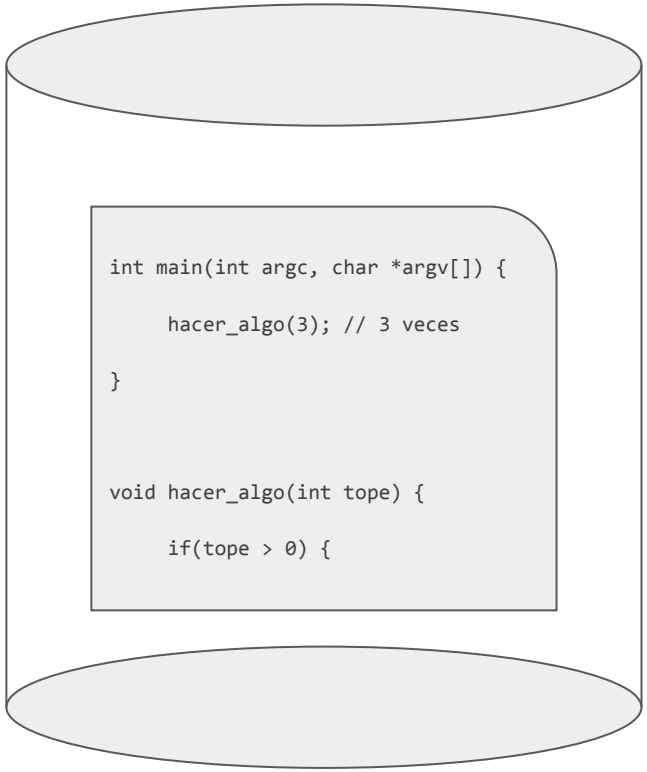
Procesos / Hilos

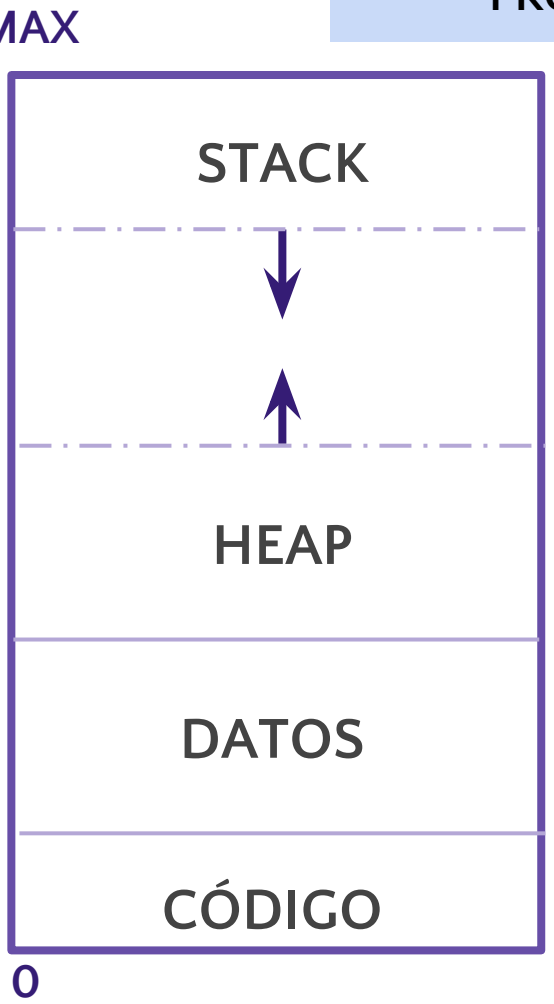
- ➔ Múltiples programas pueden ser cargados en memoria y ejecutados concurrentemente
- ➔ Un proceso es un programa en ejecución (que incluye más que el código del mismo)
- ➔ Es una entidad activa
- ➔ Es la mínima unidad de planificación

