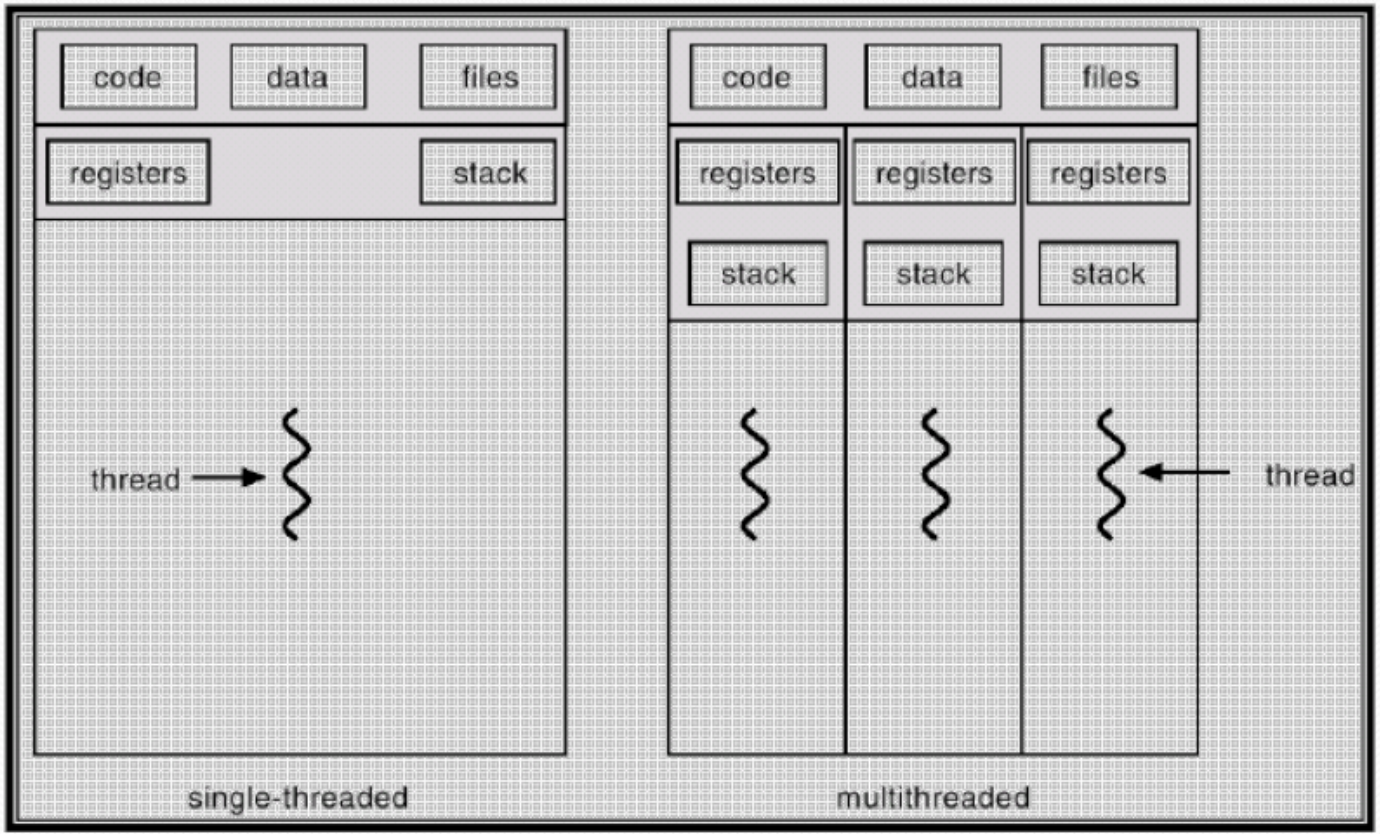
Clase 6
Threads

Autor: Esp. Ing. Ernesto Gigliotti. UTN-FRA

Threads

- 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

