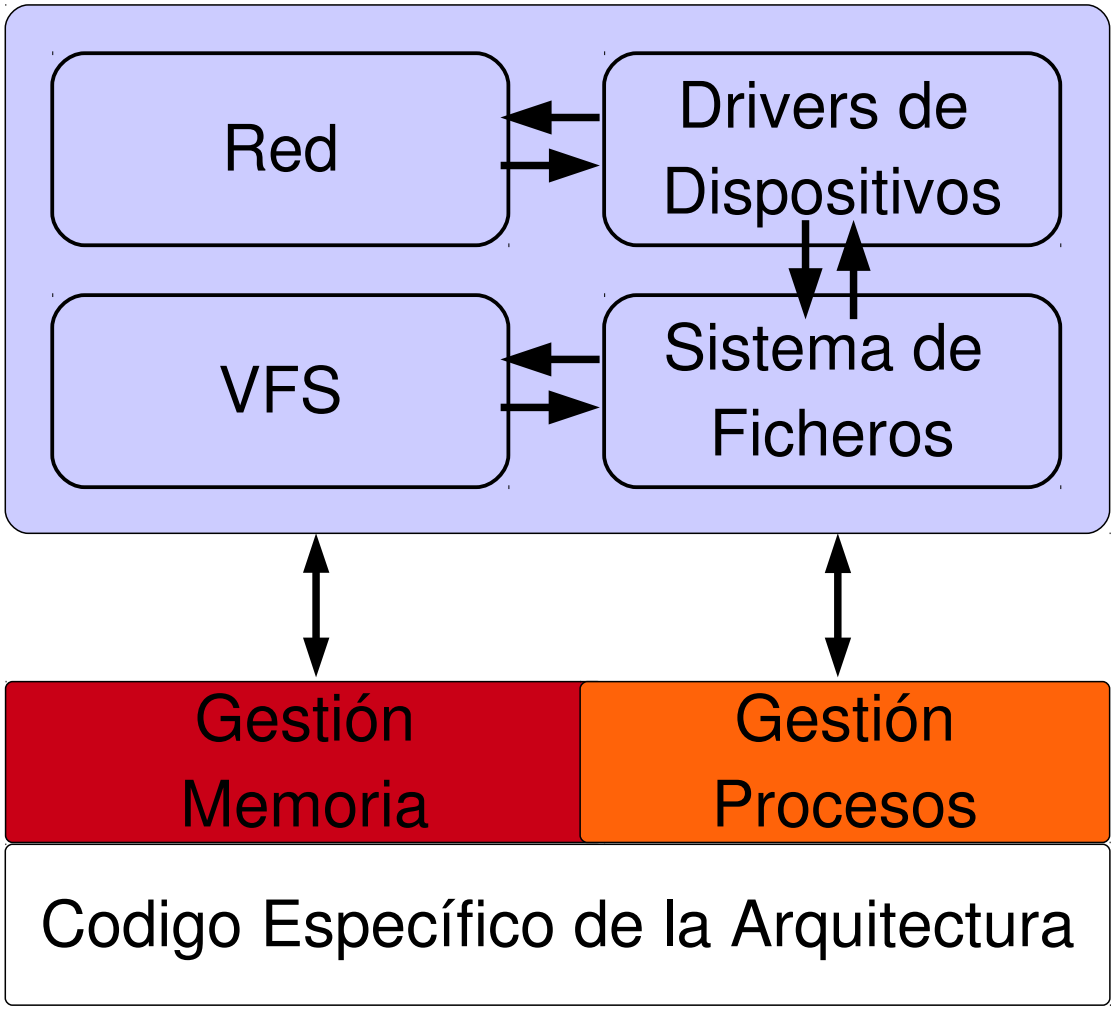
Gestión de Procesos

AISO

Manuel Prieto-Matías
High Performance Computing Division
ArTeCS Group
Universidad Complutense de Madrid

