Introducción a Concurrencia

Concurrencia y Paralelismo

Juan Quintela
quintela@udc.es

Javier París
javier.paris@udc.es

¿Qué es?

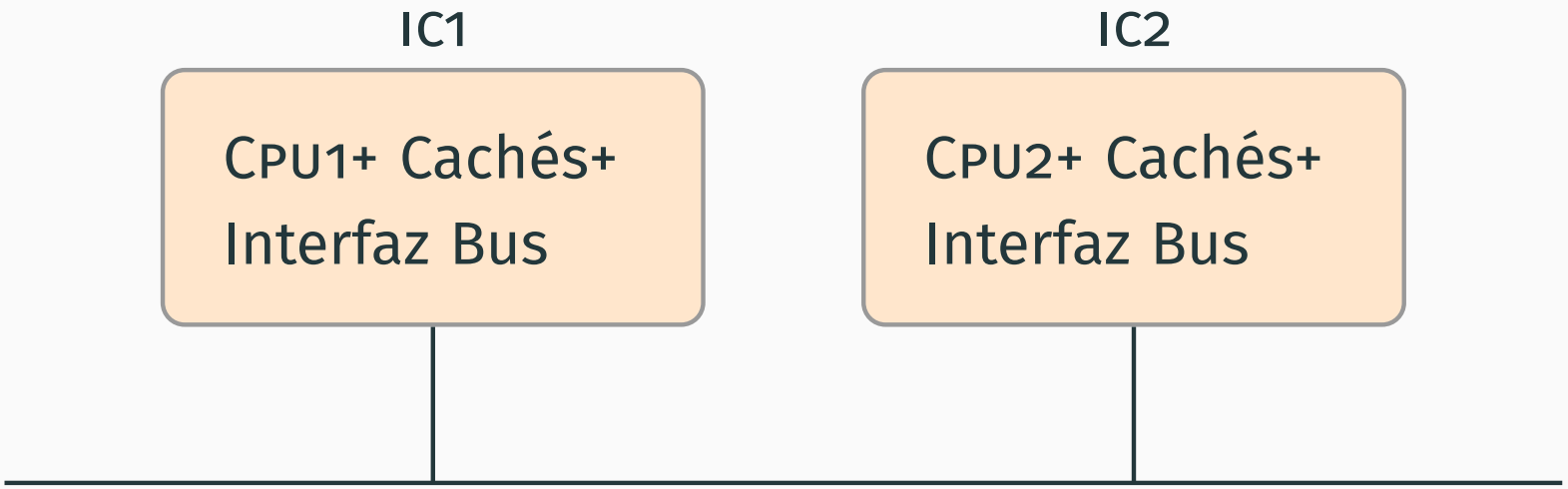
- Un Programa concurrente está diseñado para ejecutar varias tareas de forma simultánea.
- Las tareas pueden ser distintas.
- 2 Problemas principales:
  - Controlar el acceso a recursos compartidos entre tareas.
  - Comunicar a las tareas entre sí.

¿Para qué?

- Usar mejor el procesador cuando hay mucha entrada/salida (time-sharing). Mientras un proceso espera por un dispositivo otro puede usar el procesador.
- Modelar sistemas que son inherentemente concurrentes (por ejemplo, interfaces gráficas, software de control para gestionar varios dispositivos físicos).
- Aprovechar la potencia de los sistemas multiprocesador/multicore.

