Universidad de Cantabria
Grado de Ingeniería Informática

Sistemas Operativos

Práctica 2 - Parte 1

Rafael Menéndez de Llano Rozas

Guion

2

Creación de procesos

En esta parte adquirirás la visión del sistema como programador haciendo uso de las llamadas al sistema POSIX para la creación de procesos y trabajar con la información que el SO tiene de ellos, así como su sincronización en la creación.

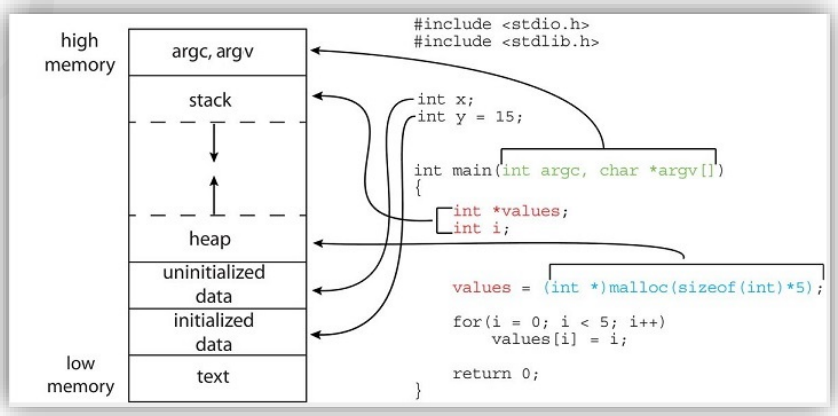
Procesos

Un programa es un conjunto de código y datos que se encuentran almacenados en un fichero del sistema operativo. Para que ese fichero pueda ser ejecutable tiene que tener en su *inodo* (estructura de datos interna que contiene las características de un fichero) un campo (número mágico) que así lo indique.

Hay varios formatos en los que puede venir el ejecutable:

- Assembler OUTput (`a.out`) es el clásico (secciones fijas).
- COFF formato de BSD y SVr3 (dos versiones de UNIX).
- ELF (Executable Link Format) formato de Linux moderno.

El formato de tu sistema lo puedes comprobar con el comando `objdump`. Si es ELF puedes leerlo con el comando `readelf` (no te dejes engañar por el nombre, un fichero `a.out` puede ser ELF).



Un programa cargado en memoria junto con la información del S.O. se convierte en un **proceso**. Los procesos tienen fundamentalmente tres bloques de memoria que aparecen en el ELF: un segmento de código (`.text`) con instrucciones máquina propias de la CPU que lo ejecute, un segmento de datos (`.data`) con los valores para las variables que se tienen que inicializar (en

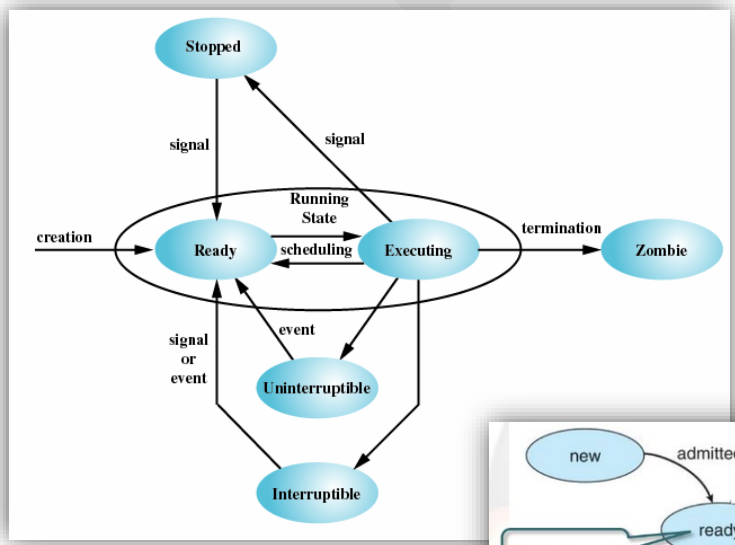
un programa en C estarían las variables globales y estáticas inicializadas), un segmento `.bss` (con datos globales no inicializados). Además, pueden tener código de inicio o terminación (`.init` `.fini`), datos de sólo lectura (`.rodata`), uno o varios segmentos de memoria compartida, segmentos para librerías específicas y por último un par de segmentos para la pila (`stack`) y el montón (`heap`) que se crean en tiempo de ejecución (no están en ELF) y donde son almacenados, por una parte las variables locales, los argumentos de las funciones y lo que devuelven, y por otra parte, memoria creada en ejecución (con operaciones de tipo `malloc/calloc`). Debido a que el proceso puede ser ejecutado en modo usuario o en modo supervisor, realmente existen dos segmentos de *stack*.

