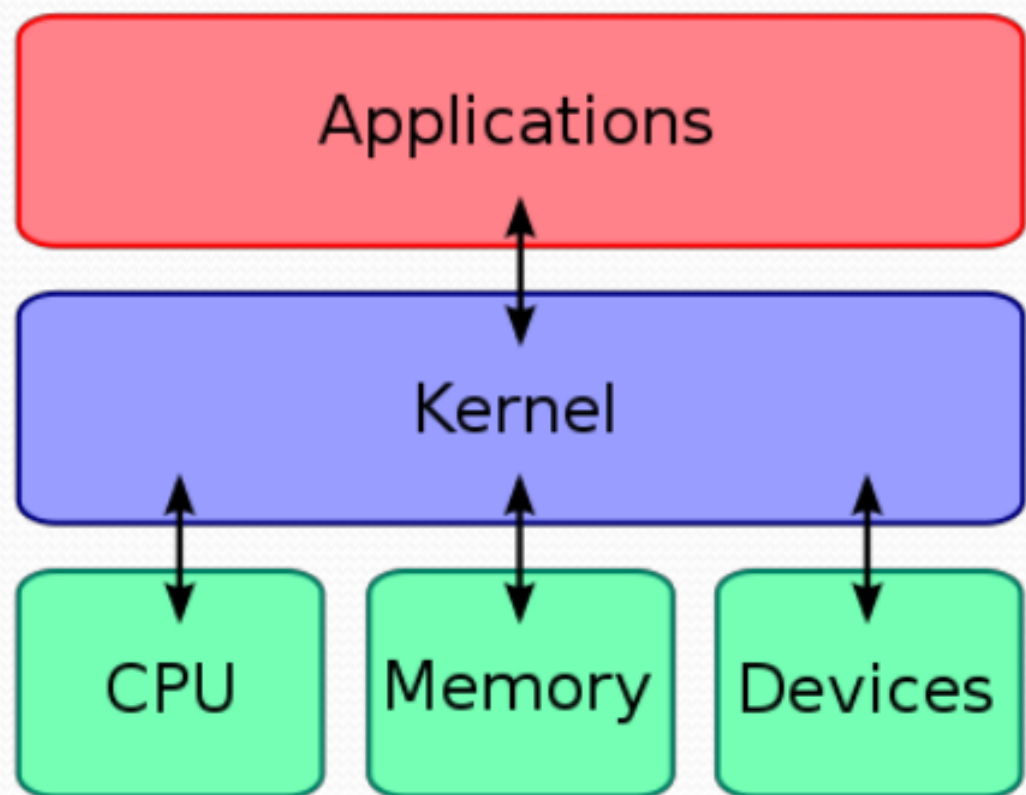




Clase 2

Kernel – Procesos



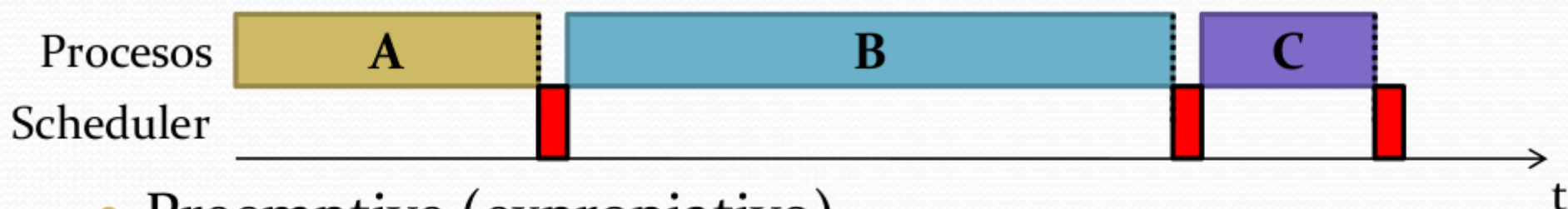
Kernel

- Núcleo del sistema operativo
- Administra el tiempo de CPU para cada aplicación
- Administra la memoria
- Se comunica con el hardware

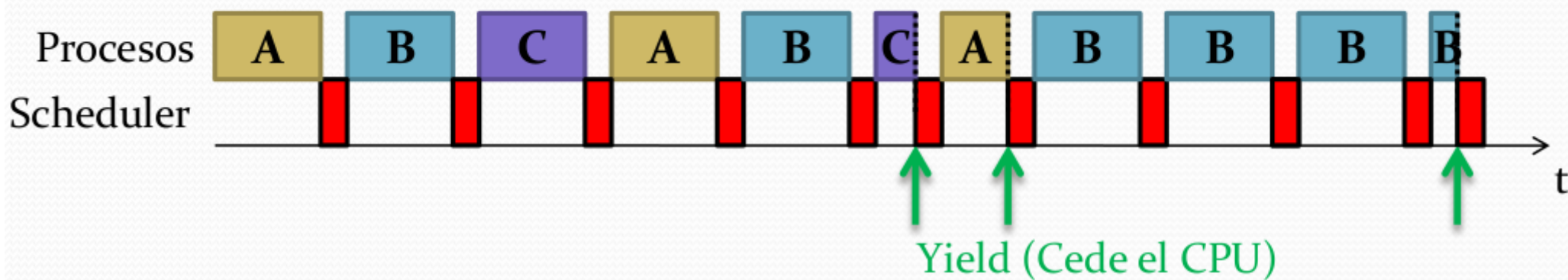


Scheduler

- Non-Preemptive (cooperativo)



- Preemptive (expropiativo)





Kernel space Vs User space

- **User Space:** Porción de memoria donde se ejecuta un proceso del usuario en el marco del OS.
- El rol del kernel es que un proceso no pueda interferir en un espacio de memoria de otro proceso
- **Kernel Space:** Porción de memoria donde se ejecuta el código del kernel.
- Los procesos de usuario tienen acceso a porciones limitadas de memoria
- El kernel tiene acceso a toda la memoria.