Elementos/Componentes del kernel



El Proceso (I)

- Una de las abstracciones más importantes de Unix/Linux
  - Conceptualmente: programa (código objeto almacenado en algún medio) en ejecución / programa activo.
  - Proporciona dos abstracciones fundamentales:
    - Procesador Virtual (gracias al Scheduling)
      - Proporciona la ilusión al proceso de que monopoliza el sistema
    - Memoria Virtual
      - El proceso puede utilizar la memoria como si fuera el "propietario" de toda la memoria del sistema

El Proceso (II)

- Programa en ejecución: no solo texto, incluye otros “recursos”:
  - Ficheros abiertos
  - Señales pendientes
  - Datos internos del kernel
  - Estado procesador
  - Espacio de direcciones
  - 1 o más Hilos/Threads de ejecución
    - Cada hilo: PC único, pila, registros del procesador
  - Sección de Datos

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 y algunos recursos y estadísticas).
    - Todos los procesos tienen a init (pid=1) como ancestro común.
    - 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.
  - **`wait4()`**
    - Permite que un proceso espere la terminación de un proceso

El Proceso (IV)

- Cuando se crea un proceso
  - Es “casi” idéntico a su padre
  - Recibe un 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 paginas (texto), tienen copias separadas de stack, bss, …
  - Los cambios en el padre en stack, bss,... son invisibles al hijo y viceversa
- **Concepto de Proceso Ligero / Lighweight Process**
  - Procesos que comparten ciertos recursos (espacio direcciones, ficheros abiertos, …)
    - Los “cambios” son visibles
  - En Linux no existe explícitamente la entidad proceso ligero pero es posible establecer diferentes grados de “compartición” (flag **`CLONE_THREAD`** de **`clone()`**)

Task Structure (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 un descriptor de proceso
  - **`struct task_struct <linux/sched.h>`**

