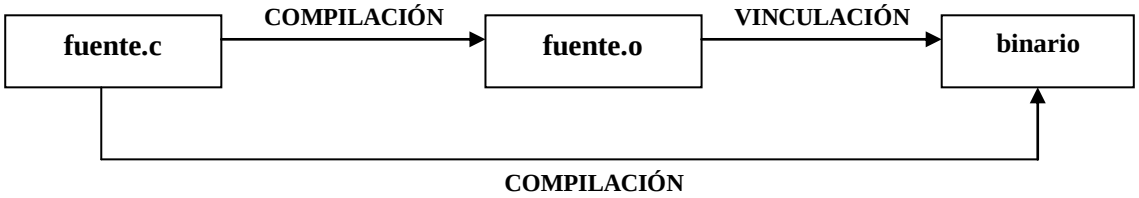
Compilando con gcc/g++

Como todos sabemos el proceso de compilación de un programa, consiste en la traducción del mismo escrito bajo cierto lenguaje de programación a otro lenguaje capaz de ser interpretado por la computadora.

Supongamos que queremos compilar un programa escrito en lenguaje C, llamemos **fuente.c** al archivo que contiene nuestro código fuente. Grafiquemos básicamente el proceso de traducción:



Vamos a realizar la compilación con el comando gcc en el caso de programar con ANSI C o g++ si programamos con C++, tendremos las siguientes variantes:

- **gcc fuente.c**
  - Solo realiza la compilación de fuente.c generando un binario ejecutable llamado a.out
- **gcc -o binario fuente.c**
  - Realiza la compilación de fuente.c generando un binario ejecutable llamado binario
- **gcc -c fuente.c**
  - Realiza la compilación de fuente.c generando un archivo objeto llamado fuente.o
- **gcc -o binario fuente.o**
  - Realiza la vinculación de fuente.o para generar un binario ejecutable llamado binario.

Algunas bibliotecas, ademas de incluirlas en el código fuente, hay que vincularlas en la etapa de vinculación, veremos unos ejemplos puntuales mas adelante.

Compilemos el clásico Hola Mundo !!!:

holamundo.c

```c
#include <stdio.h>

void main(){
   printf("Hola Mundo !!!\n");
}
```

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# gcc -o holamundo holamundo.c
root@sodium:/home/sodium/Desktop/GUIA_TP3# ls -l
total 12
-rwxr-xr-x 1 root   root   7139 2018-10-11 08:36 holamundo
-rw-r--r-- 1 sodium sodium   67 2018-10-11 04:27 holamundo.c
root@sodium:/home/sodium/Desktop/GUIA_TP3# ./holamundo
Hola Mundo !!!
root@sodium:/home/sodium/Desktop/GUIA_TP3#
```

PROCESOS PESADOS

La función que nos permite crear procesos por programa es **fork(),** esta función duplica el proceso que la invoca copiando el código y la memoria, al que llamaremos padre, generando un nuevo proceso al que llamaremos hijo, tiene la particularidad de retornar 2 valores distintos en caso de éxito, uno para el proceso padre y otro para el hijo. Cabe destacar que ambos procesos no comparten memoria, es decir; las variables de un proceso no tienen que ver con las variables del otro, aunque en el código veamos que tienen el mismo nombre, el area de datos es distinta.

Para utilizar fork( ) en nuestros programas debemos incluir la biblioteca **unistd.h**
Retorno de fork( ):

- Entero → en caso de error de creación del nuevo proceso retorna -1 al llamador .
  → en caso de éxito retorna → 0 al nuevo proceso (hijo).
  → positivo (PID del hijo) al llamador (padre).

Veamos un ejemplo:

fork.c

```c
#include <stdio.h>
#include <sys/types.h>  //Biblioteca para utilizar pid_t
#include <unistd.h>     //Biblioteca para utilizar fork
#include <sys/wait.h>   //Biblioteca para utilizar wait

int main( ){
   pid_t pid = fork(); // Equivalente a -> int pid = fork()

   if ( pid == -1){
      printf ( "Error al crear el nuevo proceso !!!" );
      return 1;
   }

   if ( pid == 0){
      printf( "Soy %d, el hijo de %d\n", getpid(), getppid() );
   }
   else{
      printf( "Soy %d, el padre de %d\n", getpid(), pid );
      wait( NULL ); // Espera SIGCHLD desde el hijo
   }

   return 0;
}
```

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# ./fork
Soy 7693, el hijo de 7692
Soy 7692, el padre de 7693
root@sodium:/home/sodium/Desktop/GUIA_TP3#
```