PROGRAMA



DISCO

PROCESO



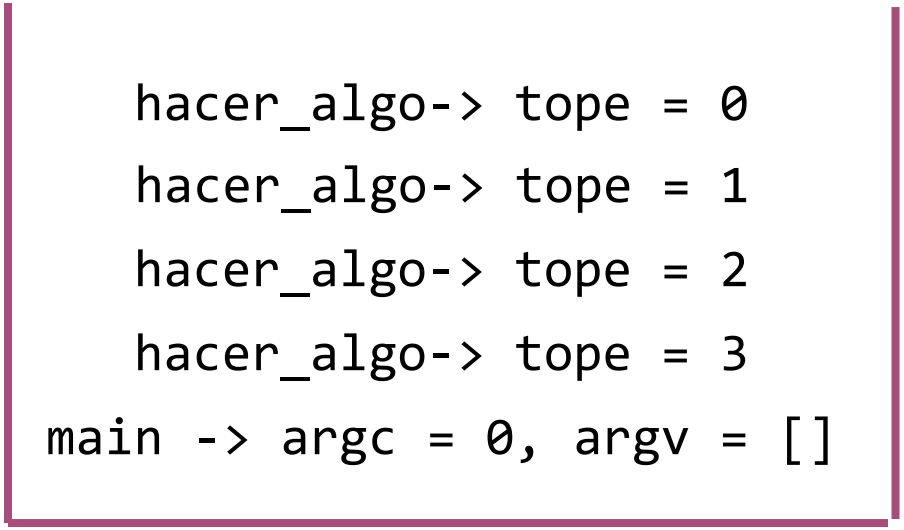
RAM

PCB

Imagen del proceso

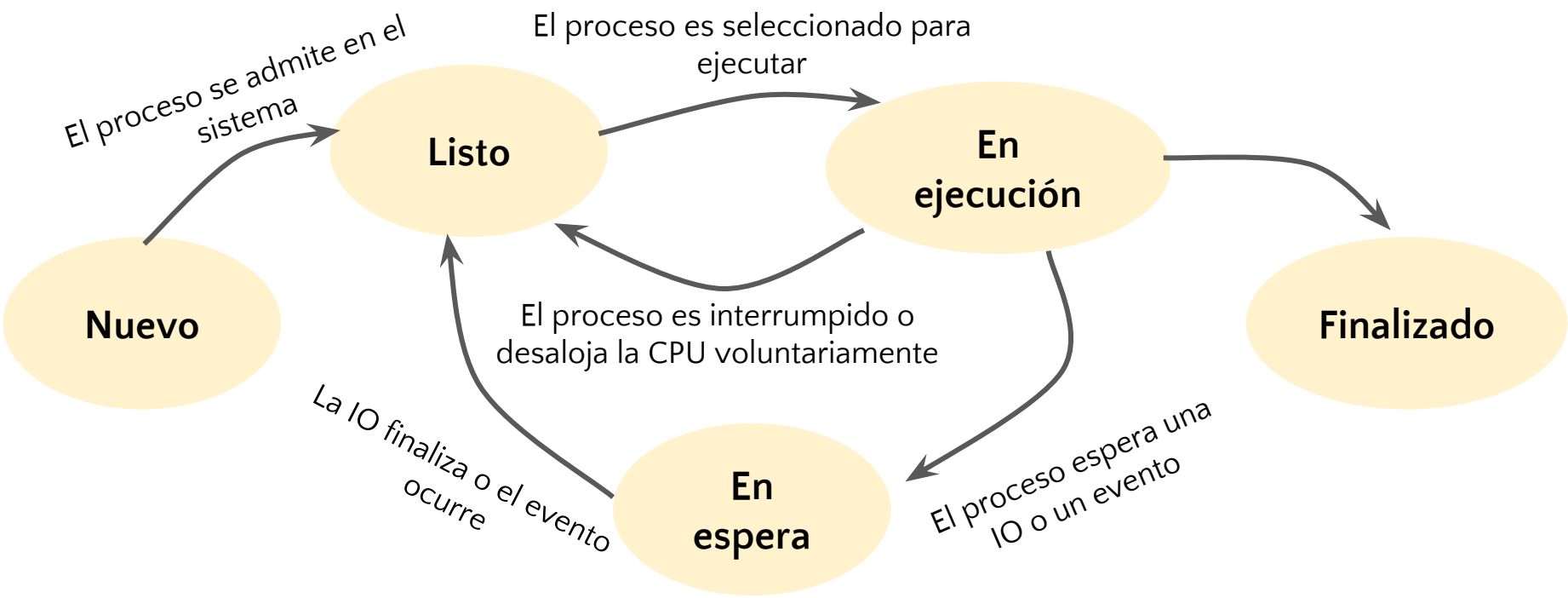
EJEMPLO DE STACK

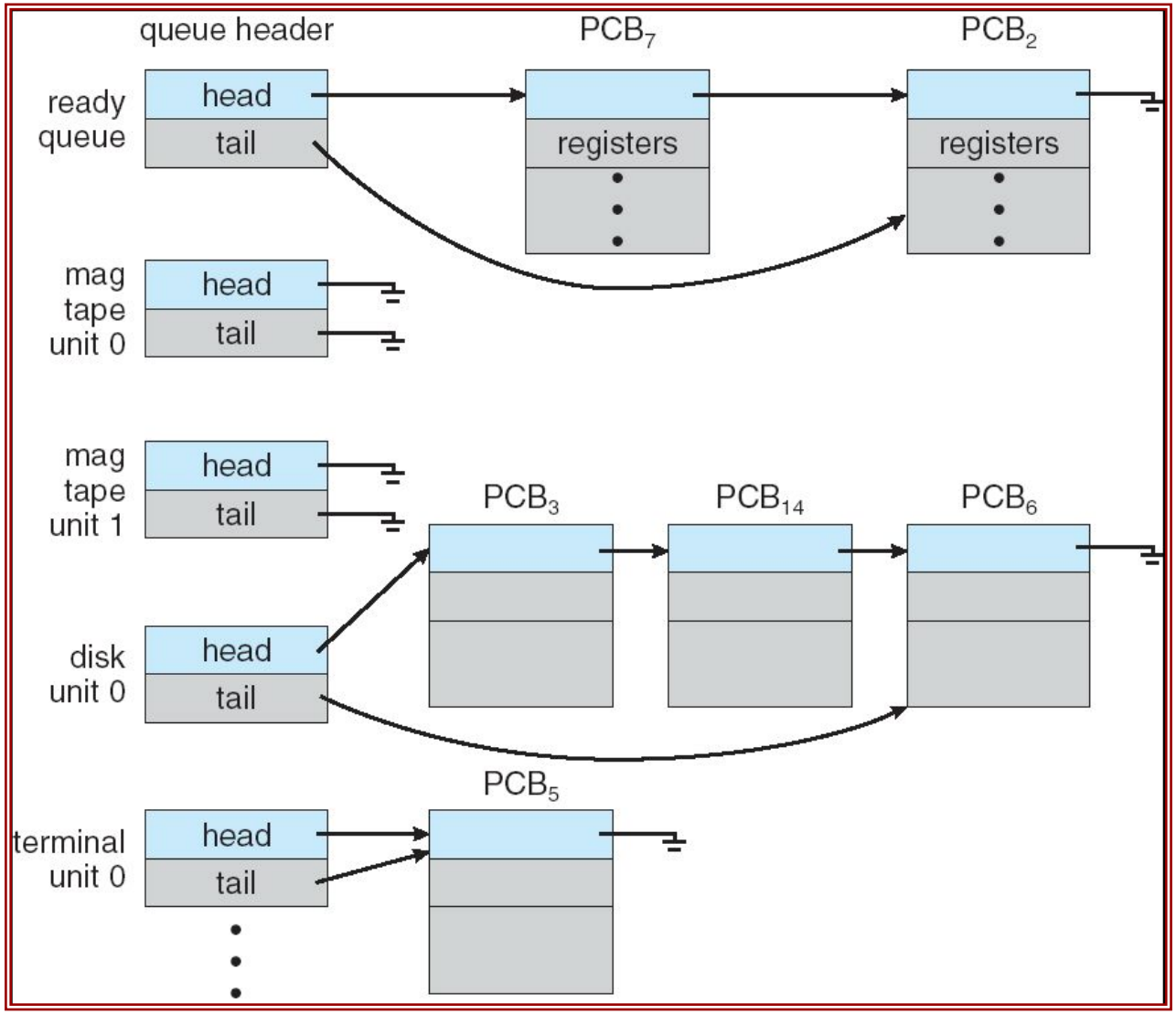
```
int main(int argc, char *argv[]) {
     hacer_algo(3); // 3 veces
}

void hacer_algo(int tope) {
     if(tope > 0) {
          hacer_algo(tope – 1);
     }
}
```



