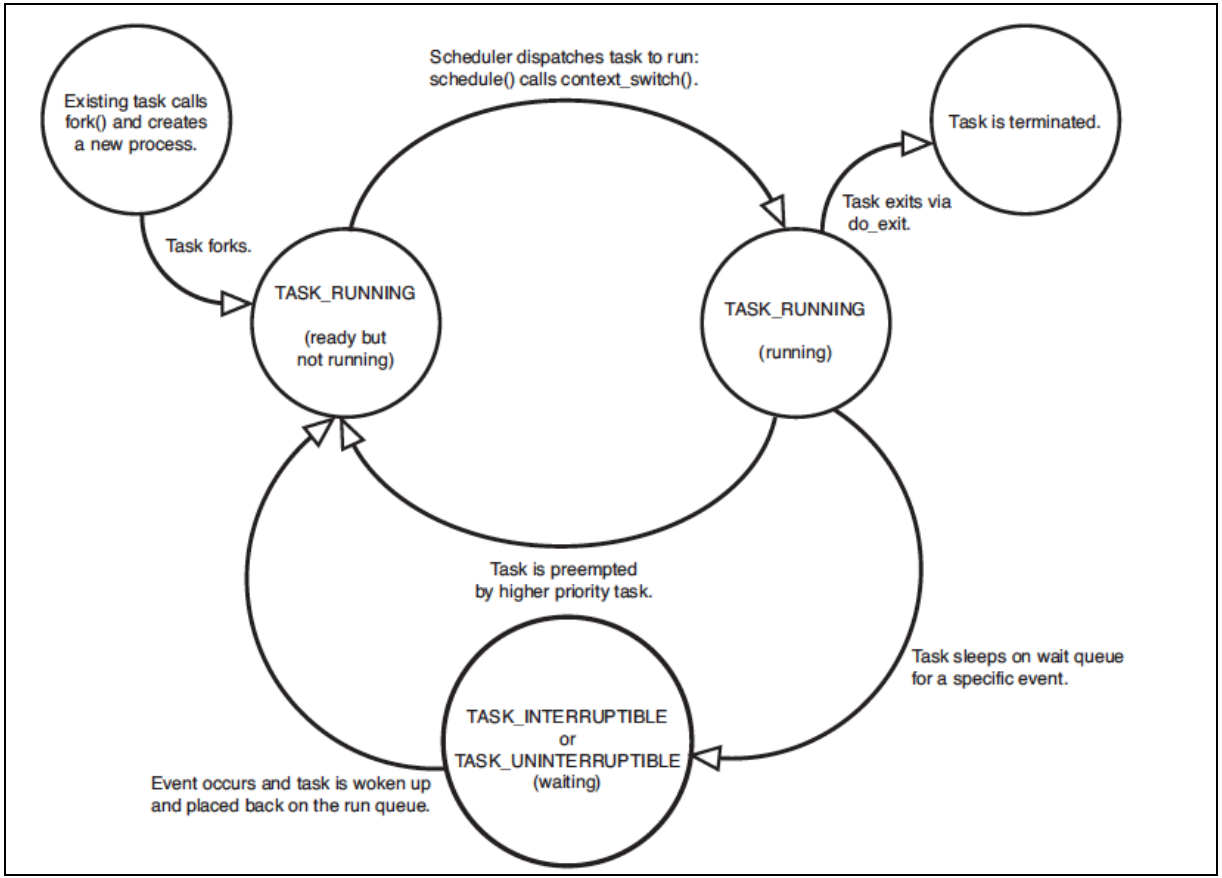
TASK_TRACED: el proceso está siendo traceado por otro proceso.

EXIT_ZOMBIE: una tarea está en estado zombie cuando ha terminado pero todavía el padre no ha tomado su valor de retorno,