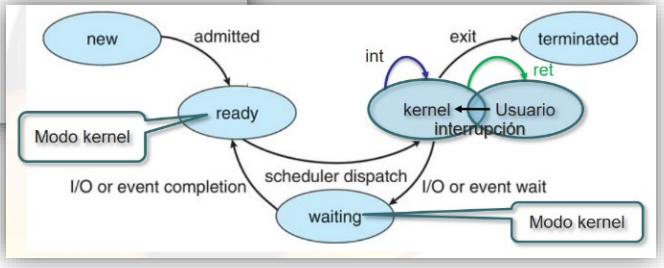
1) Utiliza el comando `readelf` con la opción `-h` y `-S` aplicado a un ejecutable que contenga un *array* global, una variable inicializada, y variables locales a `main`. Identifica al menos tres secciones que se han visto en teoría. ¿Si comentas las variables cambian los tamaños de los segmentos? Usa el comando `objdump` con la opción `-a` y `-h` de la misma manera. Los segmentos de *stack o heap* ¿aparecen en las secciones de ELF? Por último, prueba el comando `size`.

Además, por cada proceso se crea una estructura de datos para que el sistema operativo lo maneje, que es el bloque de control del proceso (PCB).

En un sistema multiproceso (con una CPU), sólo un proceso puede estar ejecutándose en un momento dado, por lo cual, los procesos pasan por una serie de estados, fundamentalmente tres: ejecutándose, listos para ser ejecutados o bloqueados esperando un recurso. Es el *kernel*, a través de sus políticas de planificación, el que escoge qué proceso se debe ejecutar y por cuánto tiempo. Cada vez que se cambia de proceso, se dice que se ha producido un *cambio de contexto*.

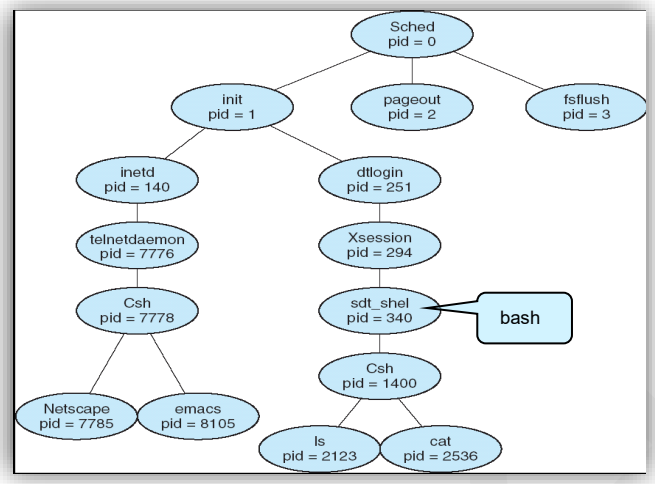
Como has visto en teoría, en el caso del Linux los estados tienen una forma más compleja, distinguiendo sobre todo el estado bloqueado que se disocia en tres: parado, interrumpible y no interrumpible. En el caso del UNIX se distingue entre los modos de ejecución (usuario o *kernel*) y un estado nuevo que es expulsado equivalente a "preparado". Las transiciones se reflejan en las siguientes figuras.