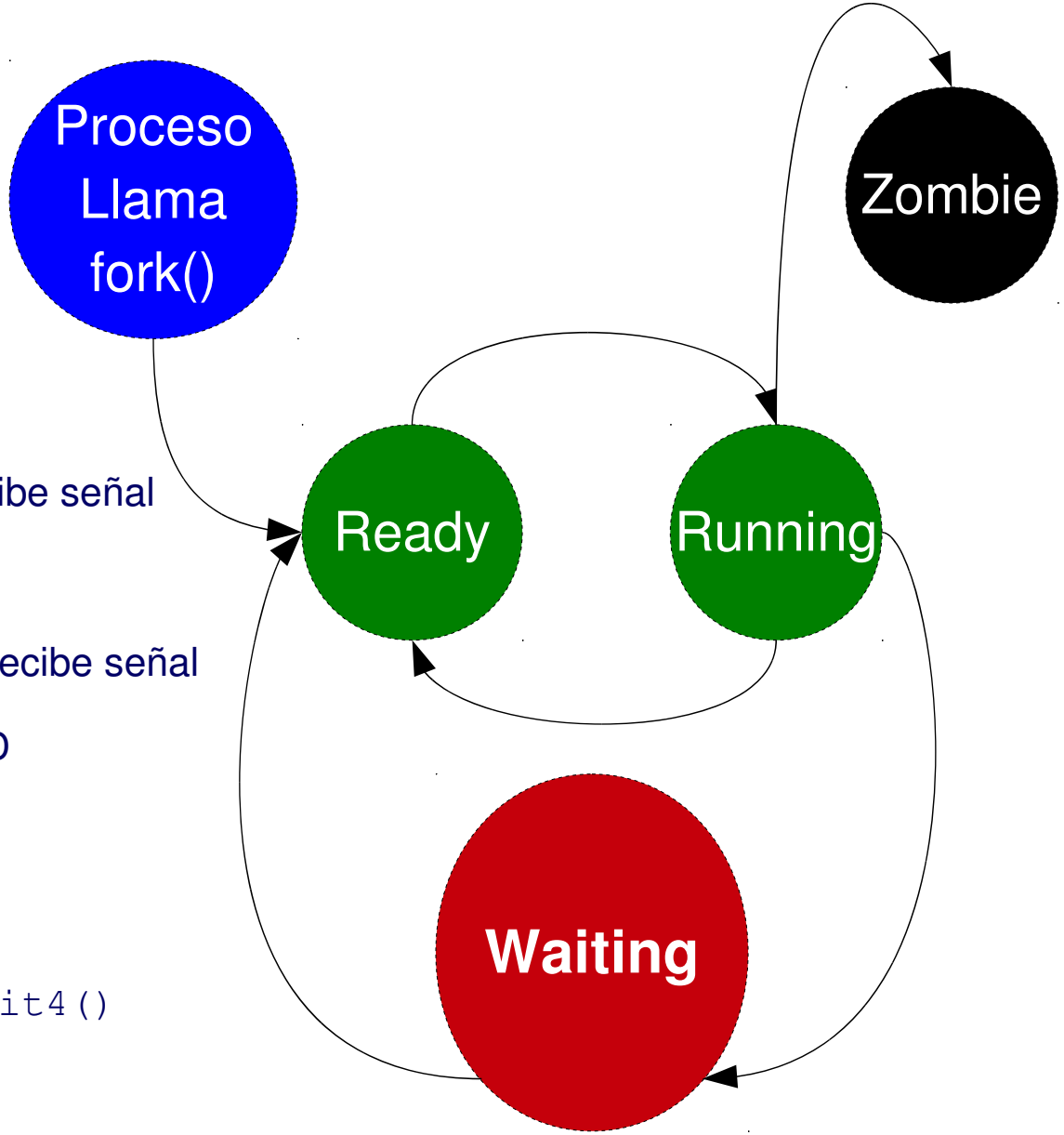
```
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    ...
    unsigned int flags;       /* per process flags, defined below */
    ...
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;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

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, …
  - Señales pendientes
  - ...
- Credenciales
  - Usuario / usuario efectivo ...
- Información sobre la Memoria Virtual Asignada
- Información sobre los ficheros que maneja el proceso
- Información sobre IPC
- Información sobre Señales
  - Manejadores

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
    - Se ha detenido la ejecución
  - EXIT_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)

__TASK_TRACED

- La ejecución del proceso ha sido detenida por un depurador:
  - Cuando un proceso es monitorizado por otro (Llamada `ptrace()` permite monitorizar un proceso), cualquier señal le pone en estado __TASK_TRACED

Cambio de Estado

- En determinadas circunstancias se podría acceder directamente al campo state. Ej: **`p->state = TASK_RUNNING`**
- El procedimiento aconsejado es vía macros (protección concurrencia)
  - **`set_task_state(ptask,value) / set_current_state(value)`**

```
/*
 * set_current_state() includes a barrier so that the write of current->state
 * is correctly serialised wrt the caller's subsequent test of whether to
 * actually sleep:
 *
 *   set_current_state(TASK_UNINTERRUPTIBLE);
 *   if (do_i_need_to_sleep())
 *           schedule();
 *
 * If the caller does not need such serialisation then use __set_current_state()
 */