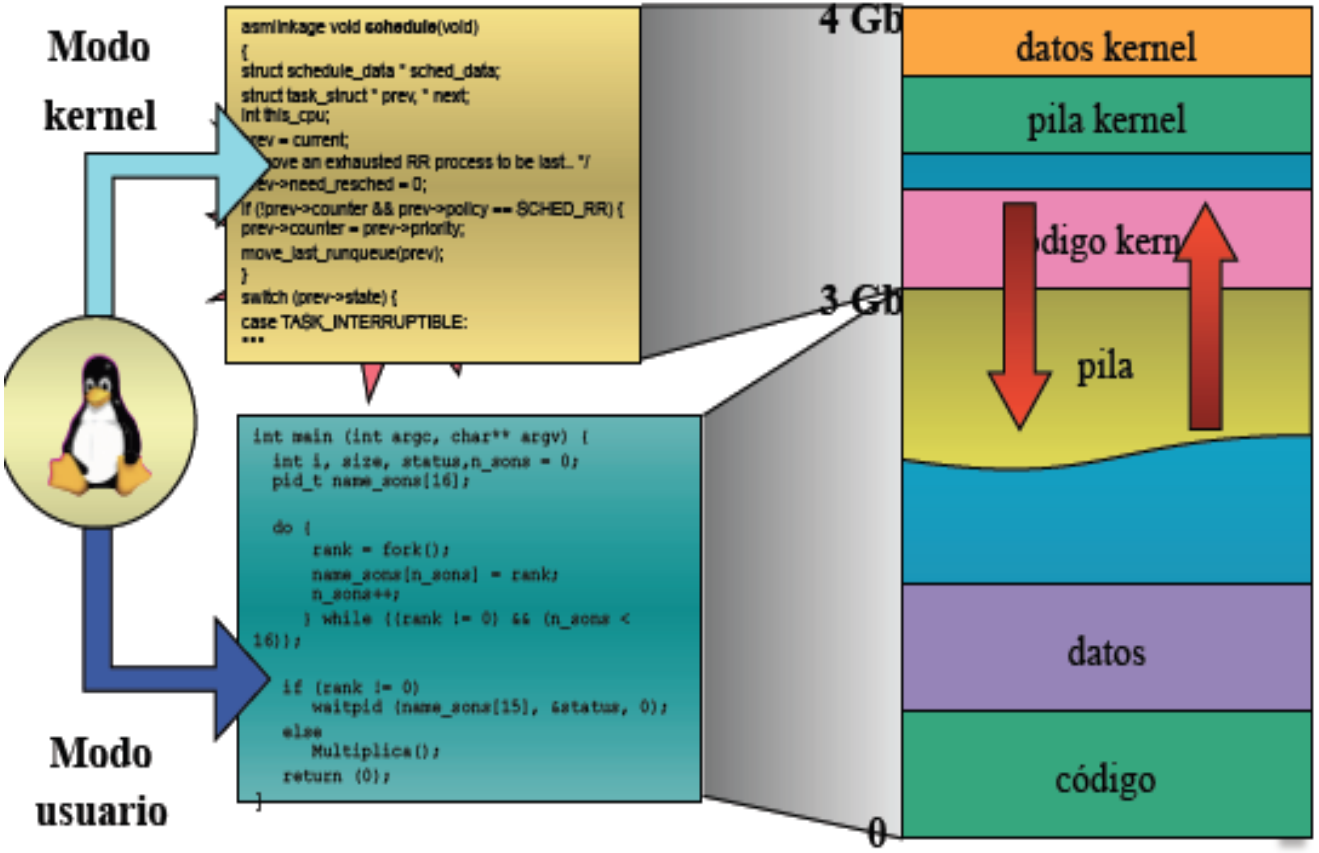
éste debe permanecer almacenado hasta que el padre lo recoja con un wait, por tanto no se puede destruir todavía su task_struct.

EXIT_DEAD: es el estado al que pasa un proceso tras la llamada al sistema wait, hasta que sea completamente eliminado del sistema.

Los valores EXIT_ZOMBIE y EXIT_DEAD pueden ser usados tanto en el campo state de task_struct como en el campo exit_state, que es específico de aquellos procesos que están terminando


Existing task calls fork() and creates a new process.
Task forks.
TASK_RUNNING (ready but not running)
Scheduler dispatches task to run: schedule() calls context_switch().
TASK_RUNNING (running)
Task exits via do_exit.
Task is terminated.
Task is preempted by higher priority task.
Task sleeps on wait queue for a specific event.
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
Event occurs and task is woken up and placed back on the run queue.

2.3 Estructura interna de un proceso en linux



Pila: zona del espacio de direcciones lógicas de un proceso que se utiliza para gestionar las llamadas a función que se efectúa en el código del programa.

Está formada por un conjunto de capas con estructura LIFO.

Cada vez que se realiza una llamada a una función se crea un marco nuevo en la pila que contiene...

las variables locales de la pila,

sus parámetros actuales,

una dirección de retorno,

y la dirección del marco anterior para poder eliminar éste.