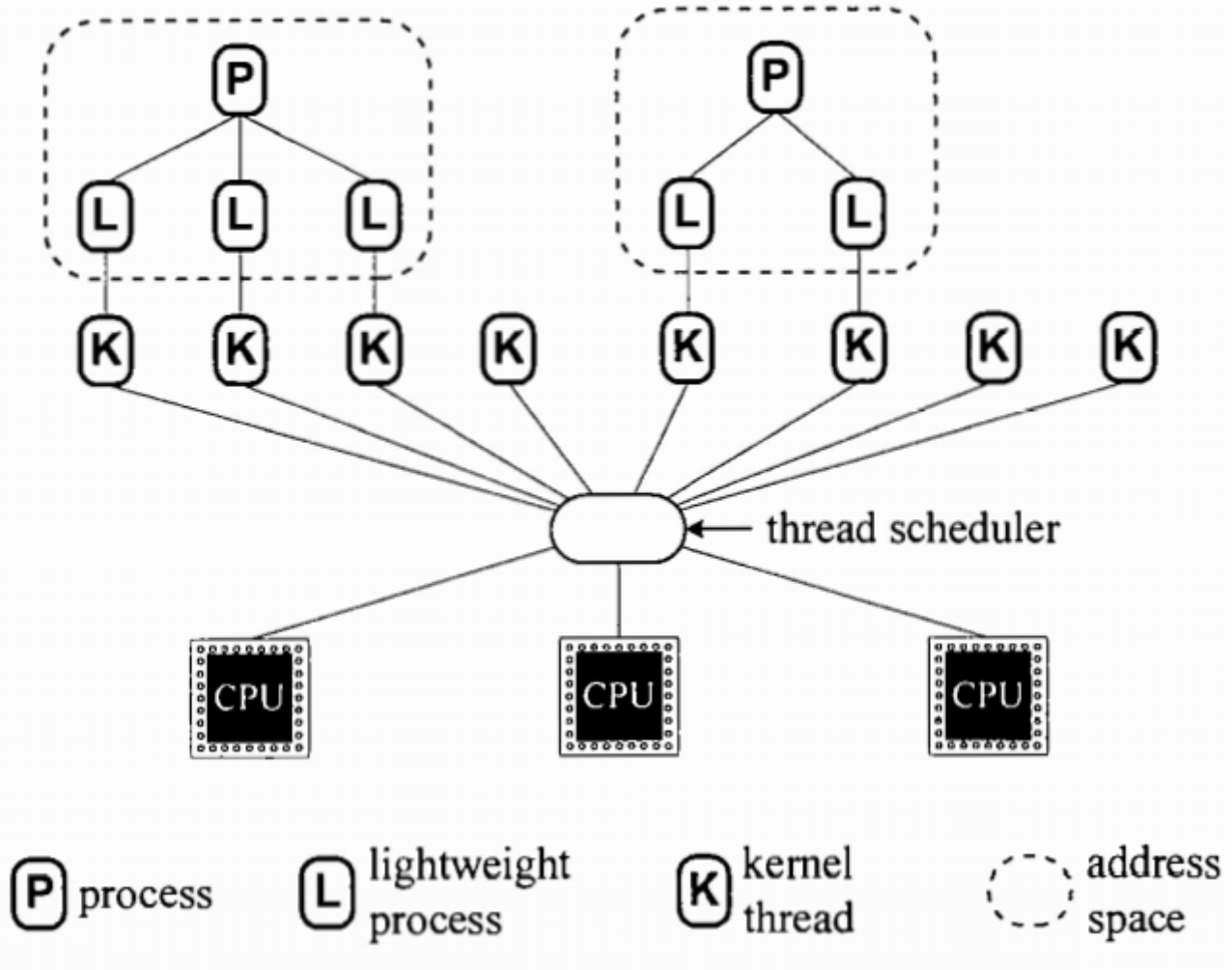
Kernel threads

- 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 ejecuta en kernel space

Lightweight processes (1:1)