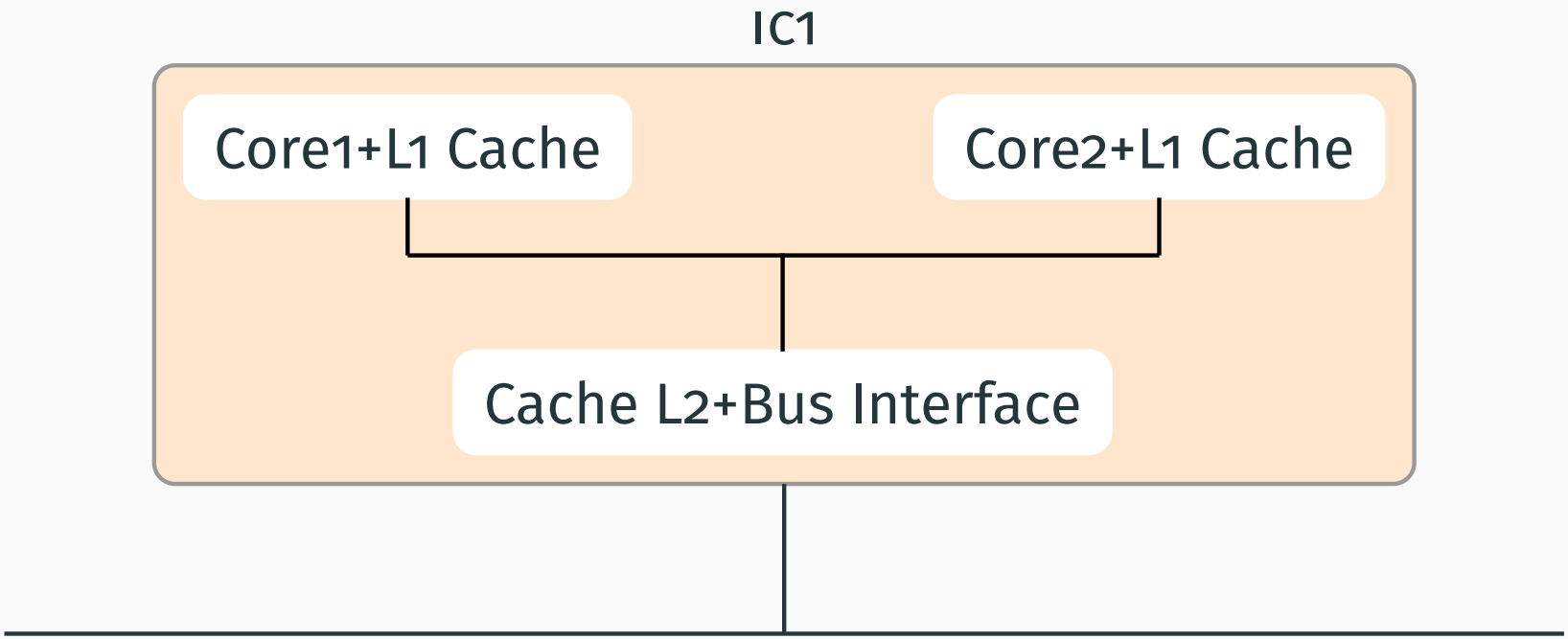
Arquitecturas HW: Multiprocesadores

- Sistemas Multiprocesador: Más de un procesador en el mismo sistema
- Cada procesador está en un circuito integrado.