Cada vez que una función retorna se elimina el marco actual.

Como el proceso puede estar trabajando en dos modos (usuario o kernel), habrá una pila para cada modo.

2.4 Contexto de un proceso [Lov10 pag. 29]

* La ejecución usual de un proceso ocurre en el espacio de usuario (user-space);

--- Si se produce una llamada al sistema, o se genera una excepción o interrupción,

.....entra en el espacio del kernel (kernel-space)

....y se dice entonces que el kernel "se está ejecutando en el contexto del proceso", o que está en el contexto del proceso.

--- Cuando se termina dicha llamada o tratamiento de excepción el proceso reanuda su ejecución en su espacio de usuario (a no ser que un proceso de más alta prioridad se haya convertido en ejecutable en ese espacio de tiempo en cuyo caso el planificador lo elegirá para asignarle CPU).

* En resumen, se pasa a ejecutar código del kernel cuando ocurre alguno de estos acontecimientos:

una interrupción o excepción,

se hace una llamada al sistema,

o se cambia la asignación de la CPU de un proceso a otro.

Observemos que los acontecimientos anteriores pueden presentarse de forma anidada,

..... por tanto el kernel deberá salvar y restaurar la información necesaria para que esto se resuelva adecuadamente.

Se consigue así que haya simultáneamente más de una ejecución de código kernel, consiguiéndose la interesante característica de **kernel reentrante**.

2.5 El árbol de procesos [Lov10 pag 29-30]

* En todos los entornos Unix hay una jerarquía de procesos definida como sigue:

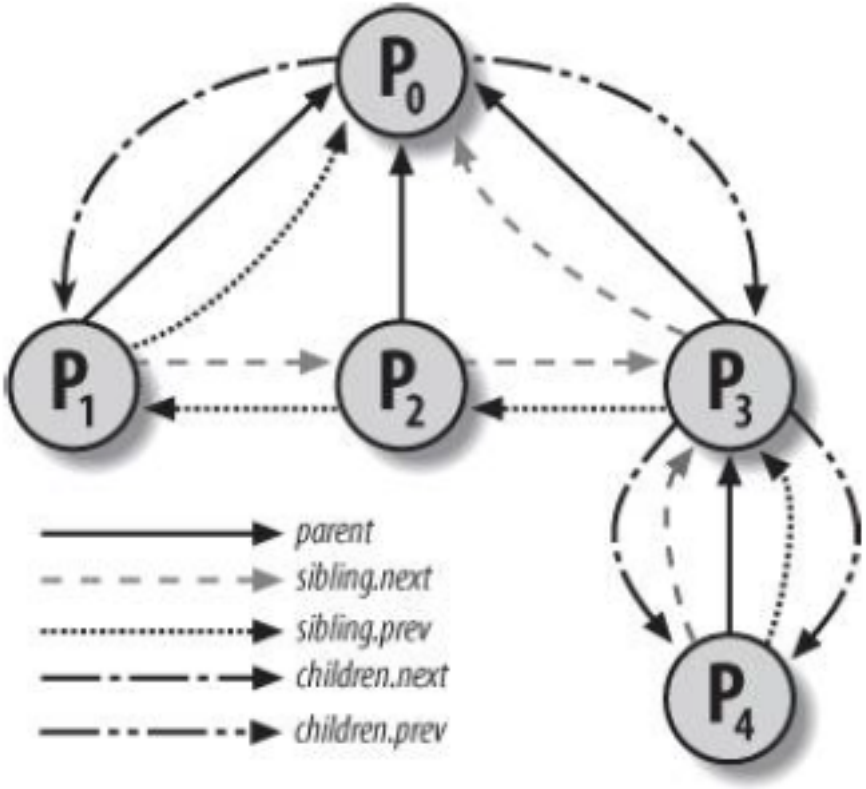
Si el proceso P0 hace una llamada la sistema fork y así genera el proceso P1, se dice que P0 es el proceso padre y P1 es el hijo.

Si el proceso P0 hace varios fork general de varios proceso hijos P1,P2,P3, la relación entre ellos es de hermanos (sibling)

* Todos los procesos son descendientes del proceso init (cuyo pid es 1);

El kernel comienza init en el último paso del proceso de **arranque del sistema**.

El proceso **init lanza a los demás procesos** completando el proceso de arranque.


