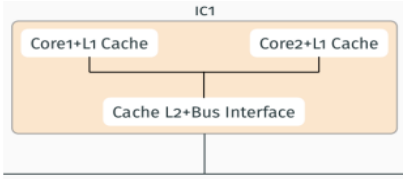
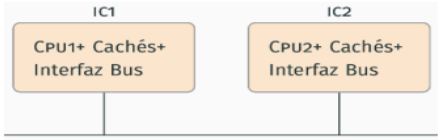
Concurrencia
Un programa concurrente está diseñado para ejecutar varias tareas de forma simultánea. Las tareas pueden ser distintas. Hay 2 problemas principales: Controlar el acceso a recursos compartidos entre tareas y asegurarse que las tareas se pueden comunicar tareas entre sí.

Nos permite emplear mejor el procesador cuando hay mucha op. E/S: mientras un proceso espera, otro puede emplear el procesador. Muy útil para sistemas que la necesiten para funcionar optimamente (interfaces gráficas…)

Arquitecturas Hardware
-Sistema multiprocesador: Más de un procesador en el mismo sistema. Cada procesador está en un circuito integrado, disponiendo de sus propias Mc y comunicándose con otros procesadores mediante buses.

-Sistema multinúcleo: Más de un procesador, pero en el mismo circuito integrado. Generalmente, cada núcleo dispone de cache L1, pero comparten niveles inferiores de cache (L2). Comparten mismo bus para comunicarse.



<- (multiprocesador)


<- (multinucleo)

-Procesador multithread: Puede ejecutar varias tareas tareas concurrentemente. 2 tipos: Multithreading Temporal (núcleo cambia periódicamente la tarea que se ejecuta para ejecutar tareas en segmentos donde habría burbujas) y Simultáneo (permite duplicar partes del núcleo para ejecutar la misma tarea en paralelo).

Procesos
Instancia de la ejecución de un programa, pudiéndose ejecutar más de uno a la vez. Cada uno tiene sus propios recursos (Memoria [tabla de páginas], Descriptores de fichero [tabla con los ficheros que el programa manipula o tiene abiertos], Sockets [interfaces para conectarse con la red u otros dispositivos]). Al crear un proceso se copian los recursos del padre. Estructura del espacio de Direcciones Virtual **(9).**

Procesos -> Creación -> Uso más común:

```
int pid;
if((pid = fork()) == 0) //Tareas que ejecuta el hijo
 else { if(pid < 0)
         perror("fork() ha fallado");
       else { //Tareas que ejecuta el proceso padre
}}
```

Un proceso padre puede crear un proceso hijo -> “Clon” con el espacio de direcciones igual que el del padre. Dicha función devuelve un nº: el hijo devuelve 0 y el padre el PID del hijo creado. En caso de error, fork() devuelve -1 y no se crea el hijo. Se copian todos los recursos del padre porque en Mv se crear una nueva TP idéntica. Las DV de la TP del hijo apuntan a las mismas direcciones físicas que las DV de la TP del padre. Hijo y padre comparten valores de los recursos.

Para ahorrar recursos del sistema, al crear un proceso se emplea el **copy on write** -> Hace que la nueva TP del hijo sea una copia del padre, pero marcada como solo “modo lectura”. Si el hijo necesita modificar alguna variable o recurso que heredara el padre, el SO genera una excepción, por lo que copia en una nueva página de Mf la página que contiene dicha variable y la marca como escritura para la modificación. Al realizar una copia, las variables tendrñan el mismo valor inicial que el padre.

Hilos (Threads)
Punto de ejecución dentro de un proceso. Los threads dentro de un mismo proceso comparten**:** Memoria (Misma TP, con lo que ven la misma memoria, pero cada thread tiene un stack propio), Descriptores de fichero, Sockets, Señales (podemos controlar cada thread independientemente con ellas). Espacio de Direcciones Virtual **(17).**

Threads -> Creación *(además necesitaremos compilar el programa empleando la opción -pthread)*

```
#include <pthread.h>
int pthread_create(pthread_t *thread, //Identificador del thread
          const pthread_attr *attr. //Atributos del thread
```

```
        void* (*start_fun) (void*), //Funcion que ejecutará el thread
        void* arg); //Parámetros de la función del thread
```

Para esperar a que finalice un thread y guardar el resultado de su ejecución, empleamos:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, //Identificador del thread en espera
            void ** retval); //Puntero donde almacenar el resultado
```

Threads -> Ejemplo

Definimos la tarea que realizara el thread, definiendo una función void* thread_function(void* p) que recibe un puntero (void *p). *p puede ser un puntero a cualquier tipo de dato y puede devolver cualquier tipo de dato. Dentro de dicha función debemos castear dicho puntero al tipo de dato que necesite manipular la función.