Todos los procesos del sistema tienen una identificación que en el caso del Linux es un número entero (`pid_t`) llamado PID (de dos bytes sin signo). Además, los procesos tienen una jerarquía determinada (todos tienen un proceso padre), el único que no tiene padre es el proceso inicial con PID=1. En Linux antiguos, Unix, o en la bash de Ubuntu de W10 a ese proceso se le llama `init`, en Linux modernos se le llama `systemd`. Puedes ver los procesos que se están ejecutando con el comando `ps` al que añadirás

la opción `-e` (para ver todos) y la `-f` (o –F) para el formato full. En la siguiente figura puedes ver la representación de la jerarquía de procesos:



2) Ejecuta el comando `pstree` y observa tu jerarquía de procesos. Indica dónde aparece el propio `pstree` y de quién cuelga. Retrocede hasta llegar al proceso inicial del sistema y escribe por todos los que se pasan para llegar a que ejecutes algún comando. Ahora entra desde un terminal virtual de texto y haz lo mismo (o si tienes la bash de W10) ¿cuál es la gran diferencia?

El sistema operativo crea un directorio virtual con la información dinámica (los PCB) que maneja y que ofrece al usuario a través de los comandos. Por ejemplo, `ps, pstree, vmstat, uname` o `top` hacen uso de este sistema de ficheros virtual. El aspecto es parecido a éste que se obtiene con `ls /proc` :

```
rafa@ubuntu:~$ ls /proc
1/      13/     195/    211/  230/    24622/  32/     47918/  49459/  49665/  49837/  50022/  846/      cpuinfo      kpagecount   stat
10/     1305/   196/    212/  231/    24623/  33/     48/     49525/  49667/  49849/  50024/  849/      crypto       kpageflags   swaps
100/    135/    19629/  213/  232/    247/    339/    48414/  49526/  49675/  49855/  50037/  851/      devices      loadavg      sys/
101/    14/     197/    214/  233/    248/    34/     48456/  49529/  49683/  49856/  50044/  855/      diskstats    locks        sysrq-trigger
102/    14304/  198/    215/  234/    25/     36/     48581/  49537/  49690/  49871/  50051/  861/      dma          mdstat       sysvipc/
103/    15/     199/    216/  235/    26/     37/     49/     49549/  49708/  49872/  50053/  889/      driver/      meminfo      thread-self@
109/    15782/  2/      217/  236/    27/     38/     49029/  49572/  49709/  49877/  50077/  9/        execdomains  misc         timer_list
11/     16/     20/     218/  237/    275/    39/     49034/  49577/  49710/  49878/  50081/  904/      fb           modules      tty/
118/    1745/   200/    219/  238/    276/    4/      49035/  49597/  49711/  49883/  50112/  939/      filesystems  mounts@      uptime
12/     1756/   201/    22/   239/    278/    40/     49051/  49600/  49714/  49886/  50117/  956/      fs/          mpt/         version
12115/  18/     202/    220/  24/      279/    41/     49059/  49612/  49716/  49906/  55/     98/       interrupts   mtrr         version_signature
12116/  187/    203/    221/  240/     28/     413/    49201/  49615/  49719/  49916/  56/     99/       iomem        net@         vmallocinfo
12123/  188/    204/    222/  241/     280/    416/    49208/  49616/  49722/  49930/  6/      995/      ioports      pagetypeinfo vmstat
12124/  189/    205/    223/  242/     283/    42/     49437/  49617/  49735/  49934/  6118/   acpi/     irq/         partitions   zoneinfo
12125/  19/     206/    224/  243/     294/    43/     49444/  49618/  49751/  49940/  7/      asound/   kallsyms     sched_debug
12126/  190/    207/    225/  24368/   298/    44/     49448/  49623/  49784/  49961/  8/      buddyinfo kcore        schedstat
12127/  191/    208/    226/  244/     30/     45/     49449/  49640/  49812/  49998/  819/    bus/      keys         scsi/
12128/  192/    209/    227/  24467/   308/    46/     49451/  49648/  49816/  50004/  829/    cgroups   key-users    self@
1278/   193/    21/     228/  245/     309/    47/     49452/  49658/  49819/  50016/  831/    cmdline   kmsg         slabinfo
1287/   194/    210/    229/  246/     31/     47509/  49457/  49660/  49836/  50019/  835/    consoles  kpagecgroup  softirqs
```