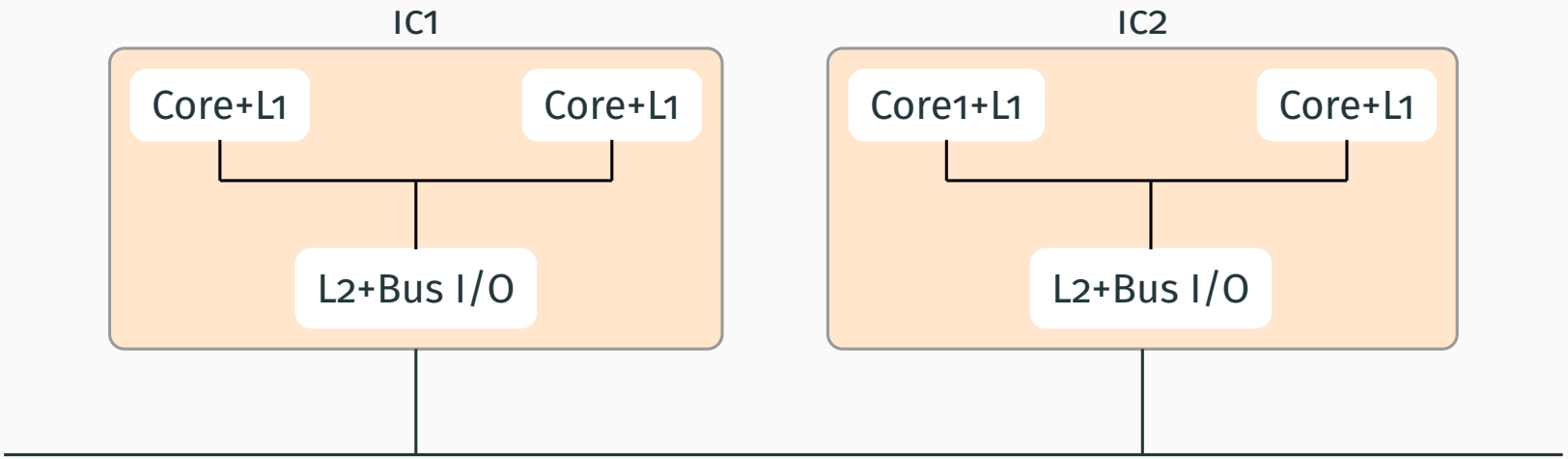
Arquitecturas HW: Multinucleos

- Sistemas Multinucleo: Más de un procesador dentro del mismo circuito integrado.
- Normalmente comparten caché e interfaz con el resto del sistema.



Arquitecturas HW:Multiprocesador+Multinucleo

- Los sistemas multiprocesador modernos normalmente también son multinucleo.



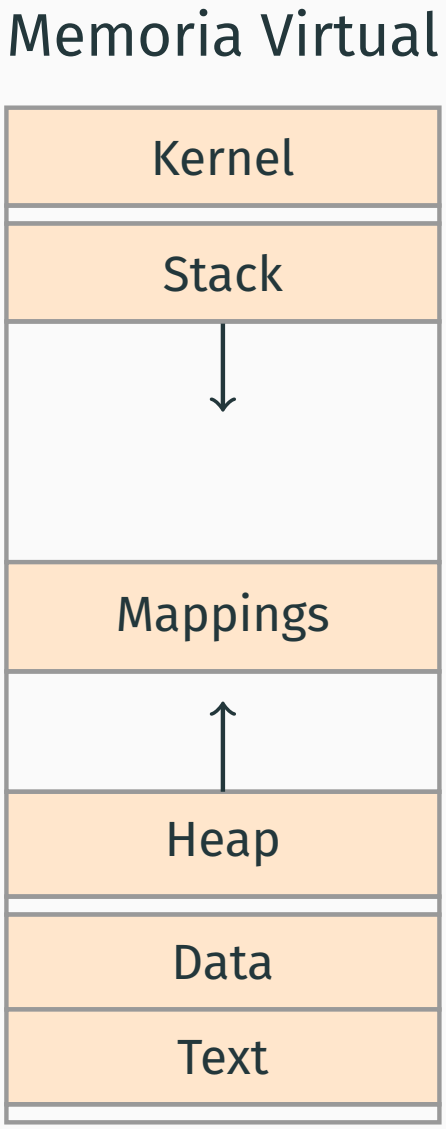
Arquitecturas HW: Multithreading / Hyperthreading

- Cada nucleo puede ejecutar más de una tarea concurrentemente.
- Tipos:
  - Temporal. El núcleo cambia periódicamente la tarea que está ejecutando. Permite al procesador utilizar tiempo de procesamiento donde normalmente se introducirían burbujas, por ejemplo, por accesos a memoria.
  - Simultánea. En núcleos superescalares se lanzan simultaneamente instrucciones de tareas distintas. Requiere duplicar partes del nucleo.

Arquitectura SW: Procesos

- Un proceso es una instancia de la ejecución de un programa.
- Puede haber más de un proceso ejecutándose a la vez para el mismo programa.
- Cada proceso tiene sus propios recursos:
  - Memoria.
  - Descriptores de fichero.
  - Sockets.
- Al crear un proceso se copian los recursos del padre.

Procesos: Estructura del Espacio de Direcciones Virtual



Memoria Kernel

Stack(variables locales, parámetros, ...)

Memoria gestionada con mmap

Memoria dinámica (malloc)

Variables Globales, constantes

Código del programa y librerías

Procesos: Estructura del Stack en linux x86_64

Registros:

- Caller-save: rax, rcx, rdx, rdi, rsi, rsp, r8-r11
- Callee-save: rbx, rbp, r12-r15

Llamada a función:

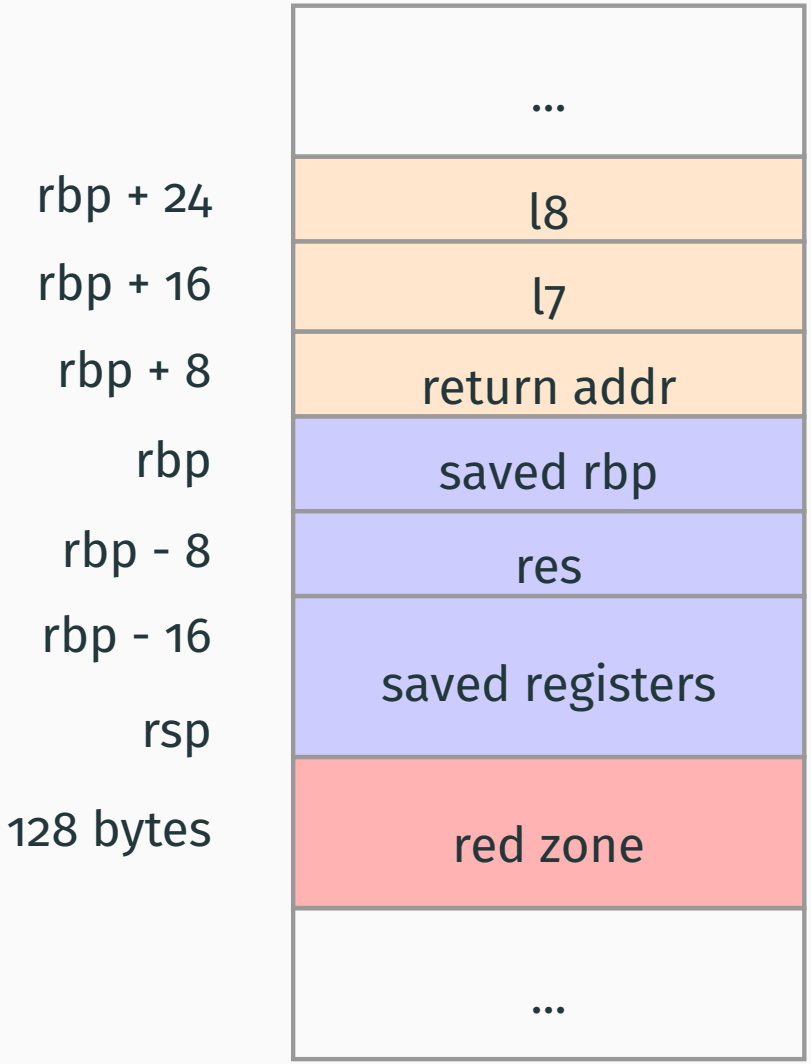
- Los 6 primeros paramétros int o puntero se pasan usando rdi, rsi, rdx, rcx, r8 y r9
- Otros parámetros van en el stack

Procesos: Estructura del Stack en linux x86_64

Ejemplo:

```
int f1(long l1, long l2, long l3, long l4,
       long l5, long l6, long l7, long l8) {
    long res;
    scanf("%ld", &res);
    res += l1+l2+l3+l4+l5+l6+l7+l8;
    return res;
}
```