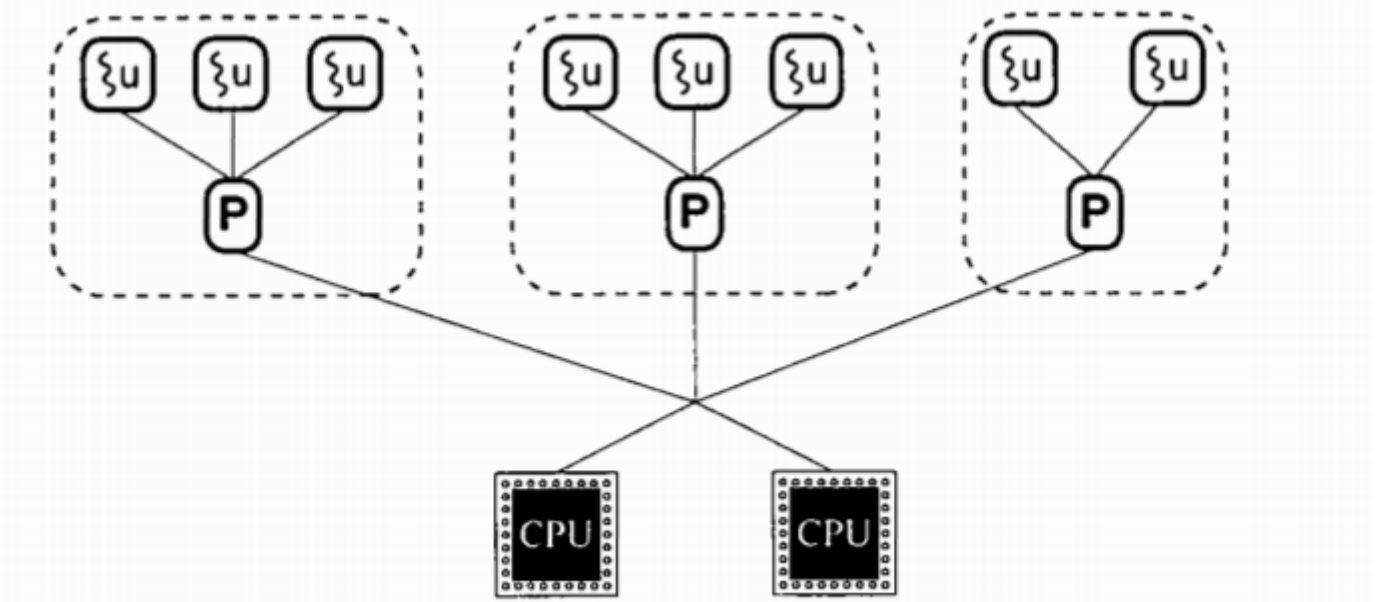
Es un thread en user space manejado por un kernel thread

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



User threads (N:1)