A continuación se detallan las porciones del código que ejecuta cada proceso:

fork.c

PROCESO PADRE

```c
#include <stdio.h>
#include <sys/types.h>  //Biblioteca para utilizar pid_t
#include <unistd.h>     //Biblioteca para utilizar fork
#include <sys/wait.h>   //Biblioteca para utilizar wait

int main( ){
   pid_t pid = fork(); // Equivalente a -> int pid = fork()

   if ( pid == -1){
      printf ( "Error al crear el nuevo proceso !!!" );
      return 1;
   }

   if ( pid == 0){
      printf( "Soy %d, el hijo de %d\n", getpid(), getppid() );
   }
   else{
      printf( "Soy %d, el padre de %d\n", getpid(), pid );
      wait( NULL ); // Espera SIGCHLD desde el hijo
   }

   return 0;
}
```

fork.c

PROCESO HIJO

```c
#include <stdio.h>
#include <sys/types.h>  //Biblioteca para utilizar pid_t
#include <unistd.h>     //Biblioteca para utilizar fork
#include <sys/wait.h>   //Biblioteca para utilizar wait

int main( ){
   pid_t pid = fork(); // Equivalente a -> int pid = fork()

   if ( pid == -1){
      printf ( "Error al crear el nuevo proceso !!!" );
      return 1;
   }

   if ( pid == 0){
      printf( "Soy %d, el hijo de %d\n", getpid(), getppid() );
   }
   else{
      printf( "Soy %d, el padre de %d\n", getpid(), pid );
      wait( NULL ); // Espera SIGCHLD desde el hijo
   }

   return 0;
}
```

pid_t wait(int *status):

Suspende la ejecución del proceso llamador hasta que reciba una señal SIGCHLD proveniente de cualquier proceso hijo, además permite obtener información sobre el estado de finalización del mismo.
Retorna el PID del proceso hijo que ha terminado o –1 en caso de fallo.

pid_t waitpid(pid_t pid, int *status, int options):

Suspende la ejecución del proceso llamador hasta que reciba una señal SIGCHLD proveniente de un proceso hijo en particular especificado por el valor de pid. Si se especifica la opcion WNOHANG, el waitpid deja de ser bloqueante. Si se especifíca -1 como pid, espera a cualquier proceso hijo. El valor de retorno es igual a wait.

En ambas funciones el valor de status permite conocer el estado de finalización del proceso hijo, por ejemplo:

estado.c

```c
#include <stdio.h>
#include <unistd.h>     //Biblioteca para utilizar fork
#include <sys/wait.h>   //Biblioteca para utilizar wait

int main( ){
   int estado;

   switch ( fork() ){
      case -1:
         printf( "Error al crear el nuevo proceso !!!" );
      break;

      case 0:
         printf( "Soy %d, el hijo de %d\n", getpid(), getppid() );
         pause();
         return 0;
      break;

      default:
         printf( "Soy %d, el padre\n", getpid() );
         wait(&estado);
         if( WIFSIGNALED ( estado ) )
            printf( "Hijo finalizado por la señal %d \n", WTERMSIG ( estado ) );
      break;
   };

   return 0;
}
```

Al ejecutar el programa, ambos procesos padre e hijo, quedarán bloqueados, el hijo por la función pause() que bloquea al proceso hasta que reciba alguna señal, y el padre por la función wait, a la espera de finalización del proceso hijo.

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# ./estado
Soy 7842, el hijo de 7841
Soy 7841, el padre
```

Desde otra terminal, enviamos una señal cualquiera al proceso hijo, en este caso la SIGUSR1

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# kill -10 7842
root@sodium:/home/sodium/Desktop/GUIA_TP3#
```

Ahora sí vemos que el proceso hijo finaliza, ocurrido esto se envía automáticamente la señal SIGCHLD desde el hijo hacia el padre, ésta es capturada por la función wait quien almacena en la variable entera estado el estado de finalización del proceso hijo.

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# ./estado
Soy 7842, el hijo de 7841
Soy 7841, el padre
Hijo finalizado por la señal 10
root@sodium:/home/sodium/Desktop/GUIA_TP3#
```

Hay una serie de macros definidas por la biblioteca wait.h que trabajan con los datos almacenados en la variable estado. Por ejemplo en este caso, se pregunta con WIFSIGNALED, si el proceso hijo fue finalizado con una señal, de ser así, con WTERMSIG se puede acceder al número de señal en cuestión.

wait(NULL) es equivalente a wait(-1, NULL, 0)

Existen otras funciones del tipo wait que permiten obtener estadísticas del proceso hijo finalizado, haciendo uso de la estructura **rusage,** donde se almacena por ejemplo la cantidad de fallos de página, cambios de contexto y señales recibidas entre otras estadísticas del proceso hijo.

pid_t wait3(int *status, int options, struct rusage *rusage)
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage)

PROCESOS LIVIANOS

Es similar en ciertos aspectos a un proceso, pero si se pasa a ejecutar un hilo distinto de la misma tarea, el cambio de contexto es menor. En los hilos, los datos no se copian, se comparten y cada hilo accede por tanto a los mismos datos y las modificaciones serán vistas por todos los hilos de la misma tarea.

Para utilizar hilos, es necesario una biblioteca externa <lpthread.h> y compilar con -lpthread

Veamos un ejemplo:

hilos.c