P0
P1
P2
P3
P4
parent
sibling.next
sibling.prev
children.next
children.prev

* **Cada task_struct tiene un puntero ...**

a la task_struct de su padre:

```
struct task_struct *my_parent
```

a una lista de hijos (llamada children):

```
struct list_head children;
      /*apunta a la cabeza a lista de mis hijos*/
```

y a una lista de sus hermanos (llamada sibling):

```
struct list_head sibling
      /* ...a la lista de hijos de mi padre */
```

El kernel dispone de procedimientos eficaces para las acciones usuales de manipulación de la lista. (Para una explicación detallada de list_head ver [Mau08 1.3.13])

* Dado el proceso actual, es posible obtener el descriptor de proceso de su padre mediante este código:

```
struct task_struct *my_parent = current->parent;
```

* El descriptor de proceso del proceso init está almacenado estáticamente como init_task.

* El código siguiente pone de manifiesto la relación existente entre cualquier proceso y el init:

```
struct task_struct *task;
for (task = current; task != &init_task; task = task->parent)
;
/* task now points to init */
```

* Recorrer todos los procesos en el sistema es fácil porque la lista de tareas es una lista circular doblemente enlazada

* Se proporciona la macro for_each_process(task) que recorre la lista completa de tareas y en cada iteración task apunta a la siguiente tarea en la lista:

```
struct task_struct *task;
for_each_process(task) {
/* this pointlessly prints the name and PID of each task */
printk("%s[%d]\n", task->comm, task->pid);     }
```

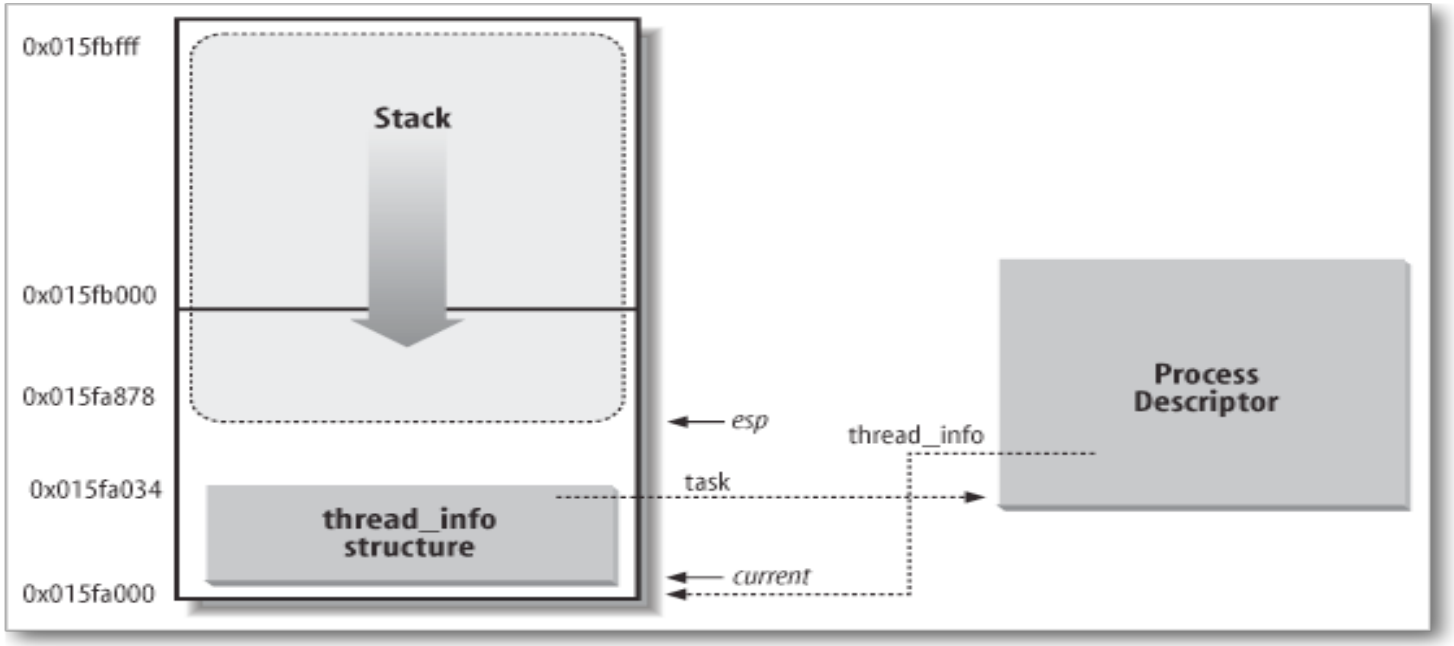
2.6 La estructura thread_info

* Contiene la información de bajo nivel del proceso, acedida por código ensamblador dependiente de la arquitectura.