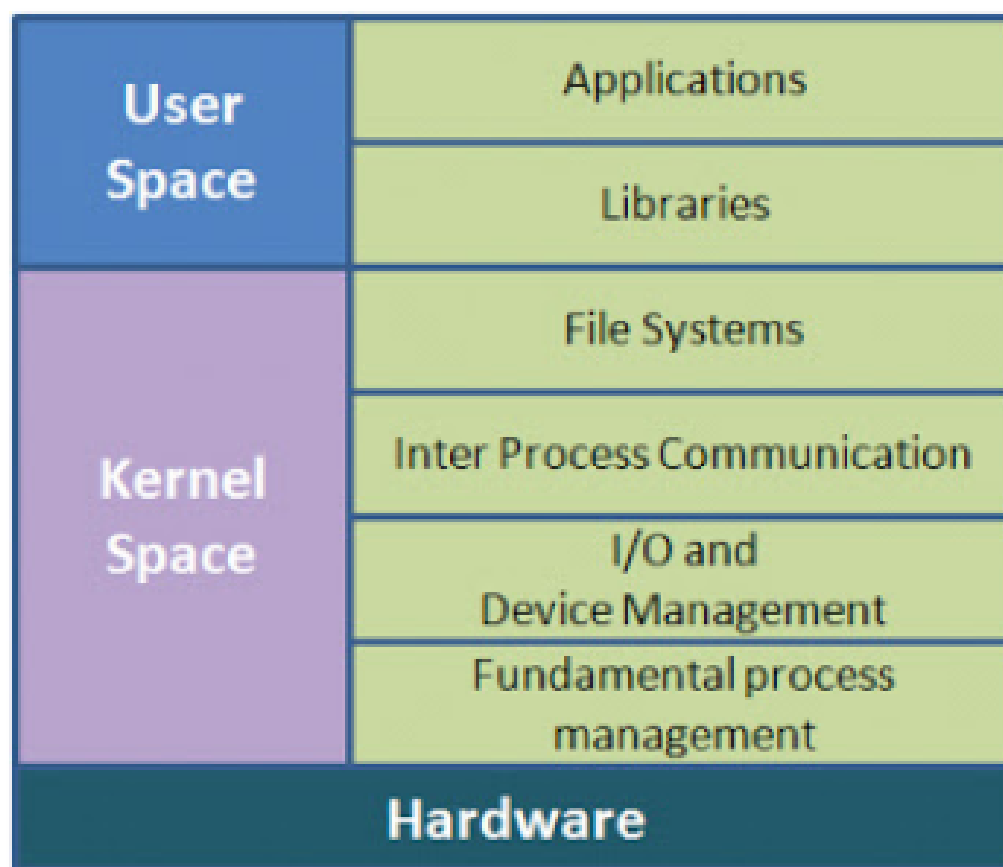
Tipos de Kernel

- **Kernel Monolítico** (Linux <2.6, MS-DOS, Windows 9x)
- **Micro kernel** (Minix)
- **Híbrido** (Windows XP, Windows 7, MAC OSX)
- **Kernel monolítico modular** (Linux>2.6)



Monolítico

Ejecuta todos los servicios del OS en kernel space.

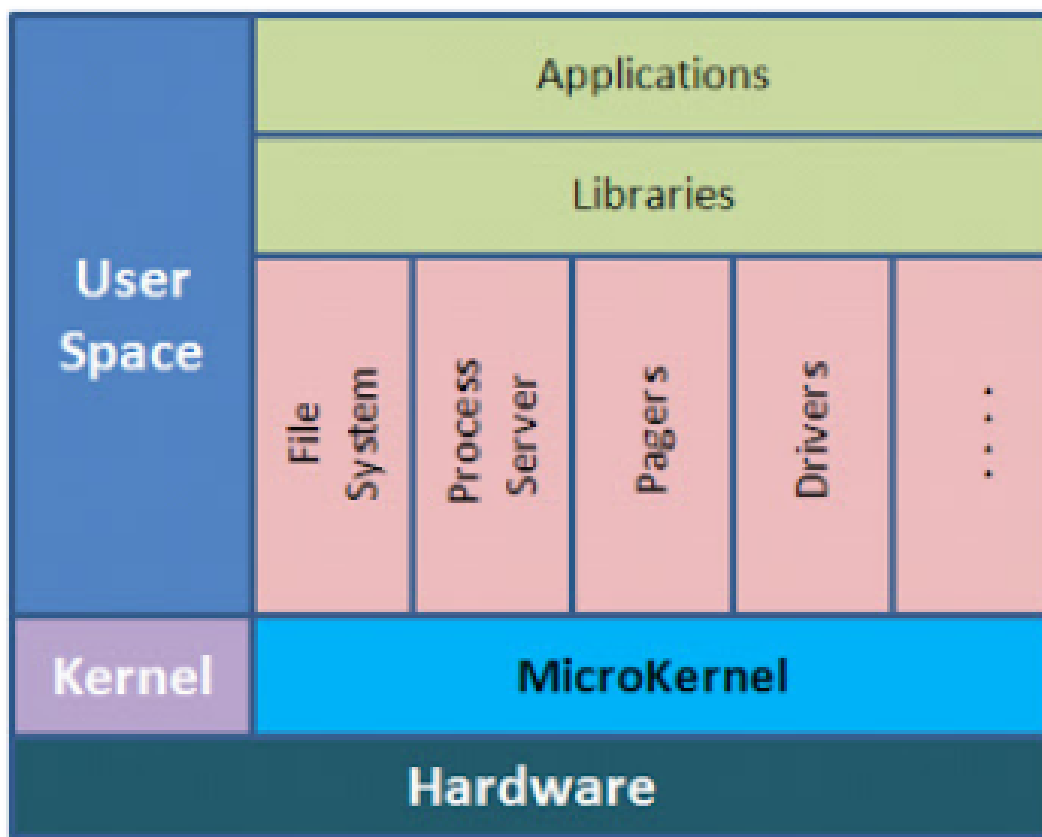


Desventajas: Kernel de gran tamaño, hay que recompilar para agregar drivers.



Microkernel

Solo el manejo de comunicación entre procesos e IO queda en kernel space, el resto de los servicios se ejecuta en user space.