En el caso de la `bash` de W10, se puede reducir y simplificar bastante:

```
rafa@jargo:~$ ls /proc
1/   3/  bus/     cmdline  filesystems  loadavg  mounts@  self@  sys/  uptime   version_signature
20/  4/  cgroups  cpuinfo  interrupts   meminfo  net@     stat   tty/  version
```

Cada número que aparece es un directorio que corresponde al PID de un proceso en ejecución, dentro de ese directorio habrá información que sólo pertenece a ese proceso. Por ejemplo, para el proceso `bash`:

```
attr        comm             fd         map_files   net            pagemap      sessionid  status
autogroup   coredump_filter  fdinfo     maps        ns             personality  setgroups  syscall
auxv        cpuset           gid_map    mem         numa_maps      projid_map   smaps      task
cgroup      cwd              io         mountinfo   oom_adj        root         stack      timers
clear_refs  environ          limits     mounts      oom_score      sched        stat       uid_map
cmdline     exe              loginuid   mountstats  oom_score_adj  schedstat    statm      wchan
```

En el caso de la `bash` de Windows (proceso 3):

```
attr  cmdline  cwd  fd      maps       mounts      net      root       smaps  statm   task
auxv  comm     exe  limits  mountinfo  mountstats  oom_adj  schedstat  stat   status
```

3) Vete al directorio `/proc` y lista su contenido. Identifica el PID de la Shell, vete a su directorio y lista su contenido. Usa `cat` o `ls` para averiguar qué contienen los ficheros que aparecen. Pon en pantalla el fichero `maps` y después identifica los segmentos de memoria que tiene ese proceso volcados. Localiza (debes saber ya el comando gracias al guion 1) en qué fichero está el parámetro `vruntime`, y recuerda para qué se usa. Indica el número de cambios de contexto del proceso y su tipo, así como el número máximo de procesos y el número máximo de PID.

Creación de procesos

Para el programador, POSIX suministra un conjunto de llamadas para la creación de procesos, entre las que pueden destacarse:

- **fork** . Usada para crear un nuevo proceso mediante la duplicación del proceso llamador. `fork` es la primitiva básica de creación de un proceso.
- **execXX** . Una familia de llamadas al sistema, cada una de las cuales realiza la misma función: la transformación de un proceso mediante el recubrimiento de su espacio de memoria con un nuevo programa.
- **wait** . Permite que un proceso padre espere a que su hijo termine. Esta llamada es un método rudimentario de sincronización entre procesos.
- **exit** Se utiliza para terminar un proceso, incluyendo cerrar sus ficheros. También hace rearrancar a un proceso esperando en un `wait` por él.

A continuación, se van a señalar las características más destacadas de estas llamadas al sistema, así como su forma básica de uso.

Creación de procesos: *fork*

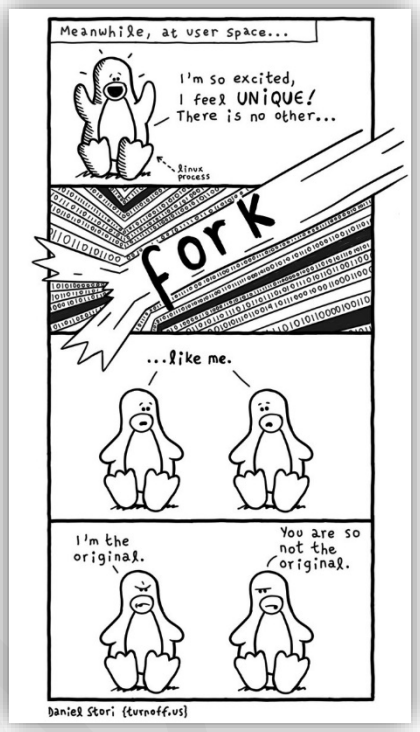
La primitiva básica de creación de procesos es la llamada al sistema `fork`. Su uso básico es el siguiente (las cabeceras necesarias para una llamada las podemos averiguar con `man`):

```
#include <unistd.h>
pid_t fork(void);
```