#define __set_current_state(state_value)                        \
    do { current->state = (state_value); } while (0)
#define set_current_state(state_value)          \
    set_mb(current->state, (state_value))
```

Identificación Procesos

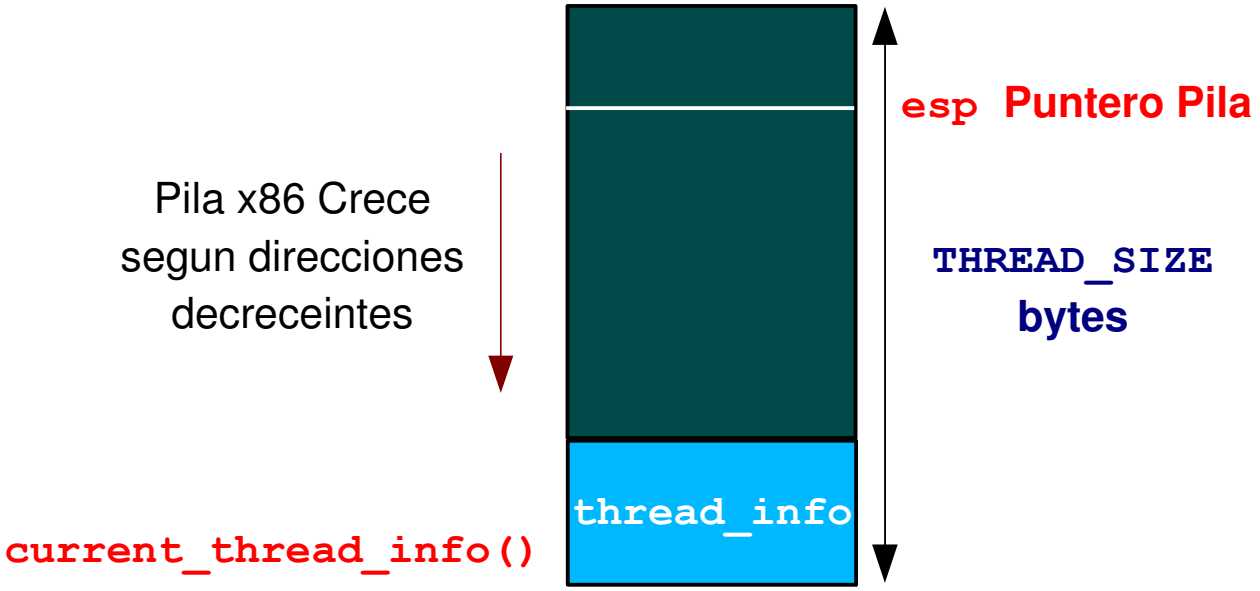
- En el kernel, la mayor parte de las referencias se hace vía puntero a **`task_struct`**
  - Correspondencia 1-1 **`task_struct`**-proceso
  - Macro **`current`**
    - La implementación de esta macro depende de la arquitectura
- Los usuarios suelen identificar los procesos vía PID
  - Se numeran de forma secuencial (habitualmente nuevoPID = PIDprevio +1)
    - Limite superior **`/proc/sys/kernel/pid_max`** (32767 por defecto)
    - Si se supera el limite se reciclan (**`pidmap`** array)

Grupos de Threads

- El estándar Posix define el concepto de grupos de threads
- En linux cada proceso ligero/thread es un proceso.
  - Para identificar el grupo, todos los threads de un mismo grupo comparten el mismo valor en el campo **`tgid`**
    - **`tgid`** = PID del “group leader”
  - Habitualmente grupos de threads constan de un único thread:
    - **`tgid = pid`**
    - **`getpid()`** devuelve el valor de **`tgid`**

pila kernel y `thread_info` (I)

- Cada proceso tiene su propia pila kernel
  - Al "final" de la pila kernel (fondo en x86) se aloja la estructura **`thread_info`** (**`thread_info.h`** en **`arch/x86/include/asm`**)
  - Se puede conocer la dirección de **`thread_info`** a partir del puntero de pila. No es necesario reservar un registro



Manuel Prieto-Matías
Universidad Complutense de Madrid

pila kernel y `thread_info` (II)

- Estructura **`thread_info`**