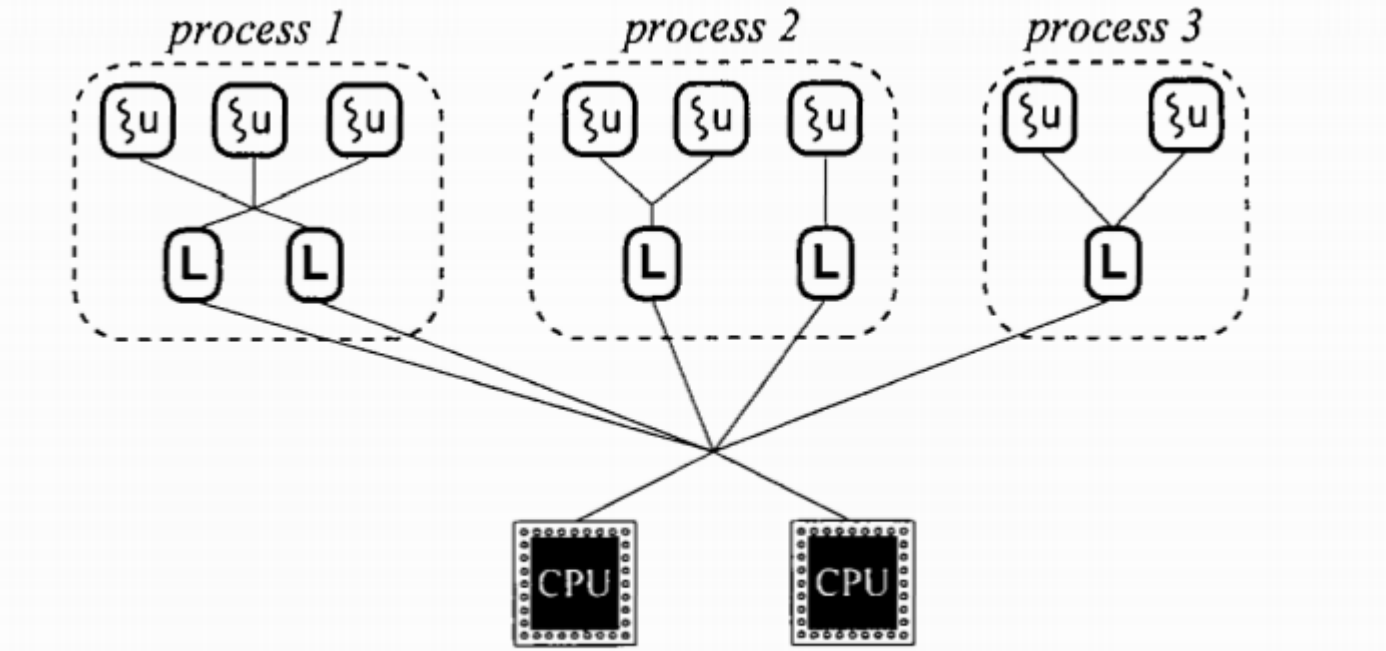
- Abstracción solo a nivel del usuario, el kernel no sabe que están estos threads.
- "Green threads" en lenguajes con máquina virtual
- No requiere comunicación con el kernel, no existen syscalls
- No se aprovecha el multicore



(a) User threads on top of ordinary processes

Hybrid threads (N:M)

- Es posible combinar user threads con lightweigh 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

PThreads

- Standard **POSIX** para manejo de threads en Unix.
- En Linux se utiliza la biblioteca **NPTL** (Native Posix Thread Library)
- Arquitectura 1:1
- Usa syscall **clone()** para crear los threads

Pthreads C API

- Manejo de threads
- Sincronización

```
#include <pthread.h>
```

```
gcc -Wall -Werror -pthread beard.c -o beard
```

Creación

```
pthread_t thread;

int ret;
ret = pthread_create (&thread,
                      NULL,
                      start_routine,
                      NULL);

if (!ret) {
   errno = ret;
   perror("pthread_create");
   return -1;
}
```

Creación

```
int pthread_create (
                    pthread_t* thread,
                    const pthread_attr_t *attr,
                    void* (*start_routine) (void *),
                    void* arg
                    );
```

```
void* start_thread (void* arg)
{

}
```

Finalización

- Existen 3 formas de finalizar un thread:
  - Si el thread retorna de su función inicial (start_thread)
  - Si el thread ejecuta **pthread_exit()**
  - Si otro thread ejecuta **pthread_cancel()** indicando el thread a finalizar.

```
void pthread_exit (void* retval);

int pthread_cancel (pthread_t thread);
```

Join (equivalente a wait)

```
void* start_thread (void* message)
{
   printf ("%s\n", (const char *) message);
   sleep(1);
}

int main (void)
{
   pthread_t thing1, thing2;
   const char* msg1 = "Thing 1";
   const char* msg2 = "Thing 2";

   pthread_create (&thing1, NULL, start_thread, (void*) msg1);
   pthread_create (&thing2, NULL, start_thread, (void*) msg2);

   pthread_join (thing1, NULL);
   pthread_join (thing2, NULL);
   return 0;
}
```

Comentar y volver a probar

Join (equivalente a wait)

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

- Recibe el thread a esperar.
- Recibe un puntero a un puntero void*, para dejar escrito allí el puntero void* que devuelve el thread (ver pthread_exit()).
- Devuelve 0 si no hubo errores.