STACK

ESTADOS DEL PROCESO




queue header
PCB7
PCB2
ready queue
head
tail
registers
registers
mag tape unit 0
head
tail
mag tape unit 1
head
tail
PCB3
PCB14
PCB6
disk unit 0
head
tail
PCB5
terminal unit 0
head
tail

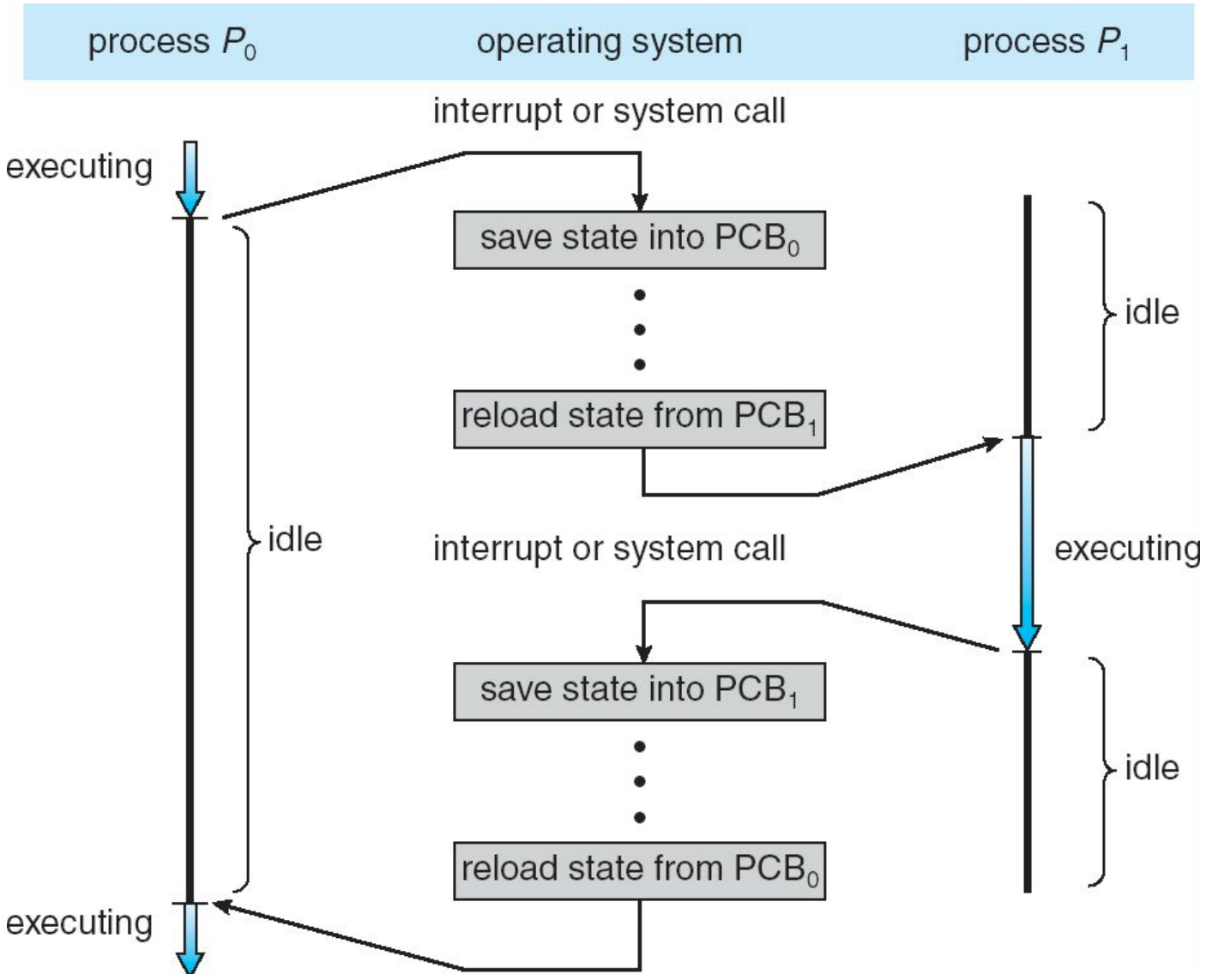
CAMBIO CONTEXTO

- Cuando se cambia el proceso que posee la CPU se debe guardar su contexto de ejecución para luego poder reanudarlo en el lugar interrumpido