Procesos: Estructura del Stack en linux x86_64



Rellenado por caller

Rellenado por callee

dirección donde saltar al terminar f1

valor de rbp para la función anterior

Si se llaman funciones guarda los registros caller-save

No se sobreescribe si se ejecuta una interrupción o un manejador de señal

l1, l2, l3, l4, l5 y l6 se pasan en rdi, rsi, rdx, rcx, r8 y r9

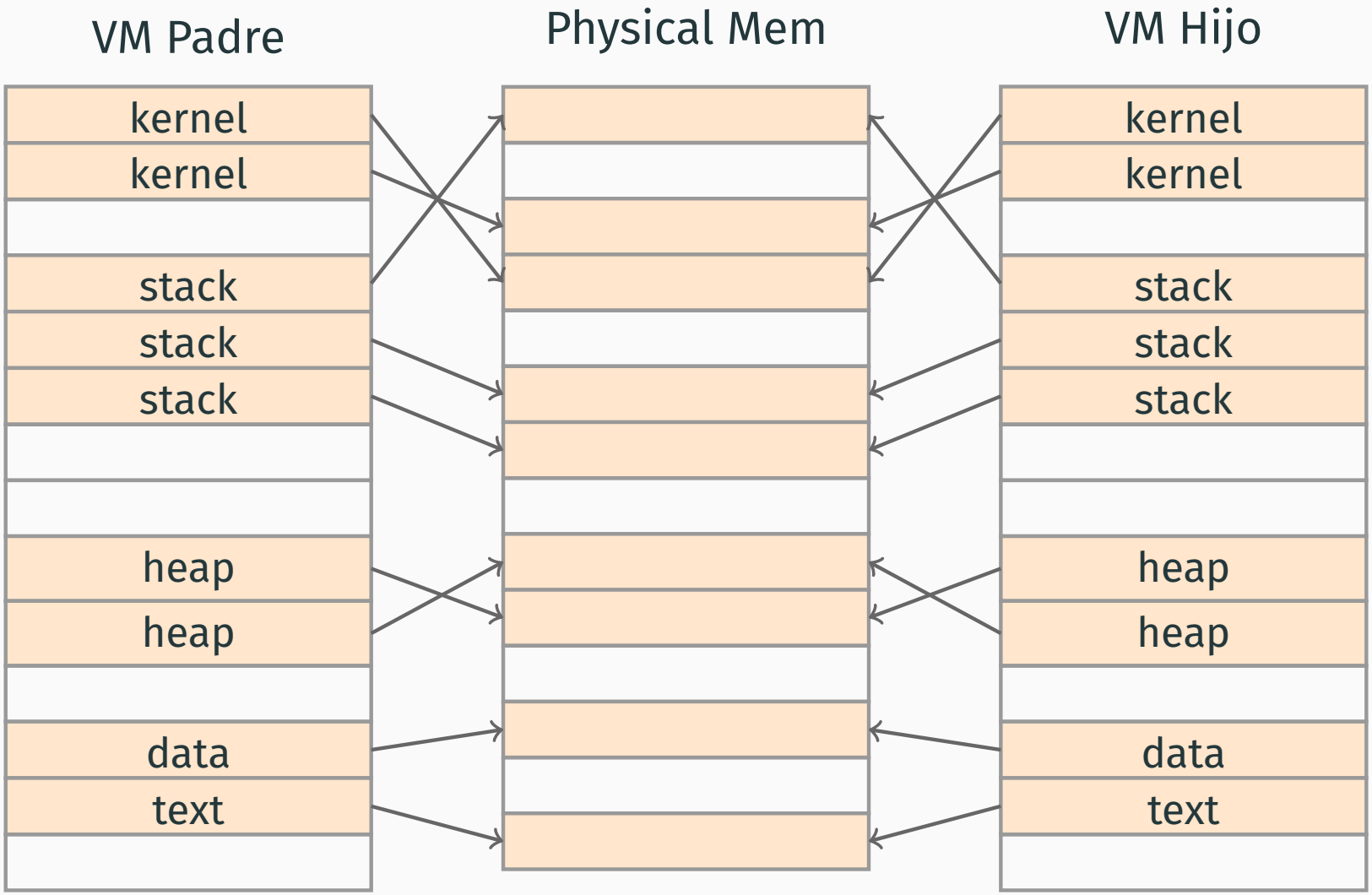
Procesos: Creación

Usando `pid_t fork(void)`:

Ejemplo:

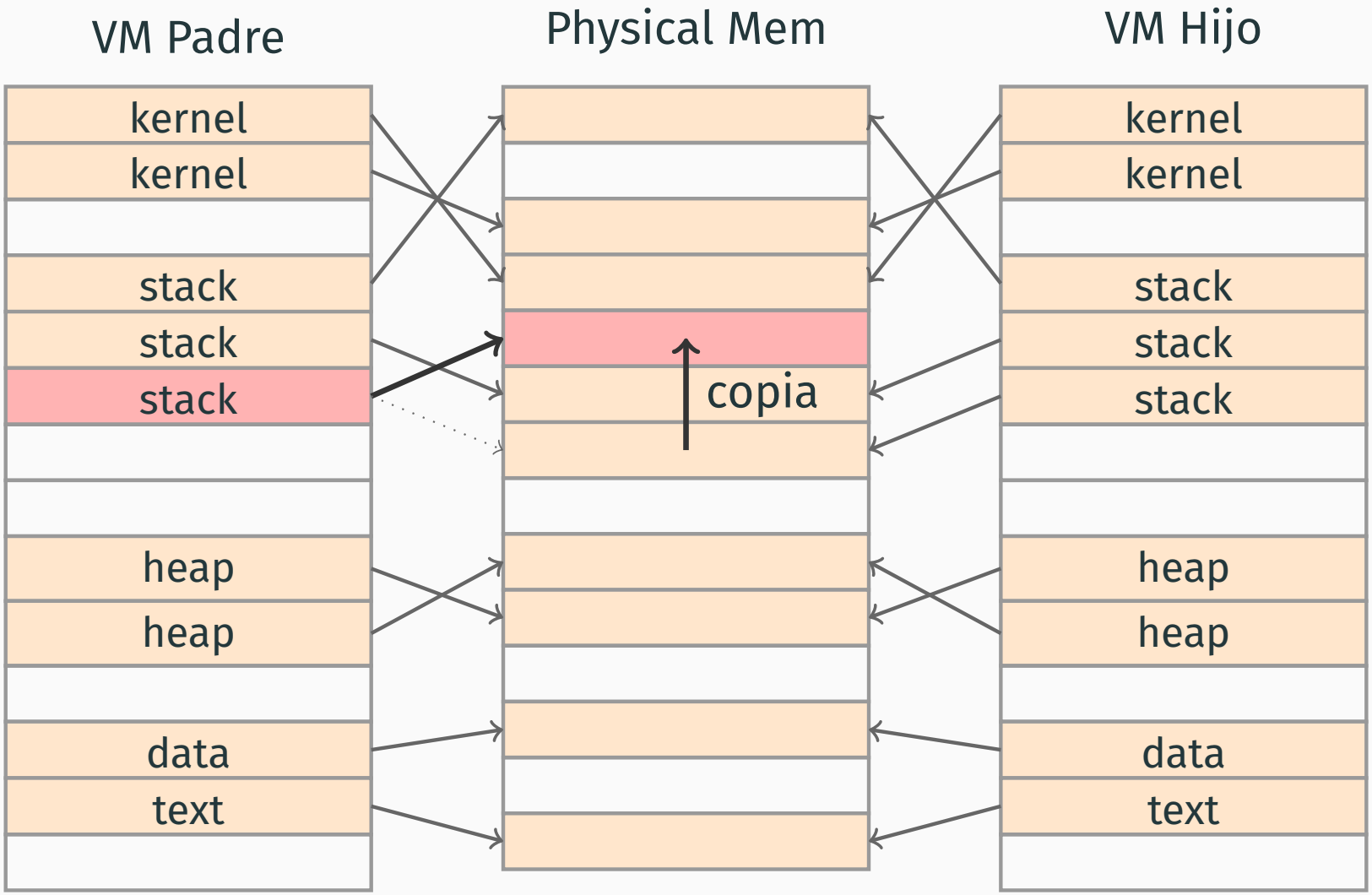
```
#include <sys/types.h>
#include <unistd.h>

int pid;
if((pid=fork())==0) { // Codigo hijo
   ...
} else { // Codigo padre
   ...
   waitpid(pid, NULL, 0);
}
```

Procesos: Creación



El kernel copia la tabla de páginas del padre.

Procesos: Creación

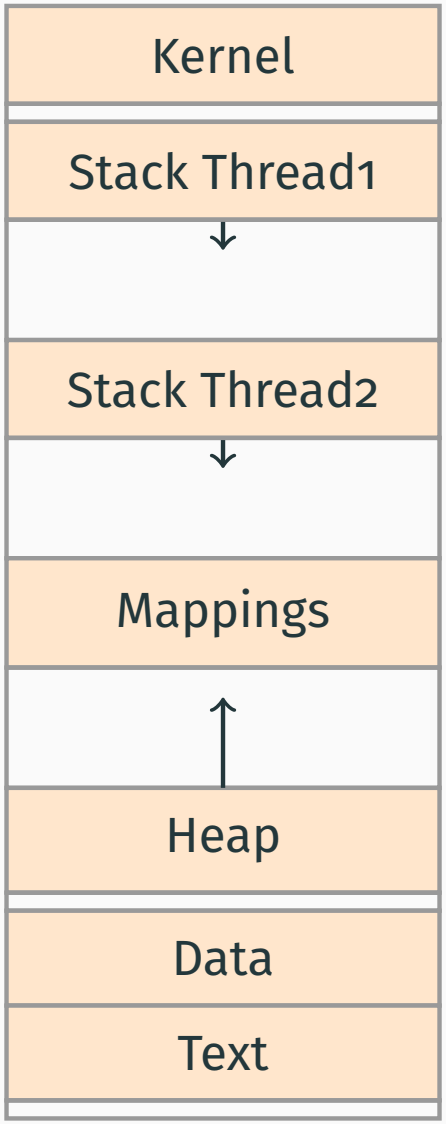


Una escritura provoca una copia de la página (copy on write).