* Forma parte de una estructura del tipo union thread_union (include/linux/sched.h) donde está también y la pila kernel de dicho proceso:

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

La figura siguiente muestra la relación entre task_struct, thread_info y la pila kernel:



* Cuando algún componente del kernel utilizada demasiado espacio de pila,

la pila kernel podría pisar la información de thread_info;

para evitar eso el kernel tiene una función para determinar si una determinada dirección está dentro de una porción válida de la pila o no.

2.7 Implementación de hilos en linux [Lov10 pag33]

Creación de hebras

* Desde el punto de vista del kernel no hay distinción entre hebra y proceso;

Linux implementa el concepto de hebra como un proceso sin más, que simplemente comparte recursos con otros procesos.

* Cada hebra tiene su propia task_struct.

* La llamada al sistema **clone** crea un nuevo proceso o hebra;

* La figura siguiente muestra los flags de control que podemos pasar a clone;
podemos especificar cómo deseamos que se comporte el nuevo proceso y cómo se se quiere que sea la compartición de recursos entre padre e hijo:

Flag	Meaning
CLONE_FILES	Parent and child share open files.
CLONE_FS	Parent and child share filesystem information.
CLONE_IDLETASK	Set PID to zero (used only by the idle tasks).
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Child is to have same parent as its parent.
CLONE_PTRACE	Continue tracing child.
CLONE_SETTID	Write the TID back to user-space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Parent and child share signal handlers and blocked signals.
CLONE_SYSVSEM	Parent and child share System V SEM_UNDO semantics.
CLONE_THREAD	Parent and child are in the same thread group.
CLONE_VFORK	vfork() was used and the parent will sleep until the child wakes it.
CLONE_UNTRACED	Do not let the tracing process force CLONE_PTRACE on the child.
CLONE_STOP	Start process in the TASK_STOPPED state.
CLONE_SETTLS	Create a new TLS (thread-local storage) for the child.
CLONE_CHILD_CLEARTID	Clear the TID in the child.

Hebras kernel [Lov10 pag 35]

* A veces es útil que el kernel realiza algunas operaciones en segundo plano, para lo cual se crean hebras kernel (procesos que existen únicamente en el espacio del kernel).

* La principal diferencia entre hebras kernel y procesos normales es que las hebras kernel no tienen un espacio de direcciones (su puntero mm es NULL), se ejecutan únicamente en el espacio del kernel.

* Por lo demás son planificadas y podrían ser expropiadas, como procesos normales.

* Se crean por el kernel al levantar el sistema, mediante una llamada a clone().

* Y como todo proceso, termina cuando realizan una operación do_exit o cuando otra parte del kernel provoca su finalización