Join Vs Detached

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

- Por default un thread en "Joineable"
  - No liberará los recursos (P. ej. Valor de retorno) hasta que no se llame a join.
- Existe un modo "Detached":
  - Al terminar libera sus recursos.
  - No se utiliza join.
  - Se debe marcar detached con pthread_detach()

Mutexes

- Evitan el acceso concurrente a un recurso

- Se crea una variable del tipo mutex
- Mediante una función, un thread "toma" el mutex
- Si otro thread intenta hacer lo mismo, este se bloqueará
- Cuando el thread que "tomó" el mutex lo libera, el thread bloqueado continuará su ejecución "tomando" el mutex que no había podido tomar previamente.

Mutexes

```
typedef struct {
   int a;
   int b;
}Data;

Data data;

void* start_thread (void* message)
{
   while(1)
   {
      data.a++;
      data.b++;
      usleep(100);
   }
}
```

Mutexes

```
int main (void)
{
   pthread_t t1,t2;
   int a,b;

   pthread_create (&t1, NULL, start_thread,NULL);
   pthread_create (&t2, NULL, start_thread,NULL);

   while(1)
   {
      a = data.a;
      b = data.b;
      printf("%d %d\r\n",a,b);
      sleep(1);
   }
```

```
1 1
12548 12549
25055 25057
37583 37585
50188 50190
62767 62770
75344 75349
87888 87893
```

Sincronizamos el acceso a la variable

```
typedef struct {
   int a;
   int b;
}Data;

Data data;
pthread_mutex_t mutexData = PTHREAD_MUTEX_INITIALIZER;

void* start_thread (void* message) {
   while(1) {
      pthread_mutex_lock (&mutexData);
      data.a++;
      data.b++;
      pthread_mutex_unlock (&mutexData);
      usleep(100);
   }
```

Sincronizamos el acceso a la variable

```
while(1)
{
   pthread_mutex_lock (&mutexData);
   a = data.a;
   b = data.b;
   pthread_mutex_unlock (&mutexData);

   printf("%d %d\r\n",a,b);
   sleep(1);
}
```

```
1 1
12718 12718
25323 25323
37938 37938
50588 50588
63181 63181
75845 75845
88488 88488
```

Threads y signals

- Cuando el proceso recibe una signal, no está garantizado en qué thread se ejecutará el handler.
- Esto puede traer problemas de concurrencia.
- Es preferible asignar un thread conocido para la ejecución de los handlers.
- Utilizaremos la función **pthread_sigmask()** para bloquear signals en un thread.