Threads

- Un thread es un hilo de ejecución dentro de un proceso.
- Los threads de un proceso comparten:
  - Memoria. La tabla de páginas es la misma, con lo que ven la misma memoria, pero cada thread tiene un stack propio.
  - Descriptores de fichero.
  - Sockets.
  - Señales.

Threads: Espacio de Direcciones Virtual



Memoria Kernel

Stack Thread 1(variables locales, parametros,

Stack Thread 2(variables locales, parametros,

Memory asignada con mmap

Memoria dinámica (malloc)

Variables globales, constantes

Codigo

Threads: Creación

Creación

```
#include <pthread.h>
int pthread_create(
  pthread_t *thread,           // Identificador opaco
  const pthread_attr_t *attr, // Atributos del thread
  void *(*start_fun)(void *), // Función inicio del thread
  void *arg);                  // Parámetro de start_fun
```

Espera

```
int pthread_join(
  pthread_t thread,  // Thread por el que esperar
  void **retval);    // puntero para valor de retorno
```

Threads: Creación

Ejemplo: Función Thread

```
#include <pthread.h>
struct args {
    int i;
    char c;
};
struct thr_res {
    int res1;
    float res2;
};
void *thread_function(void *p) { // Funcion thread
    struct args *args = p;
    struct res *r = malloc(sizeof(struct res));
    // Usar parámetros como args->i, args->c
    r->res1=...; r->res2=...;
    return r;
}
```

Threads: Creación

Ejemplo: Creación de Thread en `thread_function`

```
int main() {
    pthread_t thr;
    struct res *r;
    struct args *args = malloc(sizeof(struct args));
    args->i=...; args->c = ...;

    pthread_create(&thr, NULL, thread_function, args);
    pthread_join(thr, &r); // Esperar por thr
    // Valores devueltos en r->res1 y r->res2
    free(args);
}
```

Procesos/Threads: Comunicación

- Existen distintas técnicas para comunicar threads y procesos, como:
  - Ficheros.
  - Memoria Compartida.
  - Paso de Mensajes.
  - Pipes.
  - Colas.
- En la primera parte del curso vamos a centrarnos en memoria compartida.

Procesos/Threads

Threads

```
int i=0;

void *write_i(void *arg) {
  i=1;
}

int main() {
  pthread_t thr;
  pthread_create(&thr, NULL,
                 write_i, NULL);
  pthread_join(thr, NULL);
  printf("%d\n",i);
}
```

i está en una página de datos compartida por ambos threads (imprime 1).

Procesos

```
int i=0;

int main() {
  int pid;
  if((pid=fork())==0) {
    i=1;
    exit(0);
  } else {
    waitpid(pid, NULL, 0);
    printf("%d\n", i);
  }
}
```

i está en una página copy on write, por lo que el padre no ve el cambio (imprime 0).

Memoria Compartida con Threads

- Todos los threads de un mismo proceso comparten la tabla de páginas, por lo que todos tienen el mismo espacio de direcciones virtual.
- Como solo hay un heap y una zona de datos, todos los threads comparten las variables globales y estáticas.
- Cada thread tiene su propio stack, por lo que las variables locales no se comparten. De todas formas, al estar en el mismo espacio de direcciones virtual, un thread podría acceder a las variables locales de otro si tiene un puntero a ellas.

Memoria compartida con Procesos

- Cada proceso tiene su propia tabla de páginas. Por defecto no se comparte nada.
- Para utilizar zonas de memoria compartida hay que crearlas de forma explícita.

Memoria compartida con Procesos usando MMap

Ejemplo

```
#include <sys/mman.h>

int main() {
  char *shared;
  int size=100;
  if((shared=mmap(NULL, size, PROT_READ | PROT_WRITE,
                  MAP_SHARED | MAP_ANONYMOUS, 0, 0))==NULL) {
    exit(0);
  }
  if(fork()==0) {
    // Hijo. Shared está compartido por ambos procesos.
  } else {
    // Padre.
  }
}
```