```
struct thread_info {
    struct task_struct      *task;          /* main task structure */
    struct exec_domain      *exec_domain;   /* execution domain */
    __u32                   flags;          /* low level flags */
    __u32                   status;         /* thread synchronous flags */
    __u32                   cpu;            /* current CPU */
    int                     preempt_count;  /* 0 => preemptable,
                                               <0 => BUG */
    mm_segment_t            addr_limit;
    struct restart_block    restart_block;
    void __user             *sysenter_return;
#ifdef CONFIG_X86_32
    unsigned long           previous_esp;   /* ESP of the previous stack in
                                               case of nested (IRQ) stacks
                                            */
    __u8                    supervisor_stack[0];
#endif
    int                     uaccess_err;
};
```

pila kernel y thread_info (III)

- union thread_union (sched.h)

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

- arch/x86/include/asm/page_32_types.h

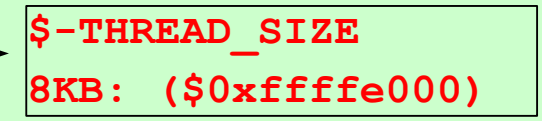
```
#ifdef CONFIG_4KSTACKS
#define THREAD_ORDER   0
#else
#define THREAD_ORDER   1
#endif
#define THREAD_SIZE    (PAGE_SIZE << THREAD_ORDER)
```

pila kernel y thread_info (IV)

- current via thread_info:
  - current =current_thread_info->task()

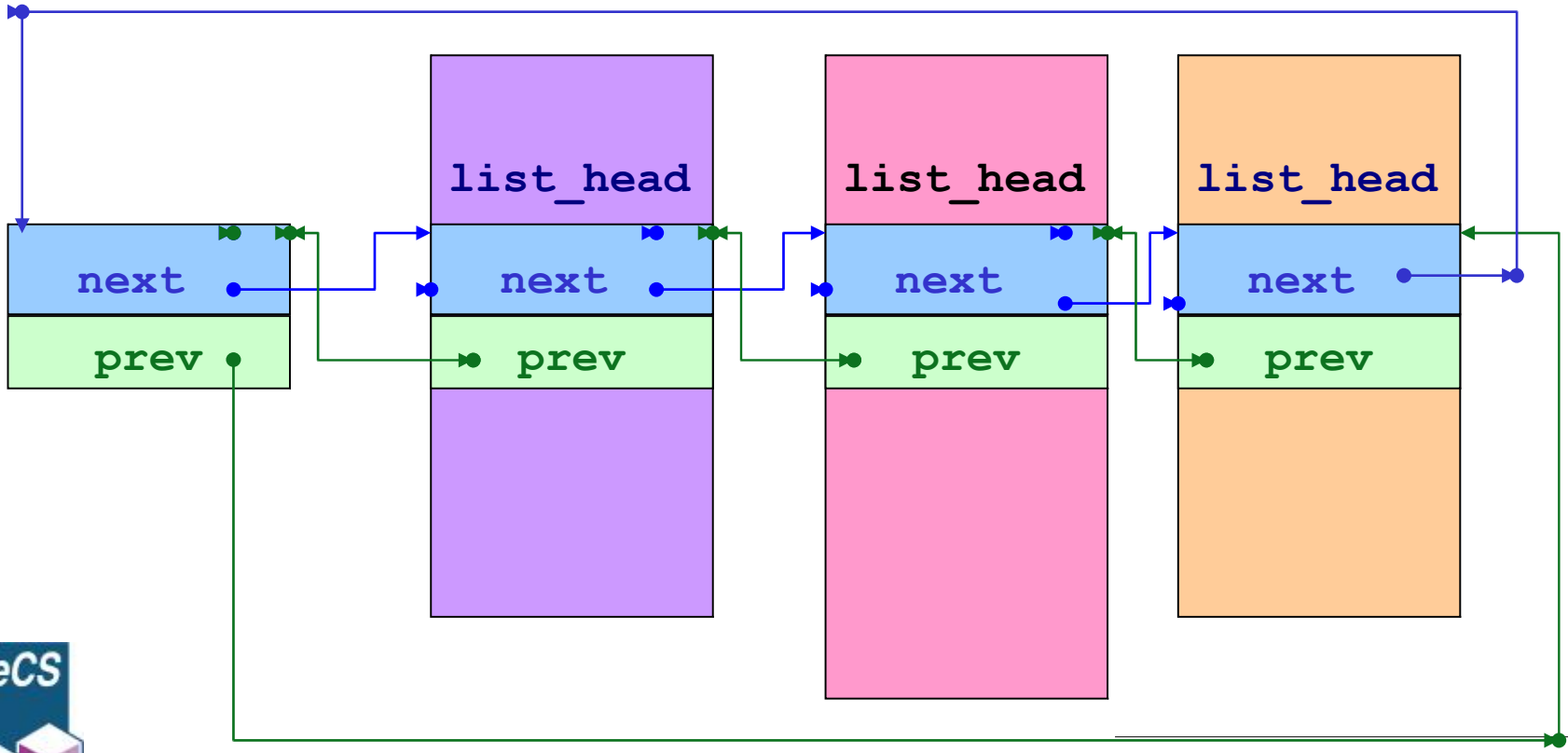
```
/* how to get the current stack pointer from C */
register unsigned long current_stack_pointer asm("esp") __used;