- El cambio de contexto puede tener como objetivo
  - Ejecutar otro proceso
  - Atender una interrupción (ejecutará el interrupt handler)
  - Ejecutar una syscall

- El tiempo que dura un **cambio de contexto** es un gasto extra; es necesario para poder ejecutar procesos pero durante el mismo el sistema no hace ningún trabajo útil (para el usuario). Por lo tanto:
  - Este tiempo es considerado **Overhead**
  - Se debe minimizar

CAMBIO CONTEXTO - CAMBIO PROCESO



- ➔ 1 Process switch  2 Context switch (se guarda el ctx de un proceso y se restaura el de otro)
- ➔ 2 Context switch  1 Process switch (Se puede elegir al mismo proceso, ejecutar una syscall, atender una interrupción)
- ➔ 1 Mode switch  1 Context switch (paso de ejecutar un proceso de usuario a ejecutar el SO o viceversa)
- ➔ 1 Context switch  1 Mode switch (puede ocurrir una interrupción cuando ya estoy atendiendo una)

Process control block

- ➔ Posee la información necesaria para que el SO administre al proceso
- ➔ Se encuentra siempre cargado en RAM

Estado del proceso
PC/IP
Registros CPU
Información de planificación de CPU
Información de manejo de memoria
Información contable
Información de estado de E/S

CREACIÓN DE PROCESOS

A través de una syscall, un proceso padre puede crear un proceso hijo
Los procesos serán identificados por un id (**PID**)

Recursos (CPU - Memoria - archivos - dispositivos)

➔ Puede obtenerlo pidiéndolo al SO

➔ Puede estar restringido a un subset de los recursos del proceso padre

Cuando un proceso crea otro puede ocurrir

➔ El padre continúa la ejecución concurrentemente con el hijo

➔ El padre se queda esperando a que todos o alguno/s de los hijos finalicen

Con respecto al espacio de direcciones del proceso, puede ocurrir:

➔ El proceso hijo es una exacta copia del padre

➔ Al proceso hijo se le pasa una nueva imagen que reemplaza la anterior

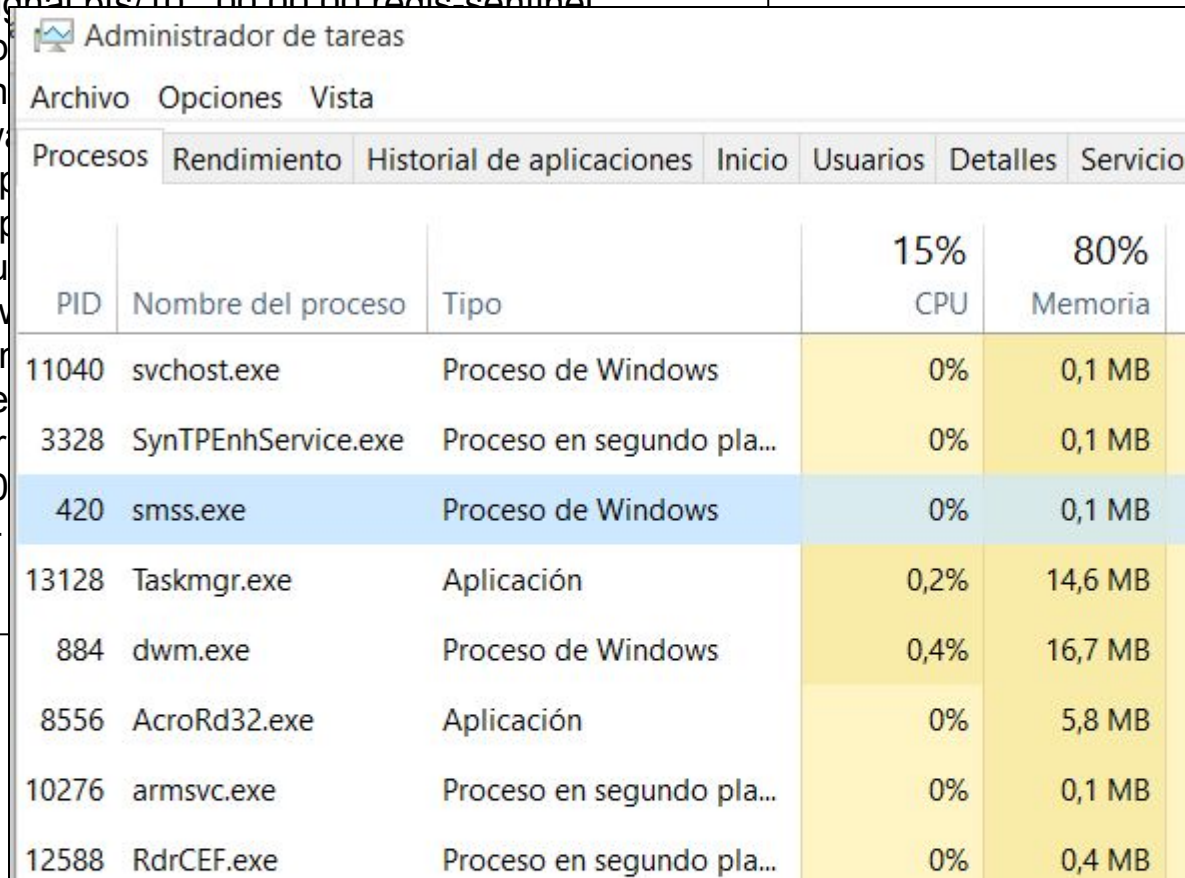
```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
   int pid;
     fork_result = fork();
     if (fork_response < 0) {
          fprintf(stderr, “Falló");
          exit(-1);
     }    else if (fork_result == 0) {
          /* Inicio de código de Proceso Hijo */
          execlp("/bin/ls“,"ls”,NULL);
          .../* Fin de código de Proceso Hijo */
     }    else {
          /* Inicio de código de Proceso Padre */
          ...
          /* Esperar término del Hijo */
          wait(NULL);
          printf("Child Complete");
          exit(0);
          ...
          /* Fin de código de Proceso Padre */
     }
}
```