Híbrido

- **Similar al microkernel**
- **Módulos más esenciales se ejecutan en modo kernel**
- **Busca aumentar la performance del sistema.**
- **Esto termina en un kernel más grande.**



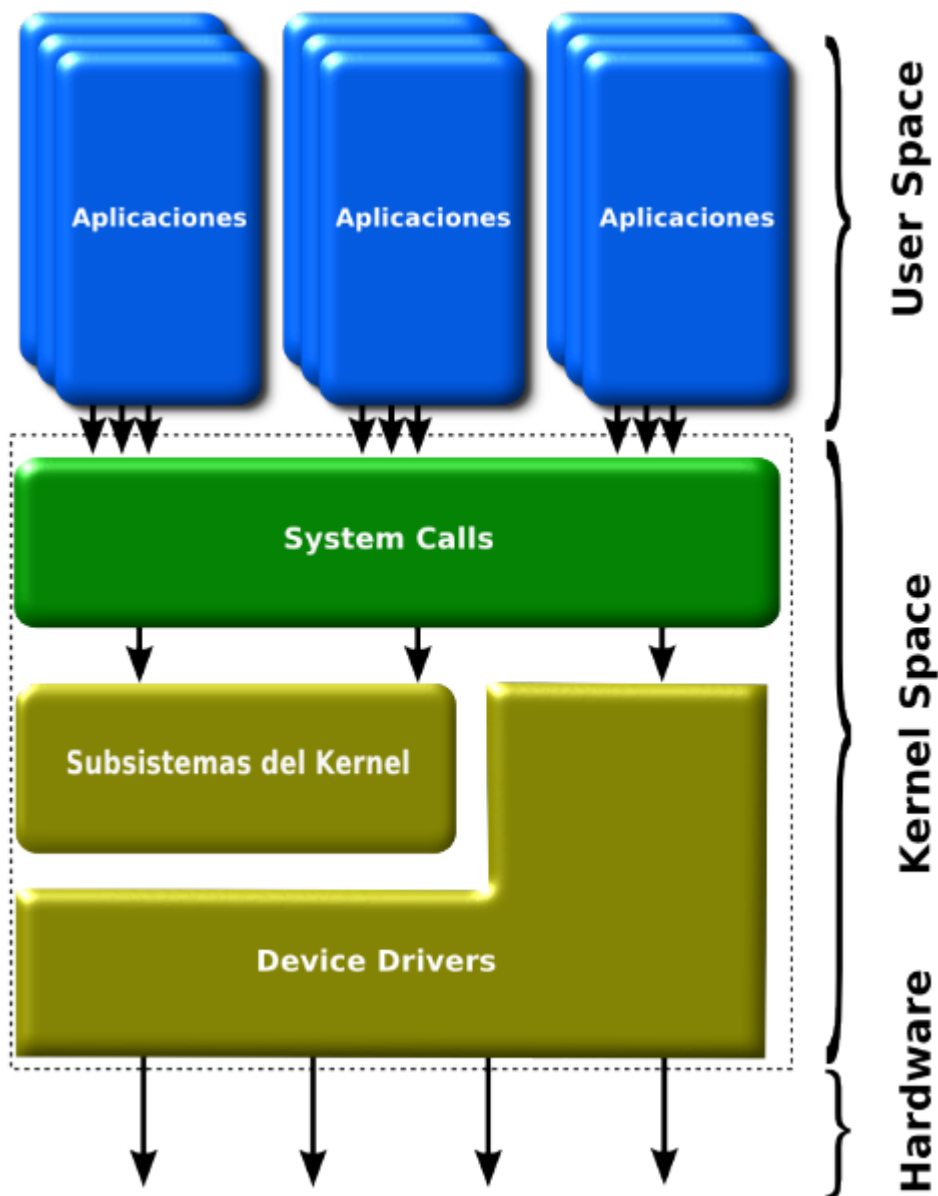
Modular

- Similar al microkernel
- Módulos más esenciales se ejecutan en modo kernel
- Los modulos se cargan al iniciar
- No se necesita recompilar para agregar/quitar drivers



Resumen

- **Monolítico:** kernel image = kernel core + kernel services
- **Microkernel:** kernel image = kernel core
- **Híbrido:** kernel image = kernel core + algunos kernel services
- **Modular:** kernel image = kernel core + IPC service modules + Memory module + Process Management module + módulos (up/down on the fly)



Kernel de Linux

- Las aplicaciones se comunican con el kernel por medio de *systemcalls*.
- El kernel resuelve los servicios accediendo al hardware o ejecutando algoritmos y devuelve un resultado