/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
    return (struct thread_info *)(current_stack_pointer & ~(THREAD_SIZE - 1));
}
```

```
/* how to get the thread information struct from ASM */
#define GET_THREAD_INFO(reg)    \
    movl $-THREAD_SIZE, reg; \
    andl %esp, reg

/* use this one if reg already contains %esp */
#define GET_THREAD_INFO_WITH_ESP(reg) \
    andl $-THREAD_SIZE, reg

#endif
```



Implementación `task_list` (I)

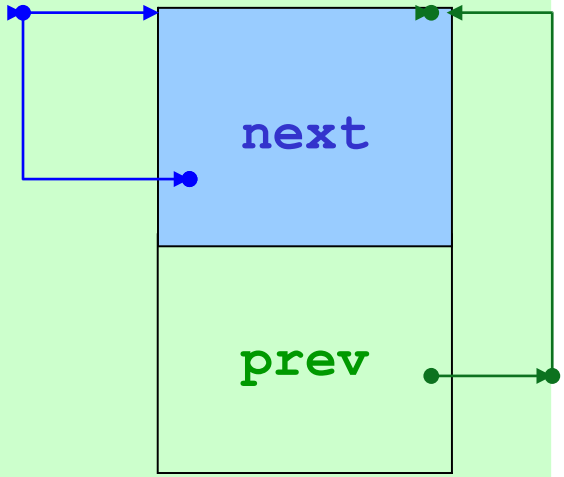
- `task_struct` incluye `struct list_head tasks;`
  - `list_head`: implememementación genérica lista doblemente enlazada
    - Solo dos campos: `next` y `prev`



Manuel Prieto-Matías
Universidad Complutense de Madrid

Macros list_head (I)

```
struct list_head{
    struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }
#define LIST_HEAD(name) \
    struct  list_head name = LIST_HEAD_INIT(name)

list_add(node,pointer)
list_add_tail(node,pointer)
list_del(pointer)
list_empty(pointer)
```



Macros `list_head` (II)

`list_entry(pointer,type,name)`

Devuelve la dirección de la estructura de datos de tipo **`type`** en la que se incluye un campo **`list_head`** con el nombre **`name`** y cuya dirección es **`pointer`**

```
next_task(task)
        list_entry(task->tasks.next, struct task_struct, tasks)

prev_task(task)
        list_entry(task->tasks.prev, struct task_struct, tasks)
```

`list_for_each(pos,head)`

```
#define list_for_each(pos, head) \
for(pos = (head)->next; pos != (head); pos = pos->next)
```

`list_for_each_entry(pos,head,member)`

Implementación `task_list` (II)

- Primer elemento: descriptor de tarea **`init`**
  - Definido estáticamente. Ej **`X86: arch/x86/kernel/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,                                 \
    .lock_depth     = -1,                                         \
    .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;
counter=1; /* for init_task */
for_each_process(task){
  if(task->state==TASK_RUNNING)
     ++counter;
}
```