FINALIZACIÓN DE PROCESOS

- Un proceso le indica al SO que quiere terminar con una syscall (**exit**)
  - Se envía al padre información de salida, su result status (via **wait**)
  - Los recursos usados por el proceso son liberados
- Un proceso padre puede terminar la ejecución de sus hijos (**abort**)
  - El hijo se ha excedido en el uso de recursos asignados
  - La tarea que realiza el hijo no es ya necesaria
  - El padre va a terminar
    - Algunos SOs no permiten que un hijo siga ejecutando si su padre termina. Cuando un proceso finaliza, todos sus hijos son finalizados– ***terminación en cascada***
    - Otros SOs transfieren el hijo al padre (del padre)

VISUALIZACIÓN DE PROCESOS

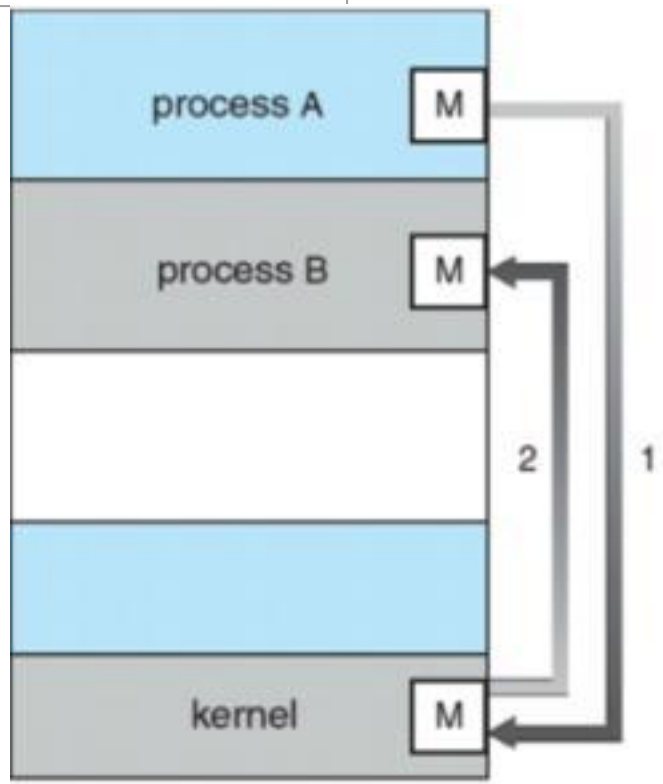
```
~$ ps -el
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 -  1114 poll_s ?        00:00:01 init
1 S     0     2     0  0  80   0 -     0 kthrea ?        00:00:00 kthreadd
0 S  1001  9861  9850  0  80   0 -  1843 n_tty_ pts/19   00:00:00 bash
0 S  1001  9917  9081  0  80   0 -  7561 ep_pol pts/0    00:00:00 redis-server
0 S  1001  9924  9125  0  80   0 -  7561 ep_pol pts/5    00:00:00 redis-server
0 S  1001  9936  9106  0  80   0 -  7561 ep_pol pts/4    00:00:02 redis-sentinel
0 T  1001  9943  9163  0  80   0 -  7561 signal pts/10   00:00:00 redis-sentinel
0 S  1001  9951  1688  0  80   0 - 60332 p
0 S  1001  9960  9951  0  80   0 -   605 un
0 S  1001  9962  9951  0  80   0 -  1826 w
0 S  1001  9985  9182  0  80   0 -  7561 ep
0 S  1001 10013  1688  0  80   0 - 60323 p
0 S  1001 10022 10013  0  80   0 -   605 u
0 S  1001 10024 10013  0  80   0 -  1826 v
0 S  1001 10045 10024  0  80   0 -  2778 r
0 S  1001 10046  9163  0  80   0 -  7561 e
1 S     0 10054     2  0  80   0 -     0 worker
0 S  1001 10060  1515  3  80   0 - 141220
0 R  1001 10106  9962  0  80   0 -  1261 -
```