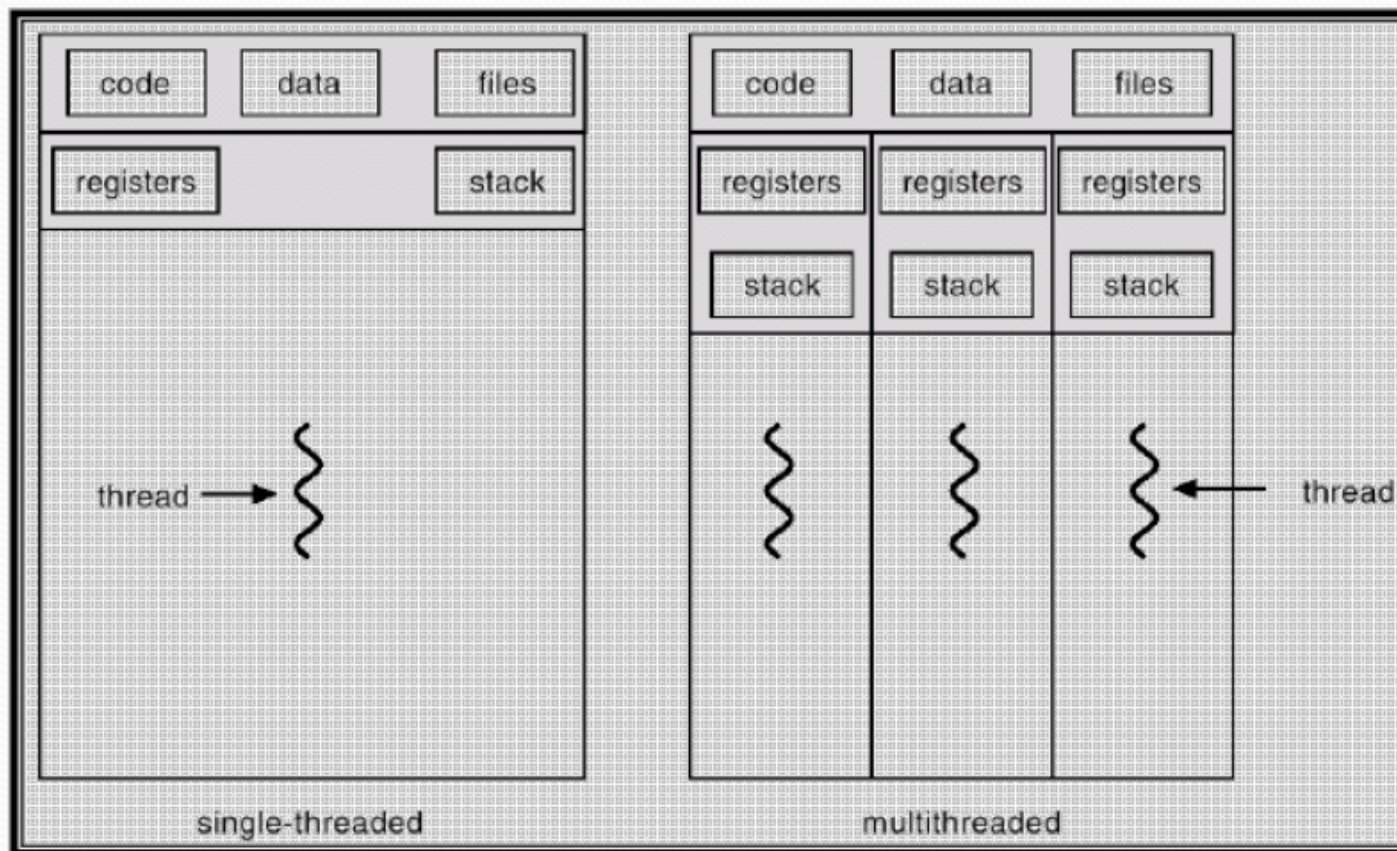
Proceso

- Instancia de ejecución de un programa
- Se compone del código en ejecución junto con los datos del proceso:
 - Archivos abiertos
 - Señales pendientes
 - Estado
 - Contexto
 - Espacio de direcciones de memoria
 - Variables globales



Thread

- Comparte el mismo espacio de memoria que otros threads
- Tiene su propio conjunto de stack y registros





Tipos de threads

- **Kernel threads**
- **Lightweight processes**
- **User threads**



Kernel threads

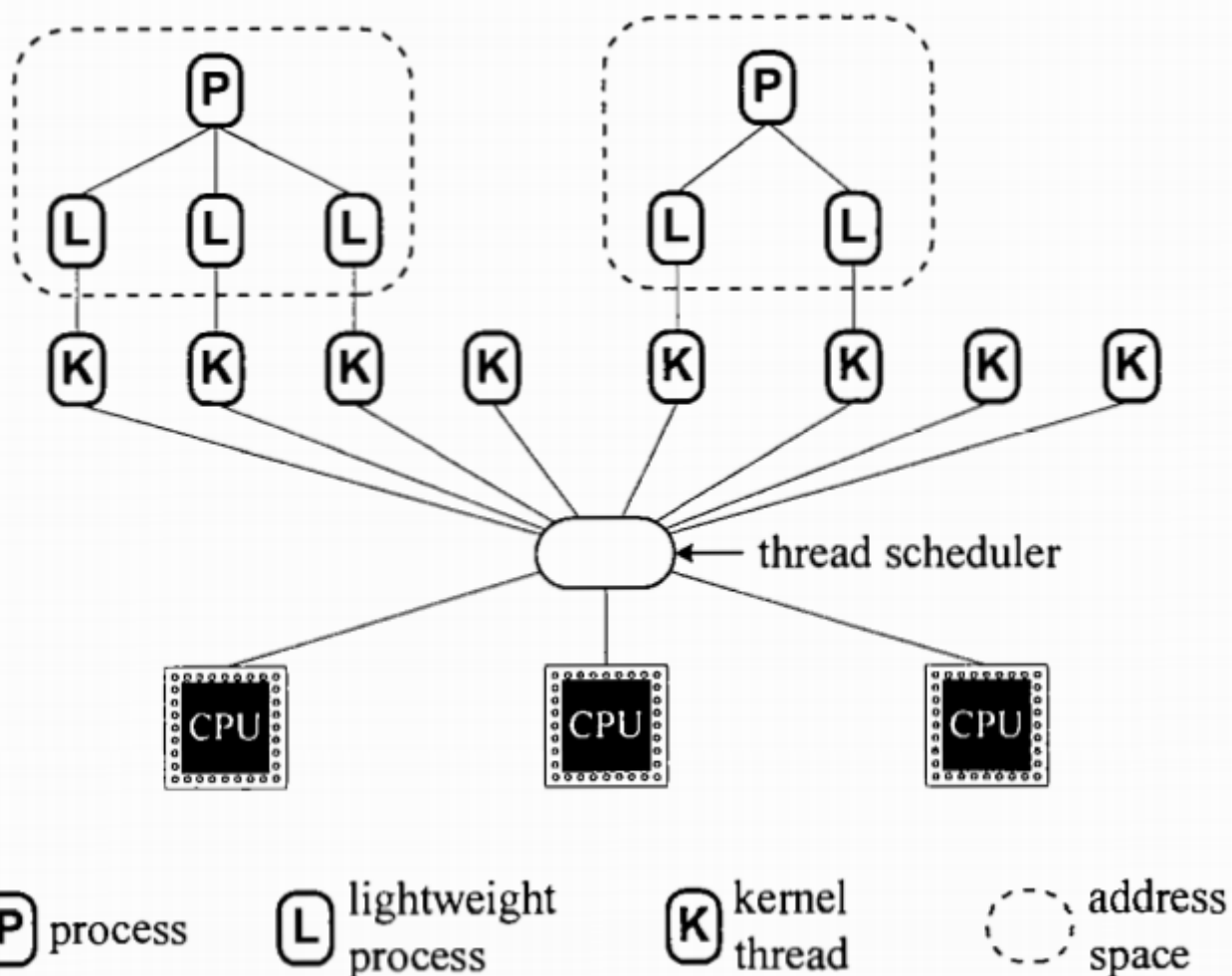
- **No está asociado con un user process**
- **Se crea y se destruye por el kernel**
- **El kernel los lanza para ejecutar ciertas tareas internas**
- **Solo acceden y se ejecutan en kernel space**