```c
#include <stdio.h>
#include <pthread.h> //Biblioteca para el manejo de hilos

int x=0;
pthread_mutex_t   mtx = PTHREAD_MUTEX_INITIALIZER; //Semáforo de hilos, mtx=1 (Abierto)

void *func(void *nombre){
   pthread_mutex_lock(&mtx);//P(mtx)
   printf("Nombre del thread %s\n", (char *)nombre);
   x++;
   pthread_mutex_unlock(&mtx);//V(mtx)
   pthread_exit(0);
}

int main(){
   pthread_t hilo1; //pthread_t -> unsigned long int
   pthread_t hilo2;

   printf("X inicial: %d\n",x);

   pthread_create(&hilo1, NULL, &func, (void *)"A");
   pthread_create(&hilo2, NULL, &func, (void *)"B");

   pthread_join(hilo1, NULL);
   pthread_join(hilo2, NULL);

   pthread_mutex_destroy(&mtx); //Liberar la memoria utilizada por el semáforo

   printf("X final: %d\n",x);

   return 0;
}
```

Compilar de alguna de las siguientes maneras:

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# gcc -o hilos hilos.c -lrt
root@sodium:/home/sodium/Desktop/GUIA_TP3# gcc -o hilos hilos.c -lpthread
```

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# ./hilos
X inicial: 0
Nombre del thread A
Nombre del thread B
X final: 2
root@sodium:/home/sodium/Desktop/GUIA_TP3#
```

int pthread_create (pthread_t *x, pthread_attr_t *y, void *(*z)(void *), void *w)

x: TID.

y: Estructura propia de pthread donde se puede establecer atributos de los threads, por ejemplo si se crean detachados o joineables, la prioridad, etc. El manejo de dicha estructura se realiza a traves de funciones propias de la biblioteca como por ejemplo;
pthread_attr_init (pthread_attr_t *y),
pthread_attr_destroy (pthread_attr_t *y),
pthread_attr_setdetachstate(pthread_attr_t *y, int detachstate)

z: función que ejecutará el thread.

w: Parámetros para el thread.

Los hilos pueden ser Joineables (dependientes) o detachados (independientes) con respecto al proceso que los creo, por defecto se crean joineables. Un hilo joineable no libera sus recursos al finalizar, el proceso que lo creo debe capturar la terminacion del mismo con la siguiente funcion:

int pthread_join(pthread_t x, void **ret)

x: TID.
ret: Guarda el resultado retornado por el thread. (NULL si no se recibirá retorno desde el thread).

int pthread_exit (void *retval): Finalización del thread y retorno de datos a traves de **retval**.

int pthread_kill (pthread_t x, int numero_de_señal): envío de señales al thread.

pthread_t pthread_self (void): Obtiene el propio TID.

pthread_mutex_trylock(pthread_mutex_t *m): Comprueba el estado del semáforo, si esta libre lo bloquea y retorna 0, si ya esta bloqueado retorna EBUSY, lo cual nos permitira realizar otras tareas si no se puede bloquear el semaforo.

TUBERIAS CON NOMBRE (FIFOs)

fifoA.c

```c
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/stat.h>
#define TAMBUF 1024

int main (){
   char frase[TAMBUF];

   mkfifo("./fifo1",0666);
   mkfifo("./fifo2",0666);
   int w = open("./fifo1", O_WRONLY);
   int r = open("./fifo2", O_RDONLY);

   for(int i=0;i<10;i++){
      frase[0]='a'+i;
      write(w,frase,TAMBUF);
      read(r,frase,TAMBUF);
      printf("%c",frase[0]);
   };

   close(r);
   close(w);

   unlink("./fifo1");
   unlink("./fifo2");
   return 0;
}
```

fifoB.c

```c
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#define TAMBUF 1024

int main (){
   char frase[TAMBUF];

   int r = open("./fifo1", O_RDONLY);
   int w = open("./fifo2", O_WRONLY);

   for(int i=0;i<10;i++){
      read(r,frase,TAMBUF);
      printf("%c",frase[0]);
      frase[0]='z'-i;
      write(w,frase,TAMBUF);
   };

   close(r);
   close(w);

   return 0;
}
```

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# g++ -o A fifoA.c
root@sodium:/home/sodium/Desktop/GUIA_TP3# g++ -o B fifoB.c
root@sodium:/home/sodium/Desktop/GUIA_TP3#
```

Al ejecutar el proceso A, el mismo se bloquea en la función read de la tubería **fifo2**, hasta que se escriba algo en dicha tubería por parte del proceso B.