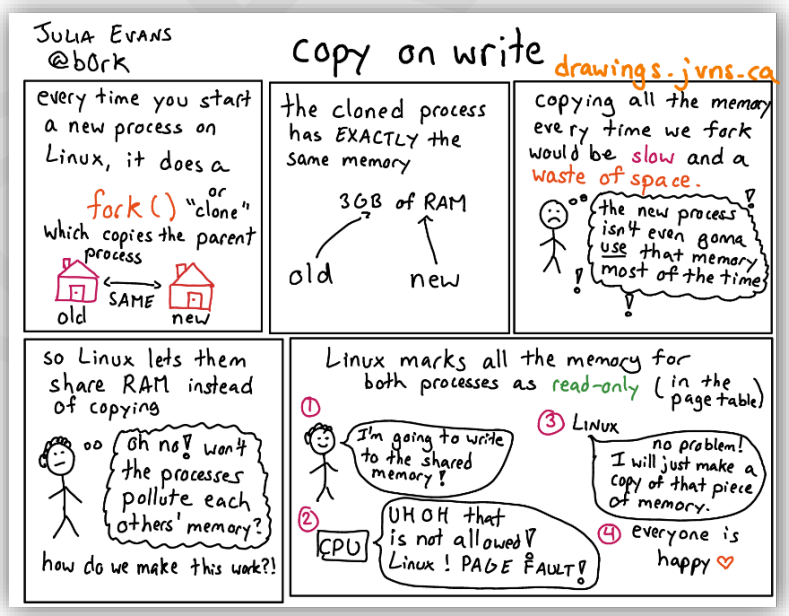
El tipo `pid_t` es encontrado en el fichero de cabecera (librería) `<unistd.h>`, que realmente lo que define es un entero positivo de dos bytes (se hace con `typedef`).

Una llamada con éxito a `fork` causa que el *kernel* cree un nuevo proceso que es una réplica exacta del proceso llamador, incluyendo sus bloques de memoria (al proceso creador se le denomina proceso *padre* y al nuevo proceso se le denomina *hijo)* en los siguientes pasos:

1. Reservar PCB para el nuevo proceso:
   - Comprobar que el número de procesos del usuario no excede del máximo permitido a él o al sistema.
   - "Buscar entrada libre en la tabla" de procesos y un PID único.
   - Marcar el estado del hijo como siendo creado.
2. Crear contexto del nuevo proceso:
   - Compartir los datos del padre al PCB del hijo.
   - Copiar e inicializar los campos que difieran: tiempo, PID, ...
   - Duplicar las zonas del padre.
3. Modificar sistema de ficheros:
   - Copiar la tabla de descriptores del padre sobre el hijo.
   - Incrementar contadores de la tabla de ficheros abiertos.
4. Devolver control:
   - Devolver PID del hijo al proceso padre y cero al hijo como resultado de la función.
   - Colocar a los dos procesos en estado preparado.
   - Llamar al Scheduler.