Lightweight processes

Es un thread en user space manejado por un kernel thread

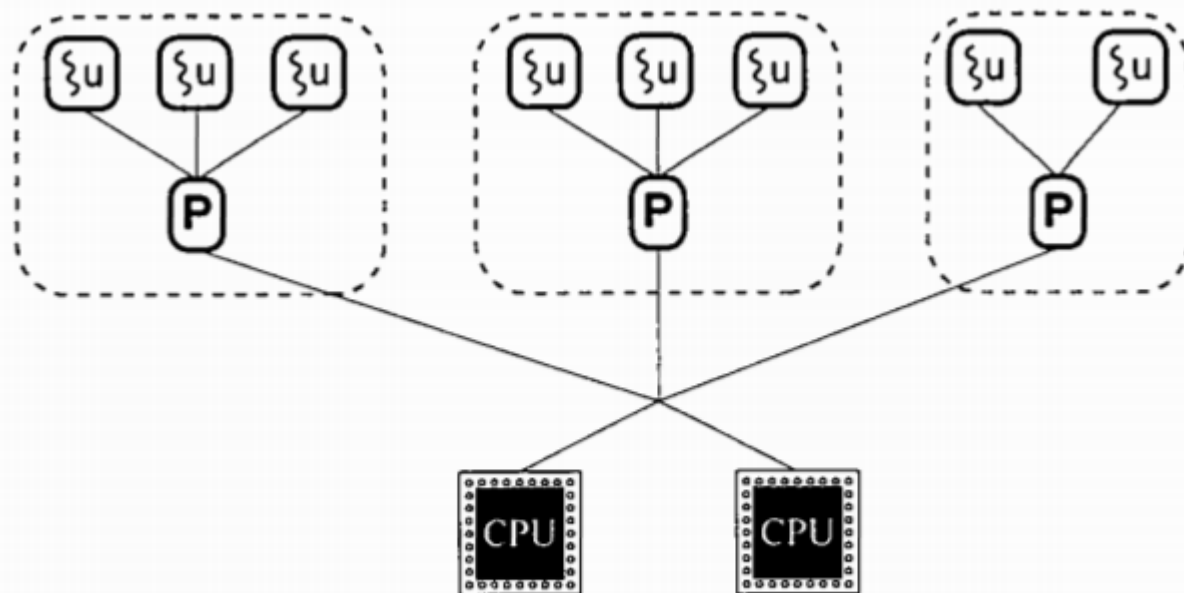
- Syscalls para crear y eliminar threads
- Consume recursos del kernel
- Paralelismo verdadero
- En pthreads de posix se usa syscall clone()





User threads

- Abstracción solo a nivel del usuario, el kernel no sabe que están estos threads.
- “Green threads” en lenguajes con máquina virtual
- No requiere comunicación con el kernel, no existen syscalls
- No se aprovecha el multicore

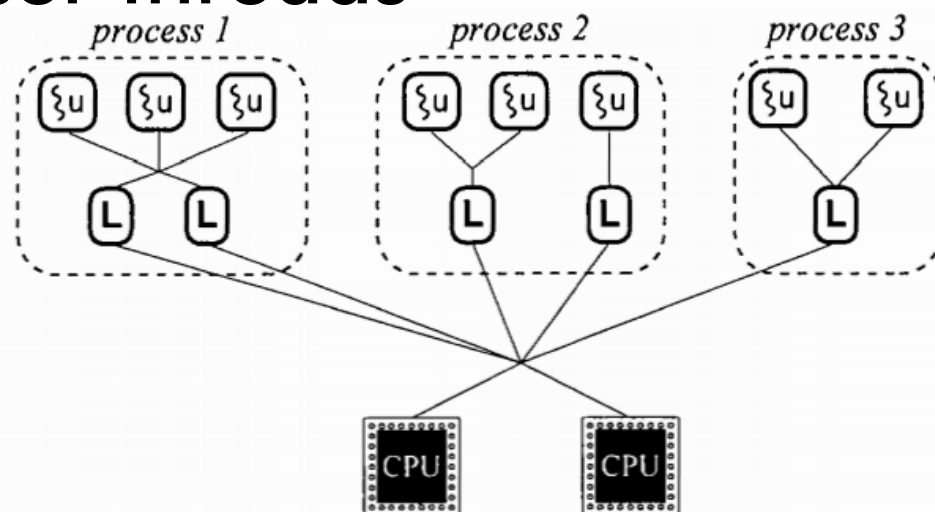


(a) User threads on top of ordinary processes



User threads

- Es posible combinar user threads con lightweight processes
- El scheduler del kernel solo conoce a los LWPs, y asigna tiempo de cpu a cada uno de ellos
- La biblioteca de user threads se encarga de realizar el scheduling entre los user threads



(b) User threads multiplexed on lightweight libraries



Creación de procesos

- Cada proceso tiene un ID único
- Se debe ejecutar la función **fork()**
 - En proceso padre: Devuelve el PID del proceso hijo o -1
 - En proceso hijo: Devuelve 0

```
pid_t pid;
```

```
pid=fork();
```

```
if (pid==0) {  
    printf("I'm the child!\n");  
} else {  
    printf("I'm the parent!\n");  
}
```



Padre

```
pid_t pid;  
  
pid=fork();  
  
if (pid==0) {  
    ...  
} else {  
    Viene aca  
    (pid vale  
    el id del  
    hijo)  
}
```

Hijo



Padre

```
pid_t pid;
```

```
pid=fork();
```

```
if (pid==0) {
```

```
    ...
```

```
} else {
```

```
    Viene aca
```

```
    (pid vale
```

```
    el id del
```

```
    hijo)
```

```
}
```

Hijo

```
pid_t pid;
```

```
pid=fork();
```

```
if (pid==0) {
```

```
    Viene aca
```

```
} else {
```

```
    ...
```

```
}
```



Padre

← Inicia en main

```
pid_t pid;
```

```
pid=fork();
```

```
if (pid==0) {
```

```
...
```

```
} else {
```

```
Viene aca  
(pid vale  
el id del  
hijo)
```

```
}
```

Hijo

```
pid_t pid;
```

```
pid=fork();
```

← Inicia en
Esta
línea

```
if (pid==0) {
```

```
Viene aca
```

```
} else {
```

```
...
```

```
}
```



Padre

← Inicia en main

```
pid_t pid;  
int a=7;  
pid=fork();
```

```
if (pid==0) {  
    ...  
} else {  
    Viene aca  
    (pid vale  
    el id del  
    hijo)  
}
```