2.8 Ejecutando llamadas al sistema para gestión de procesos

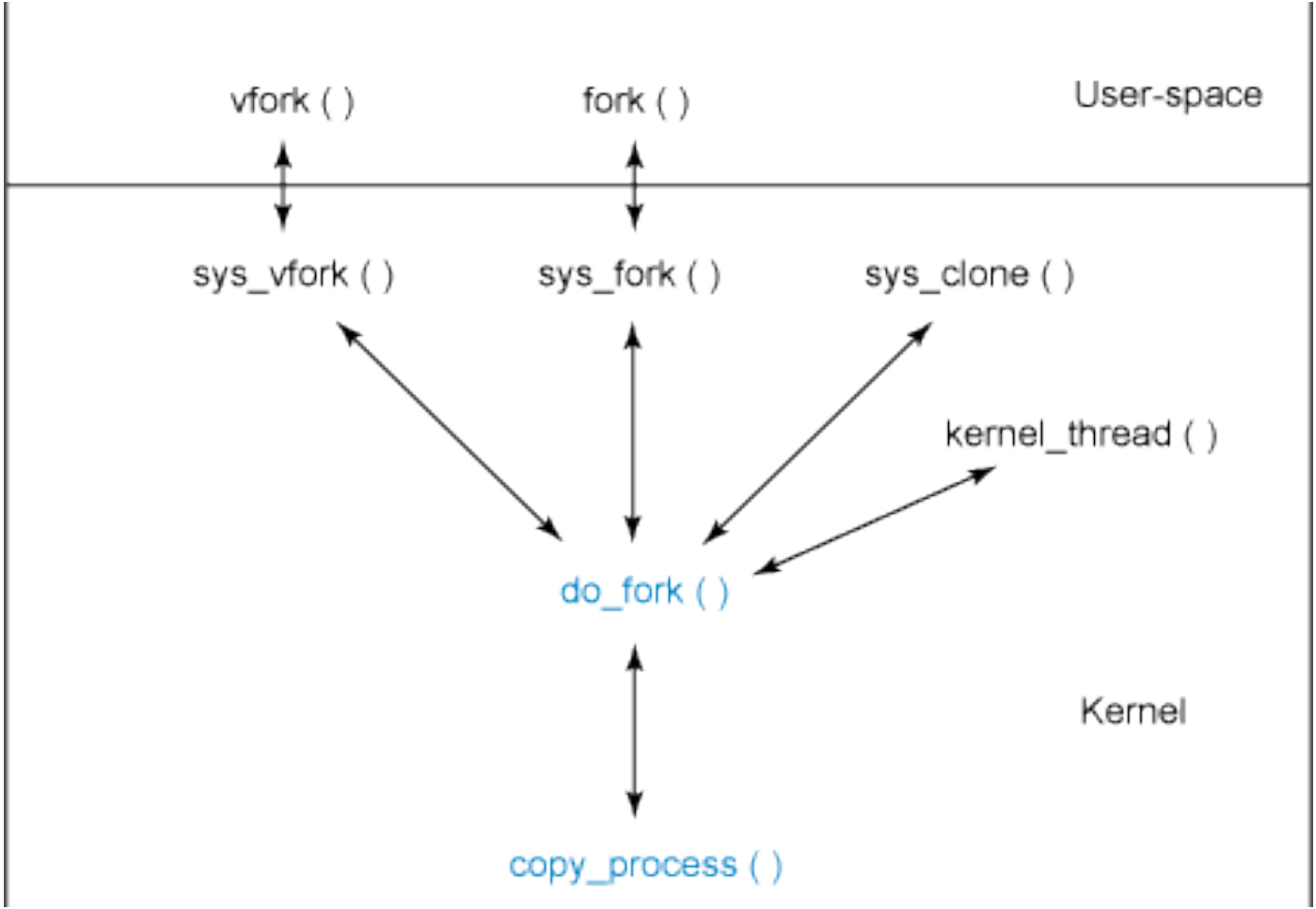
* Nos centramos en las llamadas al sistema para gestión de procesos como fork, vfork, y clone.

* Normalmente estas llamadas se invocan a través de las librerías de C que realizan la comunicación con el kernel (los métodos para cambiar de modo usuario a modo kernel varían de una arquitectura a otra).

* El punto de entrada para fork, vfork y clone son las funciones sys_fork, sys_vfork y sys-clone;

* su implementación es dependiente de la arquitectura puesto que la forma en que los parámetros se pasan entre el espacio de usuario y el espacio del kernel son diferentes en las distintas arquitecturas.

La labor de las anteriores funciones es extraer la información suministrada en el espacio de usuario (parámetros de la llamada) e invocar a la función do_fork (independiente de la arquitectura) que es quien realiza la duplicación de procesos.


vfork ( )
fork ( )
User-space
sys_vfork ( )
sys_fork ( )
sys_clone ( )
kernel_thread ( )
do_fork ( )
Kernel
copy_process ( )

2.9 Creación de procesos con fork [Lov10 pag 32]

* Situémonos en el espacio de usuario; se produce una llamada a alguna de las rutina de librería para crear un nuevo proceso como fork(), vfork() o clone();

Dado que todas ellas, básicamente, realizan la misma función que es crear un nuevo proceso (aunque varían en las características de éste), a grandes rasgos ocurre la misma secuencia de llamadas:

.... se transfiere el control a la función do_fork() del kernel (definida en <kernel/fork.c>)

... que a su vez llama a la función copy_process(), que realiza en sí la creación del nuevo proceso

tras el fin de copy_process, do_fork provocará que el nuevo hijo se ejecute.