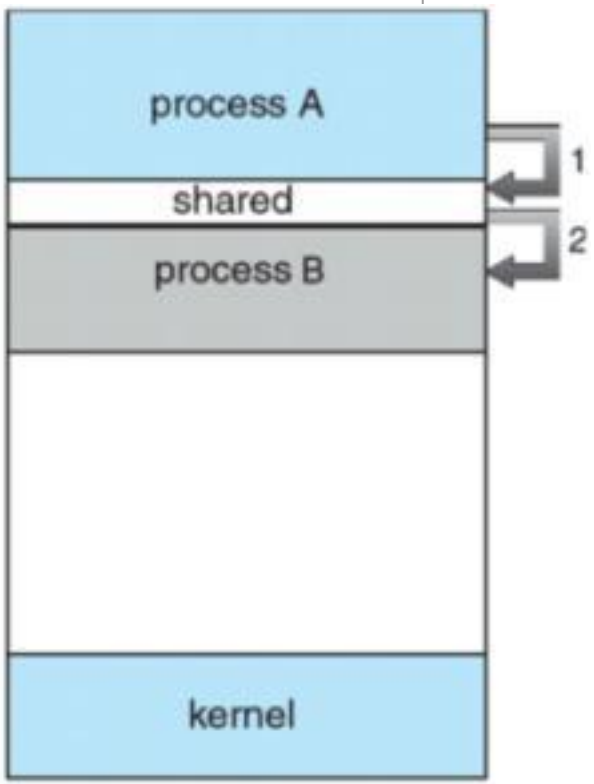


- Los procesos que no afectan ni pueden ser afectados por otros procesos son independientes
- Los procesos que comparten recursos son llamados cooperativos

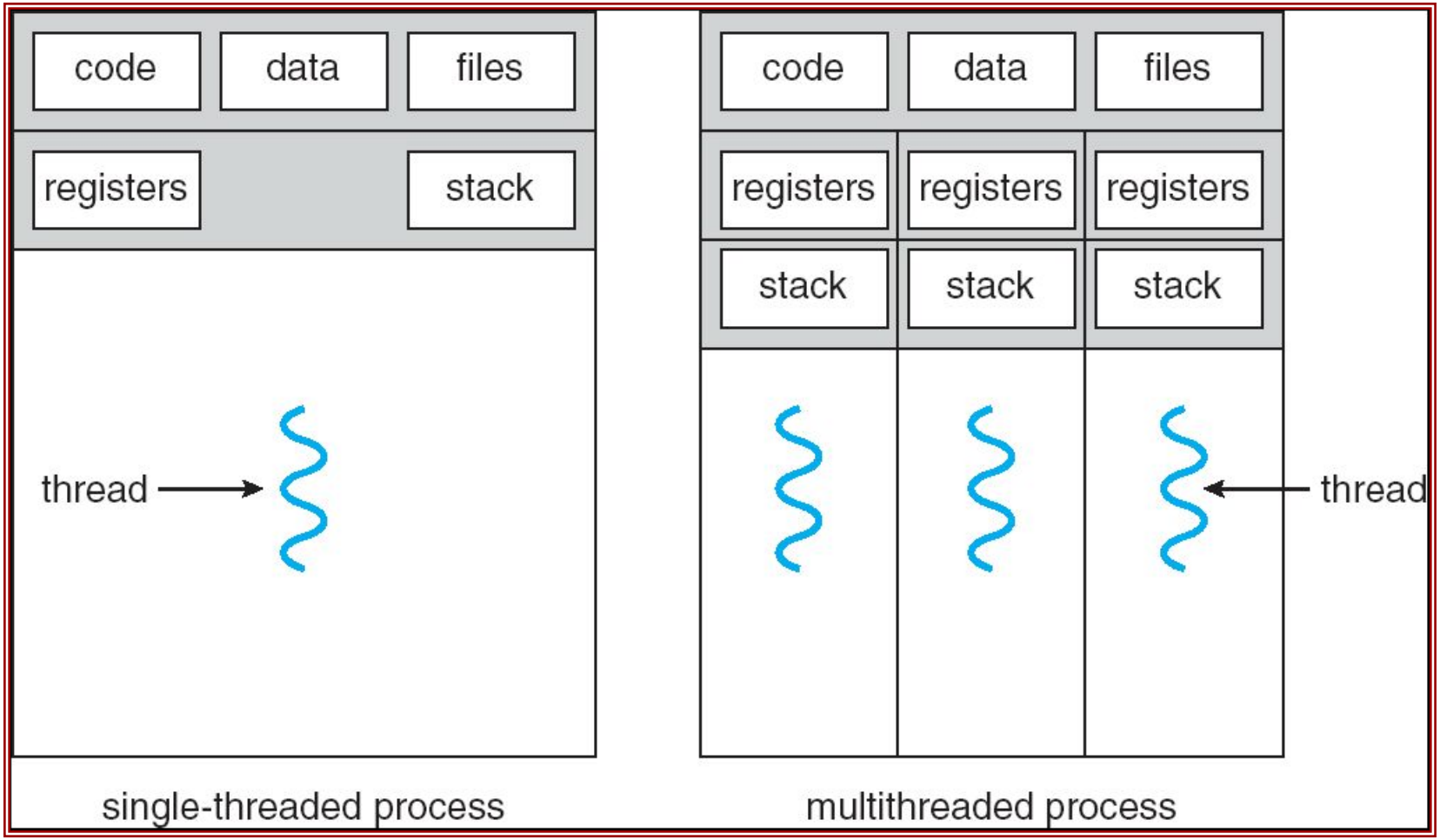
- Los procesos cooperativos requieren técnicas de IPC para poder comunicarse
  - Memoria compartida: los procesos establecen una zona de memoria compartida la cual se usa para comunicarse leyendo y escribiendo datos en la misma
  - Paso de mensajes

PASO DE MENSAJES	MEMORIA COMPARTIDA
Es útil para intercambiar pequeñas cantidades de datos Es más lento ya que requiere cambios de contexto (syscalls)	Permite una comunicación más rápida Luego de que la región compartida es establecida, no requiere intervención del SO