a=7

Hijo

```
pid_t pid;  
int a=7;  
pid=fork(); ← Inicia en  
Esta línea
```

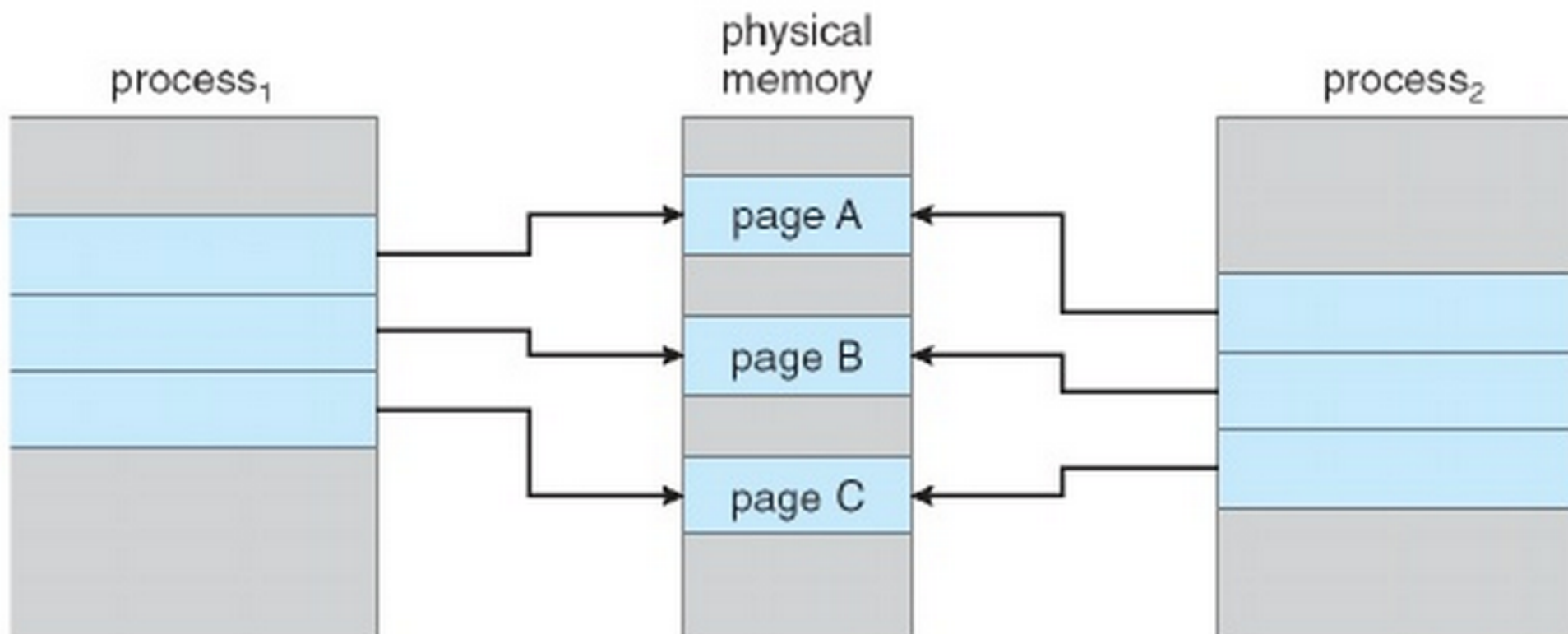
```
if (pid==0) {  
    Viene aca  
} else {  
    ...  
}
```

a=?



Copy on write

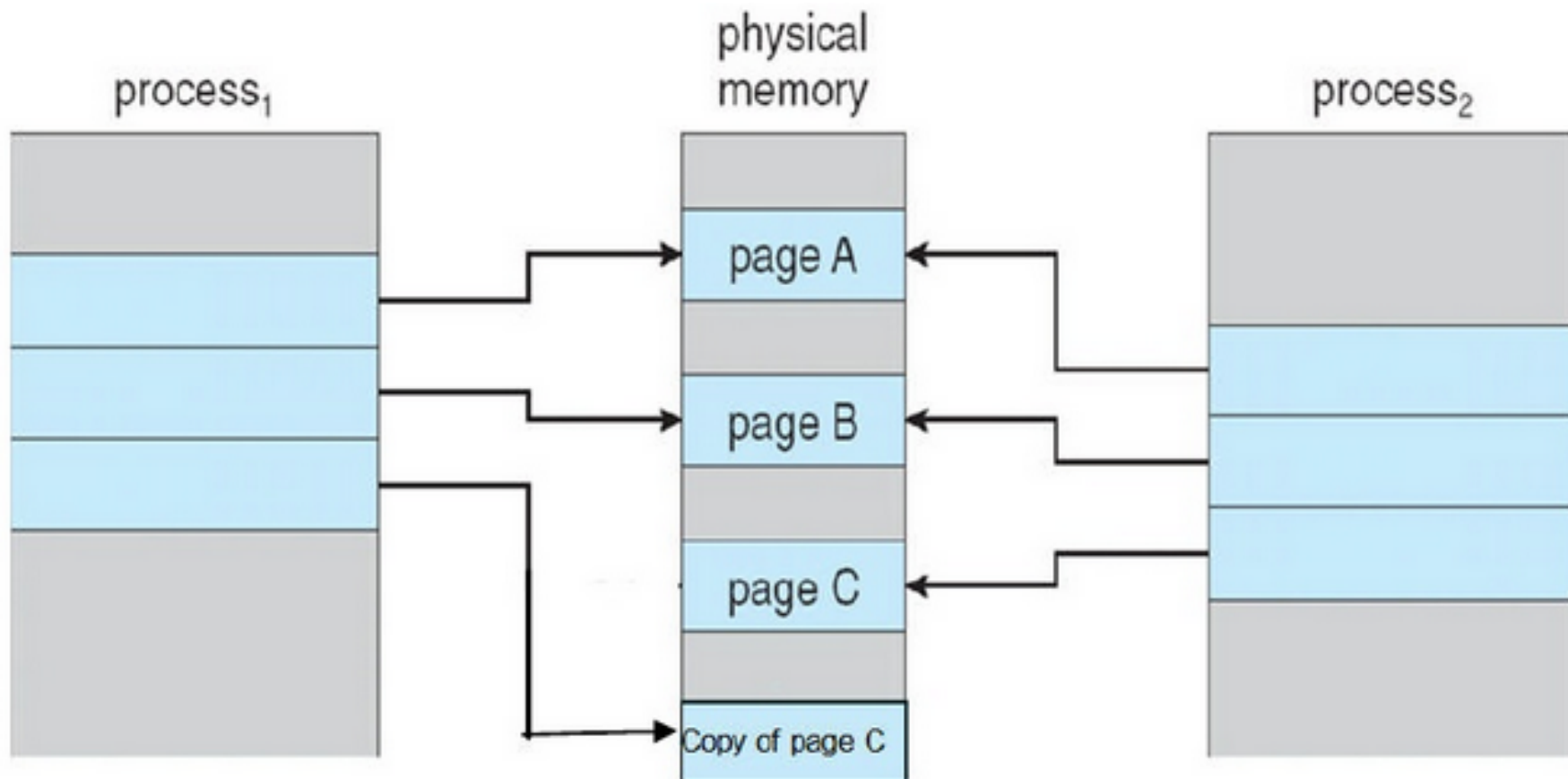
- Permite al proceso padre e hijo compartir las mismas páginas de memoria.
- Cuando algún proceso modifica una variable, se copia la página en otra dirección y se asigna a ese proceso.





Copy on write

- Cuando *process₁* escribe una variable en *page C*, se crea una copia de la misma.