```
#include <pthread.h>
struct args {
   int i;
   char c;
};
struct response {
   int res1;
   float res2;
};
void* thread_function(void *p) {
    struct args *args = p; // Casteo para manipular datos recibidos.
//Reserva espacio en el heap para devolver ahí el resultado y que no desaparezca del stack al finalizar la función:
    struct response r = malloc(sizeof(struct response));
 // Asignar los valores al resultado mediante punteros
    r -> res1 = {valor para asignar a res1}; // También (*r).res1  r -> res2 = {valor para asignar a res2}; // También (*r).res2
    return r;  }
```

Una vez definida la función que ejecutará al thread al crearse, mostraremos como se crea y destruye. También como se almacenaría el resultado que devuelva dicha función.

```
#include <pthread.h>
int main() {
   pthread_t thread;
   struct res *r;
   struct args *args = malloc(sizeof(struct args));
   args -> i = {valor del argumento i}; // También (*args).i
   args -> c = {valor del argumento c}; // También (*args).c
   pthread_create(&thread, NULL, thread_function, args); // Crear thread
   // Esperar a que thread acabe de ejecutarse y guardar su resultado
   pthread_join(thread, &r); // Resultados guardados en *r
   free(args);
   return 0;
}
```

Comunicación entre procesos/threads

Existen distintas técnicas para que los procesos/threads puedan comunicarse info. -> Ficheros, Paso de mensajes, Colas, Memoria compartida, Pipes.

Debemos explicar cómo se comparten los recursos entre los procesos y los threads. Para comenzar, las variables globales son compartidas entre todos los threads. Recordemos que al llamar a fork(), se copia todo el espacio de memorias del proceso padre, por lo que cada proceso hijo tendrá sus propias variables locales, independientes de las variables locales de otros procesos.

Recursos compartidos entre threads

Habíamos dicho que todos los procesos comparten la misma tabla de páginas y que cada proceso tenía su propio stack independiente. Como la variable i es global (no está definida dentro de ninguna función), es visible para todos los threads, porque está en una página virtual a la que todos los nuevos threads tienen acceso. Por ello, imprimirá -> Valor de i: 1

```
int i =0; // Variable global
void *write_i(void * arg) {
 i = 1;
}
```

```
int main() {
 pthread_t hilo;
 pthread_create(&hilo, NULL, write_i, NULL);
 pthread_join(hilo, NULL);
 printf("Valor de i: %d \n", i);
 return 0;
}
```

Recursos compartidos entre procesos

Con fork() se ha reservado espacio para una nueva TP para el hijo, per recordemos que al crearlas se aplica **copy on write**, por lo que inicialmente estará apuntando a las mismas entradas en la TP del proceso padre (por lo que inicialmente sus variables valen lo mismo). Cuando el hijo modifica el valor de i, crea una nueva entrada en su TP, pero la del padre permanece intacta, al ser el padre quien muestra el estado, su variable i no se modificó, asi que el resultado es Valor de i: 0.

```
int i = 0; // Variable global
int main() {
   pid_t pid;
   if ( (pid = fork()) == 0 ) { // Tareas del hijo
      i = 1;
      exit(0);
   } else { // Tareas del padre
      waitpid(pid, NULL, 0);
      printf("Valor de i: %d \n", i);
   }
return 0;
}
```

De este modo debe quedarnos claro que todos los threads de un mismo proceso comparten la misma TP, así que todos los threads tienen el mismo espacio de DV: Dado que solo existe un heap y una zona de datos (data), todos los threads comparten sus variables estáticas (porque permanecen después de finalizar la función) y sus variables globales. Sin embargo, si reservamos variables dentro de un thread empleando malloc(), la dirección de dicha variable se almacena en el heap y solo se liberara hasta que llamameos explícitamente a free(), de modo que esa variable puede compartirse entre distintos threads.

Análogamente con los procesos, debe quedarnos claro que cada proceso tiene su propia TP, asi que por defecto no se comparte ningún recurso (los recursos son independientes). Si queremos emplear zonas de memoria compatida debemos crearlas explícitamente. En realidad, la función malloc() es un "interfaz" para la función mmap, pero con mmap podemos pasar más argumentos y ajustar la memoria comaprtida según las necesidades.

```
#include <sys/mman.h>
int main() {
   char* shared; // Puntero a la memoria reservada
   size_t size = 100;
   pid_t pid;
   if ((shared = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 0, 0)) == NULL) {
      exit(0);
      if (fork() == 0) { /*Tareas del hijo. Variable shared se comparte por ambos procesos*/ }
      else { /*Tareas del padre*/ }
      return 0;
}}
```

La función devuelve un puntero a la memoria que se ha reservado. Los argumentos son:

1.- **void* addr:** dirección de memoria en donde poner las páginas de memoria que creemos.

2.- **size_t length:** nº de páginas que queremos reservar en Mv (debe ser multiplo del tam. Pág, sino se aproximará hacia arriba).

3.- **int prot:** permisos que se asignarán a las páginas: PROT_EXEC (páginas pueden ser ejecutadas), PROT_READ (pueden ser leídas), PROT_WRITE (pueden ser escritas), PROT_NONE (no se puede acceder a las páginas)