Manuel Prieto-Matías
Universidad Complutense de Madrid

Parentescos (I)

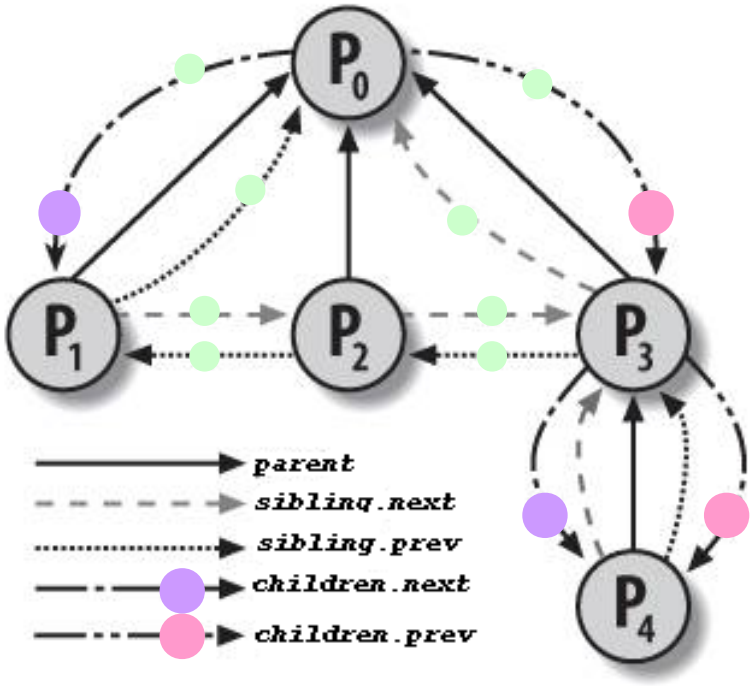
- `task_struct` incluye también

```
/*
 * 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 */
}
```



Manuel Prieto-Matías
Universidad Complutense de Madrid

Creación Procesos (I)

- Unix: Creación de procesos desacoplada combinando **`fork()`** y **`exe()`**
  - **`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 una nueva **`task_struct`**
      - Copia de las tablas de página

Creación Procesos (II)

- 3BSD: **`vfork()`**
  - Mismo efecto que **`fork()`** + COW, aunque adicionalmente:
    - No se copian las tablas de paginas.
    - El padre se bloquea hasta que el hijo **`exit()`** or **`exec()`**
  - Uso desaconsejado actualmente
- Linux: **`clone()`**
  - Permite especificar de forma precisa que recursos se comparten

Creación Procesos (III)

- `fork()`, `vfork()` y `clone()` se implementan vía `do_fork()`
  - `linux/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 (IV)

- Flags (Propiedades de Duplicación)

  - 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 (V)

- Flags (Propiedades de Duplicación)

  - 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
    - `...`
  - Implementación de **`fork()`**: `SIGCHLD`
  - Implementación de **`vfork()`**: `CLONE_VFORK | CLONE_VM |SIGCHLD`

Creación Procesos (VI)

- `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 puede examinar proceso
  - `wake_up_new_task`
    - Se añade la nueva tarea a las colas de ejecución del *scheduler*
  - Si **`CLONE_VFORK`** `wait_for_completion`

Creación Procesos (VII)

- `struct task_struct *copy_process(...)`
  - Comprobar flags (algunas combinaciones no tienen sentido)
  - `dup_task_struct`
    - Nuevo `kernel_stack,thread_info y task_struct`
  - Comprobar limites 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 (VIII)

- `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 (IX)

- `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);
```

AISO Introducción

Versión 0.1

© **Manuel Prieto Matias**

Este documento (o uno muy similar) esta disponible en
https://cv2.sim.ucm.es/moodle/course/view.php?id=3235