Actuación de copy_process()

1. Se crea una nueva pila kernel, la estructura thread_info y la task_struct para el nuevo proceso con los valores de la tarea actual.

2. Para los elementos de task_struct del hijo que deban tener valores distintos a los del padre, se les dan los valores iniciales correctos (como por ejemplo datos para estadísticas).

3. Se establece el estado del hijo a TASK_UNINTERRUPTIBLE mientras se realizan las restantes acciones.

4. Se establecen valores adecuados para los flags de la task_struct del hijo:

   pone a 0 el flag PF_SUPERPRIV (indica si la tarea usa privilegio de superusiario)

   pone a 1 el flag PF_FORKNOEXEC (que indica si el proceso ha hecho fork pero no exec)

5. Se llama a alloc_pid() para asignar un PID a la nueva tarea.

6. Según cuáles sean los flags pasados a clone(), copy_process() duplica o comparte recursos como archivos abiertos, información de sistemas de archivos, manejadores de señales, espacio de direccionamiento del proceso.....

Normalmente estos recursos son compartidos entre tareas de un mismo proceso (contrario al caso de que se creen nuevos recursos para el hijo con los valores iniciales que tenía del padre).

7. Finalmente copy_process() termina devolviendo un puntero a la task_struct del hijo.

Copy-on-Write [Lov10 pag 31]

*Con esta técnica, en la creación de un nuevo proceso, al hijo no se le asigna nuevo espacio de memoria sino que las páginas del padre resultan ahora compartidas por ambos, padre e hijo.

* A cada páginas se le asigna un bit llamado copy-on-write (cow) con valor 1;

* cow=1 significa página compartida por varios procesos:

-- las operaciones de lectura están permitidas

-- cuando un proceso intente escribir se producirá una excepción por "violación de protección"; el sistema operativo, al resolver esta excepción, hace una copia de la página para el proceso que generó la excepción en un nuevo marco de págna.

* Si finalmente la página va a ser escrita, con esta técnica se posterga lo más posible la asignación de nuevos marcos de página para que cada proceso tenga una página privada.

* Si finalmente nadie escribe en ella, hemos ahorrado un marco de página.

* Conclusión: "Copy-on-write" es una técnica que **evita el consumo excesivo de recursos que supondría que fork duplicara físicamente todos los recursos del padre al hijo**,

particularmente en el caso, bastante frecuente, de que el proceso hijo haga pronto un exec.

Terminación de procesos [Lov10 pag36]

* Cuando un proceso termina, el kernel libera todos sus recursos y notifica al padre su terminación.

* Normalmente un proceso termina cuando .....

1) realiza la **llamada al sistema exit();**

   esto puede ser **explícito** si el programador incluyó esa llamada en el código del programa,

   o **implícito**, pues el compilador incluye automáticamente una llamada a exit() cuando main() llega termina.

2) **recibe una señal** ante la que tiene la acción establecida de terminar

* Independientemente de qué acontecimiento ha provocado el fin del proceso, el grueso del trabajo lo hace la función **do_exit()** definida en <linux/kernel/exit.c>

Actuación de do_exit()

1. Establece el flag PF_EXITING de task_struct

2. Para cada recurso que esté utilizando el proceso, por ejemplo espacio de direcciones o archivos, se decremente el contador correspondiente que indica el nº de procesos que lo están utilizando;

si este contador vale 0 se realiza la operación de destrucción oportuna sobre el recurso, por ejemplo si fuera una zona de memoria, se liberaría.

Así por ejemplo, se libera el campo mm_struct de este proceso; si no hay otros procesos que estén usando este espacio de direcciones el kernel lo destruya.

3. El valor que se pasa como argumento a exit() se almacena en el campo exit_code de task_struct; Hay que almacenar lo por si el padre quisiera obtenerlo para tener información sobre cómo ha terminado el hijo.

4. Se manda una **señal al padre** indicando la finalización de su hijo.

5. **Si aún tiene hijos**, se pone como padre de éstos al proceso init

   (dependiendo de las características del grupo de procesos al que pertenezca el proceso, podría ponerse como padre a otro miembro de ese grupo de procesos).

6. Se establece el campo exit_state de task_struct to EXIT_ZOMBIE

7. Se llama a schedule() para que el planificador elija un nuevo proceso a ejecutar

Puesto que este es el último código que ejecuta un proceso, do_exit nunca retorna.