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# ./A
```

Al ejecutar el proceso B, el mismo escribe en la tubería hacia el proceso A, entonces ahora sí se comienza con la comunicación.

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# ./B
abcdefghijroot@sodium:/home/sodium/Desktop/GUIA_TP3#
```

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# ./A
zyxwvutsrqroot@sodium:/home/sodium/Desktop/GUIA_TP3#
```

Mientras no se eliminen las tuberías, si ejecutamos un ls – l, podremos visualizar las tuberías como se muestra a continuación:

```
prw-r--r-- 1 root   root       0 2018-10-11 12:22 fifo1
prw-r--r-- 1 root   root       0 2018-10-11 12:22 fifo2
```

MEMORIA COMPARTIDA (SYSTEM-V) - SEMÁFOROS (POSIX) NOMBRADOS

En el siguiente ejemplo se aplica memoria compartida del estándar SYSTEM-V como medio de comunicación entre dos procesos no emparentados. Tambien se utilizan semáforos del estándar POSIX para dar un orden en la ejecución de ambos procesos.

mcB.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/ipc.h> //Biblioteca para los flags IPC_
#include <sys/shm.h> //Para utilizar memoria compartida
#include <fcntl.h>   //Para utilizar los flags O_
#include <semaphore.h> //Para utilizar semáforos POSIX

int main(){

   sem_t *leer=sem_open("/leer",O_CREAT,0666,0);
   sem_t *leido=sem_open("/leido",O_CREAT,0666,0);

   int shmid = shmget(234, sizeof(int), IPC_CREAT | 0666);

   int *cien = (int *)shmat(shmid,NULL,0);

   sem_wait(leer);
   printf("%d\n",*cien);
   sem_post(leido);

   shmdt( &cien );

   sem_close(leer);
   sem_close(leido);

   return 0;
}
```

mcA.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/ipc.h> //Biblioteca para los flags IPC_
#include <sys/shm.h> //Para memoria compartida SYSTEM-V
#include <semaphore.h> //Para semáforos POSIX
#include <fcntl.h>   //Para utilizar los flags O_

int main(){
   //Creación de semáforo
   sem_t *leer=sem_open("/leer",O_CREAT,0666,0);
   sem_t *leido=sem_open("/leido",O_CREAT,0666,0);

   //Creación de memoria compartida
   int shmid = shmget(234, sizeof(int), IPC_CREAT | 0666);
   //Vincular la memoria compartida a una variable local
   int *cien = (int *)shmat(shmid,NULL,0);

   *cien = 100;

   sem_post(leer);
   sem_wait(leido);

   //Desvincular la memoria compartida de la variable local
   shmdt( &cien );
   //Marcar la memoria compartida para borrar
   shmctl(shmid,IPC_RMID,NULL);

   //Cierre de semáforo
   sem_close(leer);
   sem_close(leido);
   //Marcar el semáforo para destruirlo
   sem_unlink("/leer");
   sem_unlink("/leido");

   return 0;
```

Compilamos ambos programas con la opción –lrt, necesarios para las funciones de semáforos.

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# g++ -o A mcA.c -lrt
root@sodium:/home/sodium/Desktop/GUIA_TP3# g++ -o B mcB.c -lrt
root@sodium:/home/sodium/Desktop/GUIA_TP3#
```

Mientras no se eliminen, la memoria o los semáforos, podemos visualizarlos. Para la memoria compartida es con el comando ipcs

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# ipcs -m

---- Segmentos memoria compartida ----
key        shmid      propietario perms      bytes      nattch     estado

0x00000000 0          sodium     600        393216     2          dest

0x00000000 32769      sodium     600        393216     2          dest

0x00000000 65538      sodium     600        393216     2          dest

0x000000ea 2031637    root       666        4          1
```

Para los semáforos es visualizar el directorio /dev/shm

```
root@sodium:/home/sodium/Desktop/GUIA_TP3# ls /dev/shm/
pulse-shm-1438137444  pulse-shm-3201691442  sem.leer
pulse-shm-1730393502  pulse-shm-3748379577  sem.leido
pulse-shm-1949816569  pulse-shm-3810107686
root@sodium:/home/sodium/Desktop/GUIA_TP3#
```