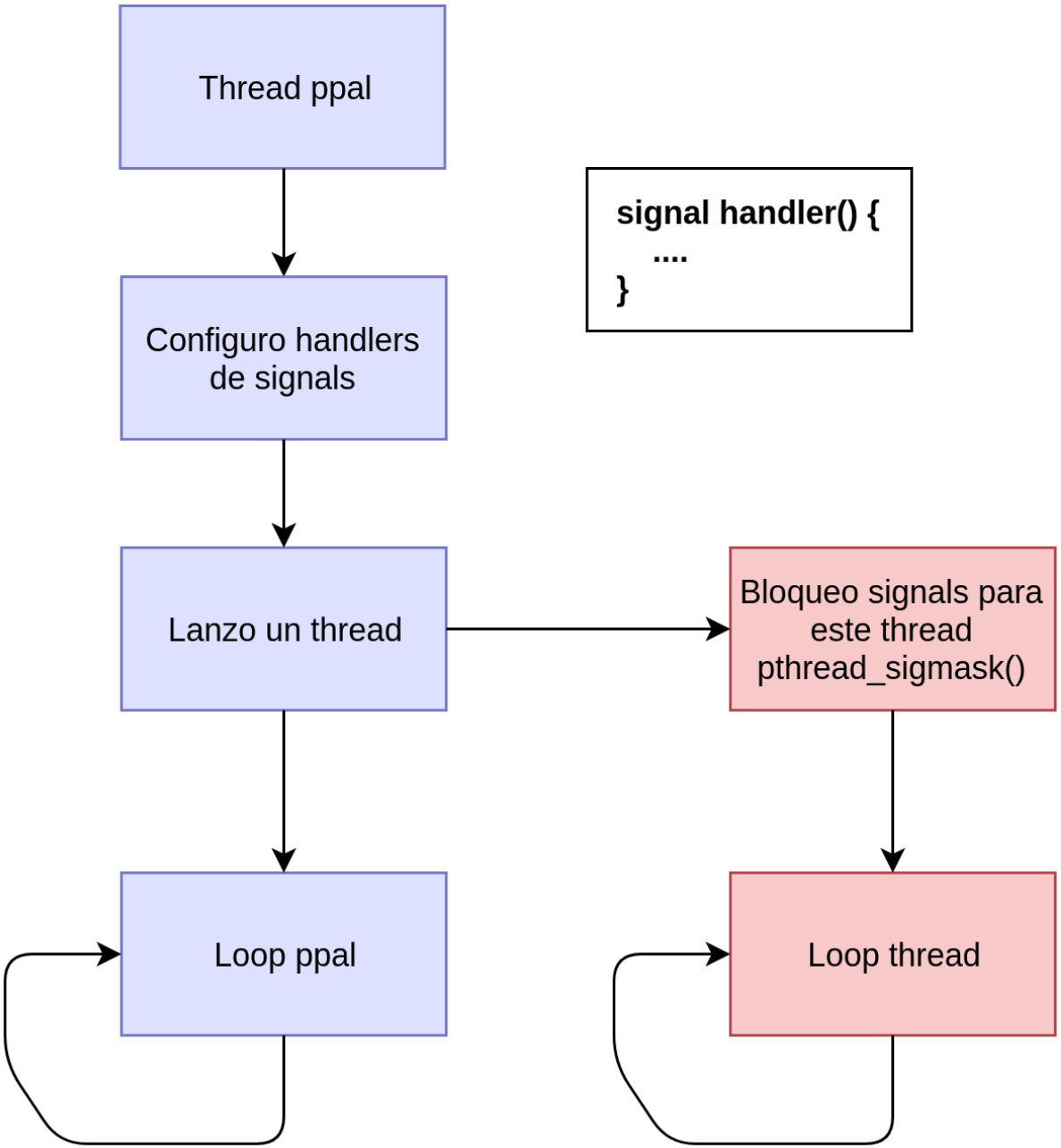
Threads y signals

```
int pthread_sigmask(int how, const sigset_t* set,
                                sigset_t* oset);
```

- Modifica la máscara de signals del thread desde el cual se invoca.

- **how**: SIG_BLOCK o SIG_UNBLOCK o SIG_SETMASK.
- **set**: Conjunto de signals a alterar según “how”.
- **oset**: Almacena la máscara anterior.