Creación de procesos

Finalización proceso hijo

- Existe un área de datos que no es eliminada.
- El proceso padre puede consultar estos datos (p. ej. Valor retorno)
- Se eliminará cuando el proceso padre ejecute la función **wait()** o **waitpid()**
- Mientras tanto el proceso hijo se encuentra en un estado "Zombie"



Creación de procesos

Finalización proceso padre

- El hijo pasa a tener un nuevo proceso padre: **init**
- El proceso **init** elimina periódicamente los procesos hijos que están en estado "Zombie".



```
int main(void)
{
    pid_t pid;
    int rv;
    switch(pid = fork())
    {
        case -1:
            perror("fork"); /* something went wrong */
            exit(1);

            /* parent exits */
        case 0:
            printf(" CHILD: This is the child process!\n");
            printf(" CHILD: My PID is %d\n", getpid());
            printf(" CHILD: My parent's PID is %d\n", getppid());
            printf(" CHILD: Enter my exit status (make it small): ");
            scanf(" %d", &rv);
            printf(" CHILD: I'm outta here!\n");
            exit(rv);

        default:
            printf("PARENT: This is the parent process!\n");
            printf("PARENT: My PID is %d\n", getpid());
            printf("PARENT: My child's PID is %d\n", pid);
            printf("PARENT: I'm now waiting for my child to exit()...\n");
            wait(&rv);
            printf("PARENT: My child's exit status is: %d\n", WEXITSTATUS(rv));
            printf("PARENT: I'm outta here!\n");
    }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```



Estados de procesos

<u>Estado</u>	Descripción
TASK_RUNNING	Se encuentra en la lista del scheduler (puede estar ready o executing)
TASK_INTERRUPTIBLE	Se encuentra bloqueado (por ejemplo funcion sleep) pero puede ser despertado (interrupted)
TASK_UNINTERRUPTIBLE	Se encuentra bloqueado. Solo el propio proceso lo puede despertar (Por ejemplo un driver)
TASK_ZOMBIE	El padre no ejecuto wait
TASK_STOPPED	Proceso detenido por un debugger



Estados de procesos

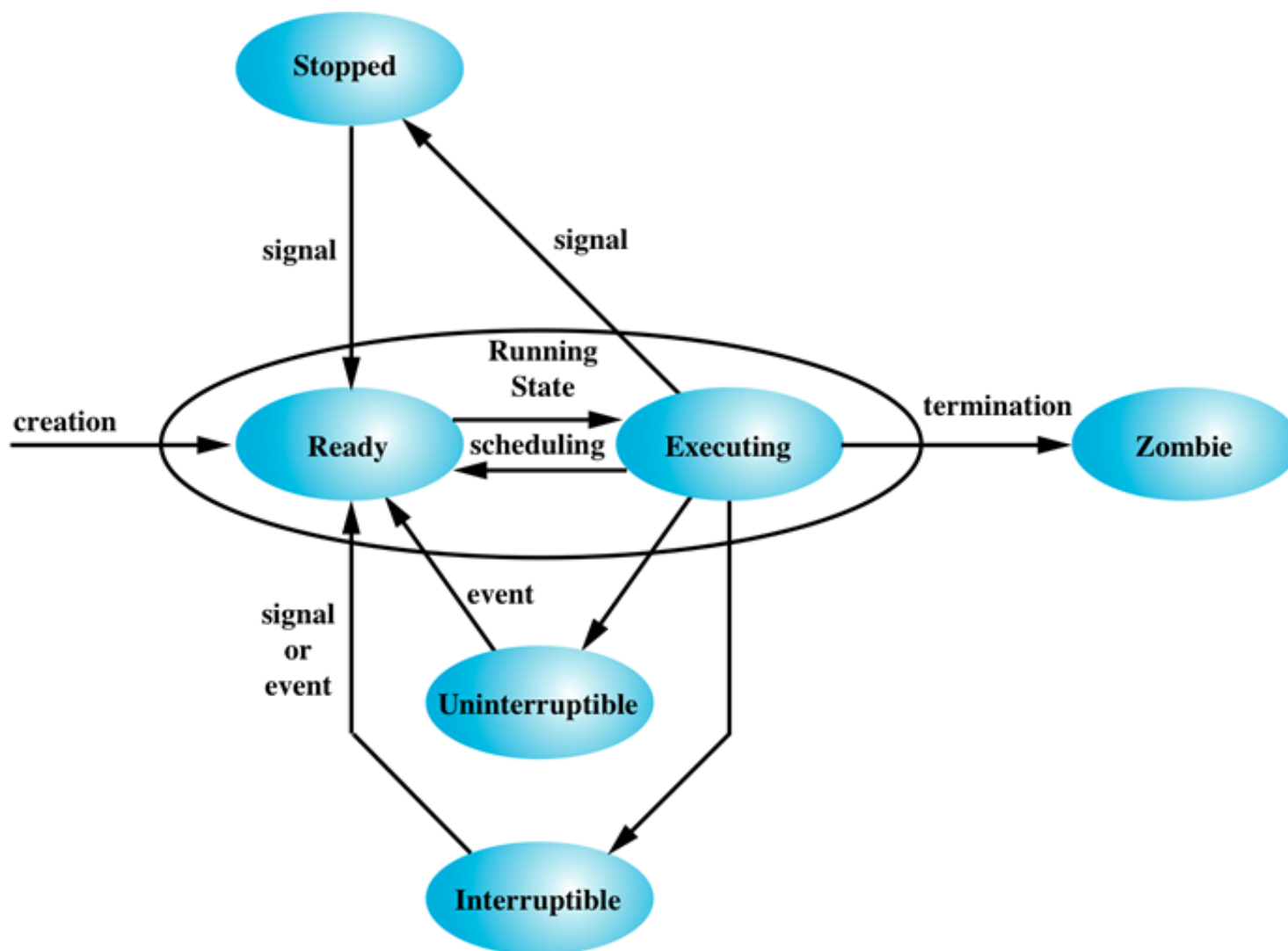
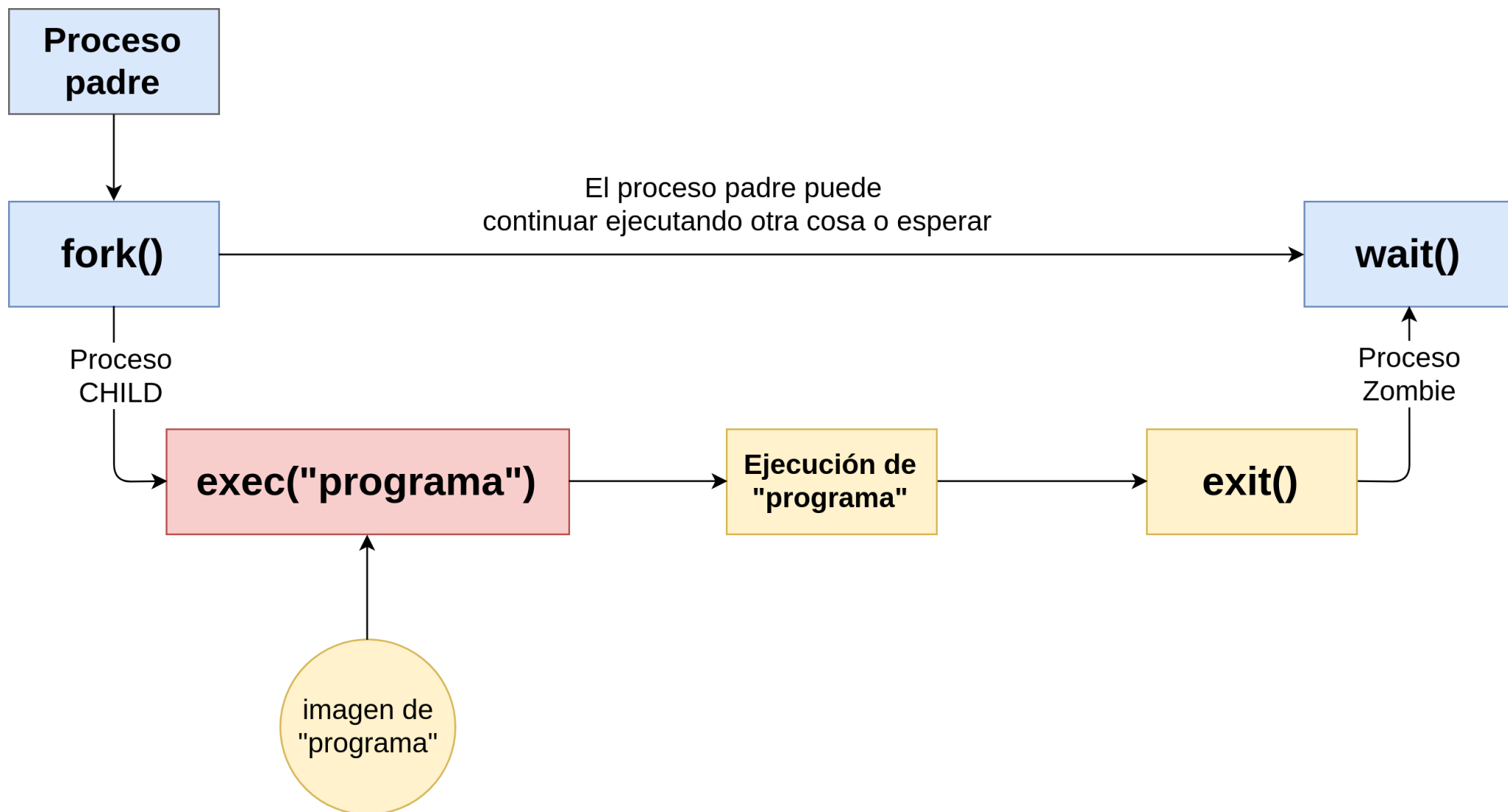


Figure 4.18 Linux Process/Thread Model



¿Cómo se lanza un proceso?





Visualización estados

`ps -elf`