- Un **hilo** o **thread** es una unidad básica de utilización de la CPU y consiste en un juego de registros y un espacio de pila.
- Comparte el **código, los datos y los recursos** con los otros hilos del mismo proceso
- Cada hilo posee su sección de **stack**
- Cada hilo es administrado por un **TCB**, cuya referencia se encuentra en el **PCB** del proceso al que está asociado
- Al compartir recursos, pueden comunicarse sin usar ningún mecanismo de comunicación inter-proceso del SO
- No hay protección entre hilos. Un hilo puede escribir en la pila de otro hilo del mismo proceso


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );

main() {
     pthread_t hilo1, hilo2;
     char *message1 = “Hilo 1";
     char *message2 = “HIlo 2";
    int iret1, iret2;

     iret1 = pthread_create( &hilo1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &hilo2, NULL, print_message_function, (void*) message2);
     pthread_join( hilo1, NULL);
     pthread_join( hilo2, NULL);

     printf(“El Hilo 1 devuelve: %d\n",iret1);
     printf(“El Hilo 2 devuelve: %d\n",iret2);
     exit(0);
}
void *print_message_function( void *ptr ) {
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Ventajas

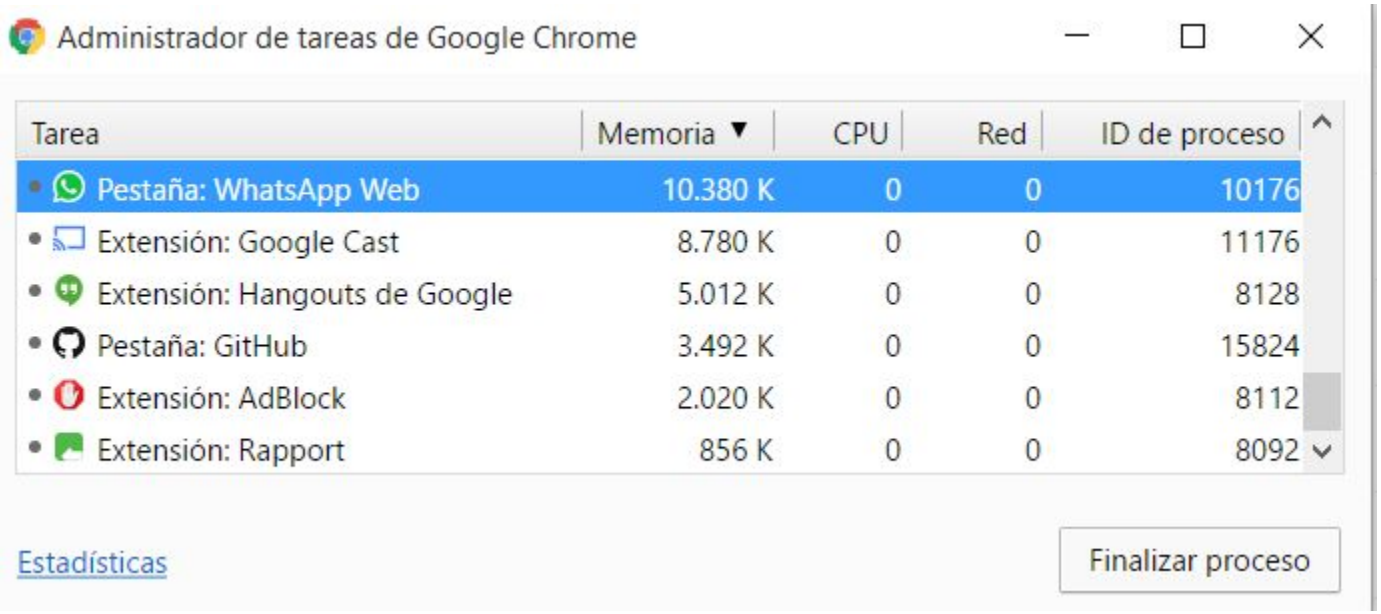
- Permiten paralelismo dentro de un proceso o aplicación.
- Permiten la comunicación privada entre varios hilos del mismo proceso, sin solicitar intervención del S.O. (es más rápida)
- Mayor eficiencia en el cambio entre hilos (del mismo proceso)
- Mayor eficiencia en la creación de un Hilo que en la creación de un Proceso Hijo.
- Un Proceso Multihilo puede recuperarse de la "muerte" de un Hilo, pues conoce los efectos de esta, y toma su espacio de memoria (excepto para el main).