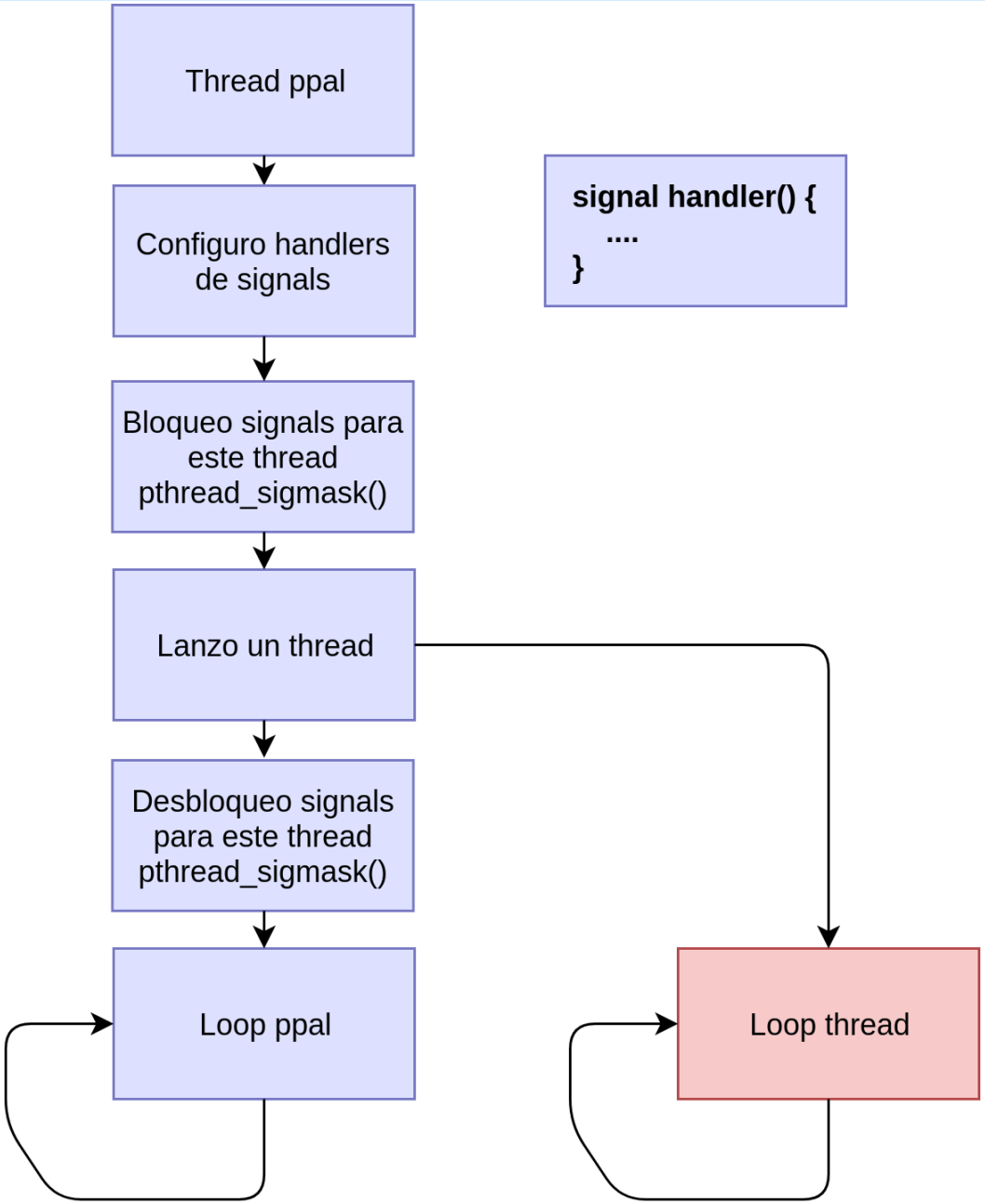
Ejemplo 1: ¿Es correcto?



Threads y signals

- Al crear un thread, el mismo heredará la máscara de signals del thread que lo creó.
- Una signal quedará pendiente si el thread la bloqueó, hasta que la desbloquee.
- Solo puede quedar pendiente una sola signal de cada tipo (no hay cola).

Ejemplo 2: Forma correcta



Threads y signals

```
void bloquearSign(void)
{
    sigset_t set;
    int s;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    //sigaddset(&set, SIGUSR1);
    pthread_sigmask(SIG_BLOCK, &set,
                                    NULL);
}
```

Threads y signals

```
void desbloquearSign(void)
{
    sigset_t set;
    int s;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    //sigaddset(&set, SIGUSR1);
    pthread_sigmask(SIG_UNBLOCK, &set,
                                 NULL);
}
```

Bibliografía

- Uresh Vahalia. (1996). UNIX Internals. The New Frontiers. New Jersey, Prentice Hall.
- Robert Love. (2013). Linux System Programming 2nd Edition. USA, O'Reilly Media.