```
ernesto@ernesto-X401A1:~$ ps -elf
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	STIME	TTY	TIME	CMD
4	S	root	1	0	0	80	0	-	6733	poll_s	feb25	?	00:00:00	/sbin/init
1	S	root	2	0	0	80	0	-	0	kthrea	feb25	?	00:00:00	[kthreadd]
1	S	root	3	2	0	80	0	-	0	smpboo	feb25	?	00:00:02	[ksoftirqd/0]
1	S	root	5	2	0	60	-20	-	0	worker	feb25	?	00:00:00	[kworker/0:0H]
1	S	root	7	2	0	60	-20	-	0	worker	feb25	?	00:00:00	[kworker/u:0H]
1	S	root	8	2	0	-40	-	-	0	cpu_st	feb25	?	00:00:00	[migration/0]
1	S	root	9	2	0	80	0	-	0	rcu_gp	feb25	?	00:00:00	[rcu_bh]
1	S	root	10	2	0	80	0	-	0	rcu_gp	feb25	?	00:00:06	[rcu_sched]
5	S	root	11	2	0	-40	-	-	0	smpboo	feb25	?	00:00:00	[watchdog/0]
5	S	root	12	2	0	-40	-	-	0	smpboo	feb25	?	00:00:00	[watchdog/1]
1	S	root	13	2	0	80	0	-	0	smpboo	feb25	?	00:00:02	[ksoftirqd/1]
1	S	root	16	2	0	60	-20	-	0	worker	feb25	?	00:00:00	[kworker/1:0H]
1	S	root	17	2	0	60	-20	-	0	rescue	feb25	?	00:00:00	[cpuset]
1	S	root	18	2	0	60	-20	-	0	rescue	feb25	?	00:00:00	[khelper]
5	S	root	19	2	0	80	0	-	0	devtmp	feb25	?	00:00:00	[kdevtmpfs]
1	S	root	20	2	0	60	-20	-	0	rescue	feb25	?	00:00:00	[netns]
1	S	root	21	2	0	80	0	-	0	bdi_fo	feb25	?	00:00:00	[bdi-default]
1	S	root	22	2	0	60	-20	-	0	rescue	feb25	?	00:00:00	[kintegrityd]
1	S	root	23	2	0	60	-20	-	0	rescue	feb25	?	00:00:00	[kblockd]



Columna F: Flags. 1: Se creo con fork (y no se llamó a exec).

4: Utilizado con permisos de root

Columna S: State. D: Uninterruptible. R: Running. S: Interruptible. T: Stopped

Columna UID: id del usuario

Columna PID: El id del proceso

Columna PPID: El id del proceso padre

Columna C: Uso del cpu

Columna PRI: Prioridad actual del proceso. A mayor valor menor prioridad.

Columna NI: Prioridad asignada al proceso (Valor nice) de -19 a 20. A mayor valor menor prioridad. Se puede modificar con el comando "nice".



Bibliografía

- Uresh Vahalia. (1996). UNIX Internals. The New Frontiers. New Jersey, Prentice Hall.
- Brian “Beej Jorgensen” Hall. (2015). Beej's Guide to Unix IPC.