Desventajas

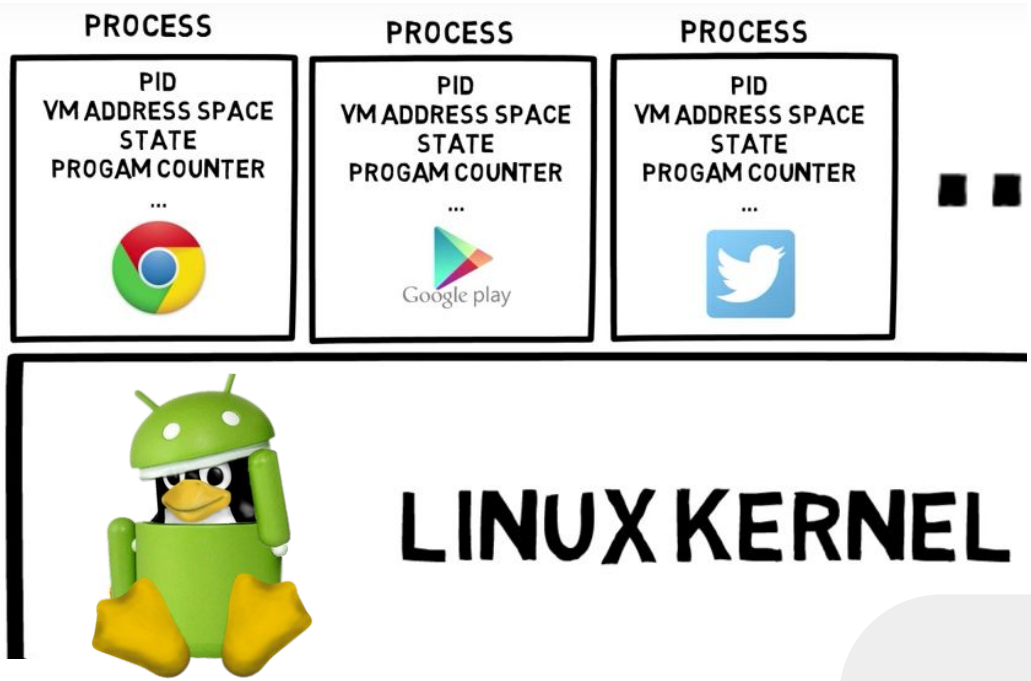
- ➔ Cuando un Proceso “muere” todos sus Hilos también, pues los recursos de Proceso son tomados por el Sistema Operativo.
- ➔ Cuando un hilo muere, no se liberan automáticamente todos sus recursos
- ➔ Un problema en un hilo, puede afectar al resto
- ➔ No hay protección entre los distintos hilos

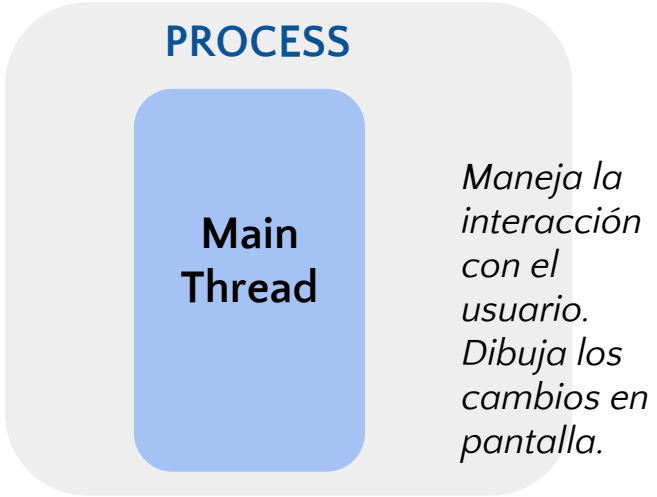
KLT (Kernel level threads)	ULT (User level threads)
• El SO los conoce, por los que los puede planificar • El cambio de entre KLTs es más lento (interviene SO) • Si un KLT se bloquea, cualquier KLT puede continuar • Pueden ser ejecutados en paralelo en múltiples CPUs ya que son unidades independientes de ejecución	• El SO “no los ve”, sino que son administrados por una biblioteca en nivel de usuario • El cambio entre ULTs (del mismo KLT) es más rápido (no interviene SO) • Si un ULT se bloquea, todos los ULTs asociados a la misma unidad de planificación se bloquean • No permite aprovechar de ejecución en paralelo en caso de múltiples CPUs

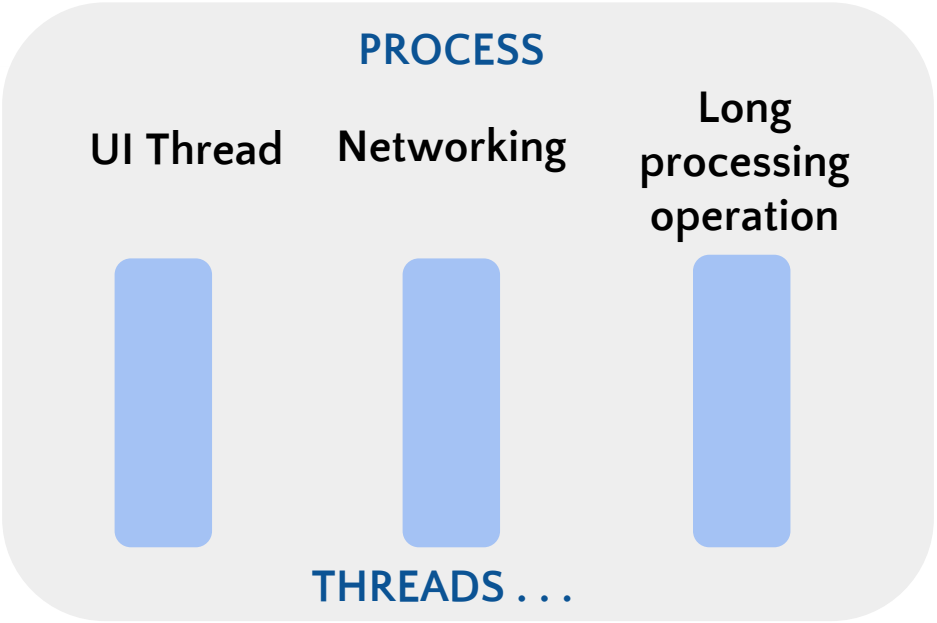
chrome



USO DE PROCESOS







PROCESOS/HILOS EN ANDROID


MULTI-THREADING
YOU DON'T SCARE ME ANYMORE