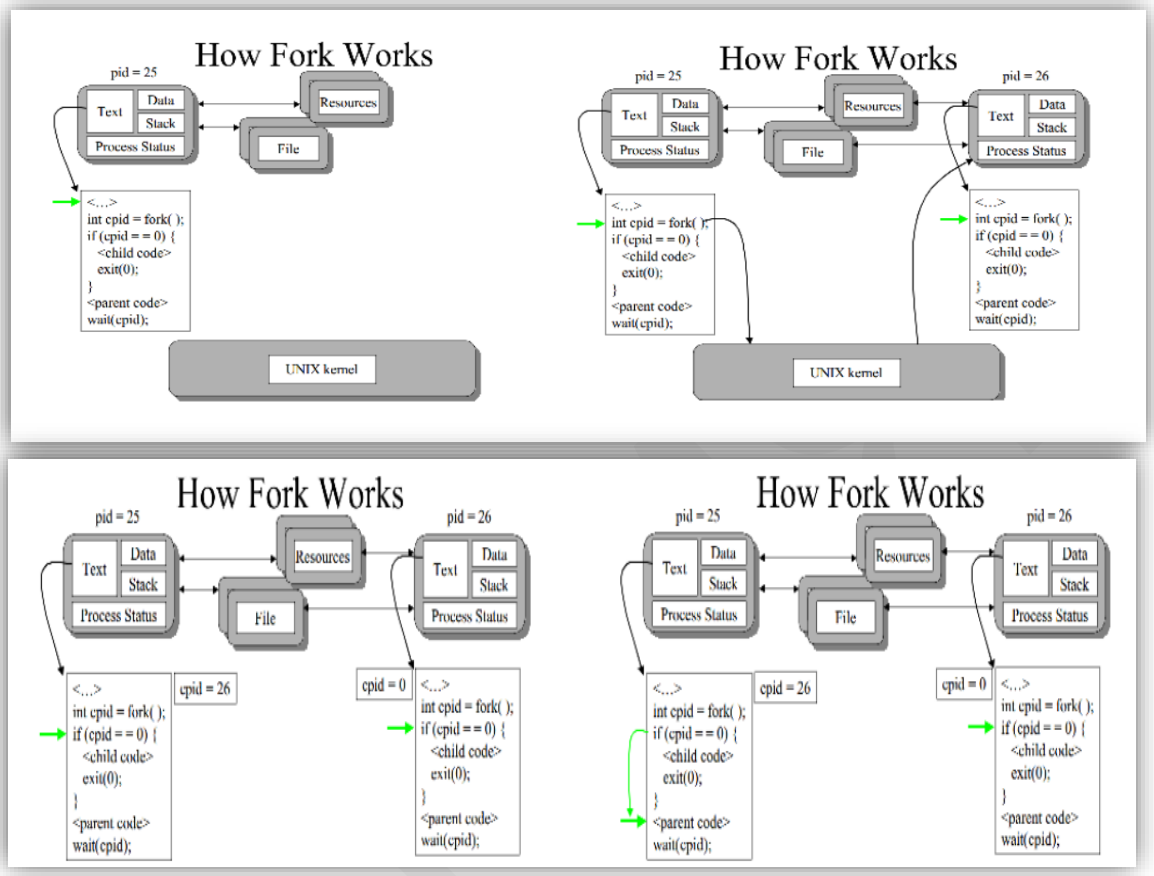
Los dos procesos serán distintos: distintos PID, cada uno con sus propios datos, cada uno con su propia pila, cada uno con sus propios recursos (salvo que se aplique la técnica COW vista en teoría), cada uno tiene un PC distinto. Como en la mayoría de los casos después de llamar a `fork` se llama a `exec` para realizar algo nuevo, en la creación no se duplica nada, y simplemente se comparten la mayoría de datos entre padre e hijo usando la técnica COW (*copy on write* - muy rápido): el código si es reentrante se puede compartir, los datos sólo se copian si se modifican.



La llamada a `fork` se usa de forma sencilla, basta ver el código de estas tres líneas:

```
puts("Antes");
creado = fork();
puts("Despues");
```

aparecerá en pantalla primero un mensaje "*Antes*" correspondiente a la ejecución del proceso padre antes de la llamada, y dos mensajes "*Despues*", uno correspondiente a la ejecución del padre y otro a la del hijo. Esto significa que el proceso creado empieza a ejecutarse en el mismo lugar que el padre, es decir, ambos después de la llamada a `fork`.



La llamada a `fork` (simplificación de `clone` con la que se crean procesos e hilos y que puede hacer que el contexto del hijo sea compartido) se realiza sin argumentos y devuelve un entero con el PID del proceso creado (en caso de error el valor devuelto será "-1"). En el padre, esa variable se pone a un valor entero positivo distinto de cero. Y en el hijo se pone a cero (no ha creado nada), esto permite al programador especificar diferentes acciones para cada proceso.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
        pid_t creado;
        puts("Solo un proceso en marcha");
        puts("voy a llamar a fork...");
        creado = fork();     /* crea un nuevo proceso */
        if (creado == 0)
              puts("Yo soy el hijo");
        else if (creado > 0)
               printf("Yo soy el padre,el hijo tiene el pid %d\n",creado);
        else /* valor igual a -1 */
        {
              perror("fork ha devuelto error, no hay hijo");
              exit(1);
        }
}
```

4) Obtén este código (`ejem_fork.c`) de la página, entiéndelo y pruébalo. Date cuenta cómo aparecen dos mensajes de los dos procesos e indica por qué canal saldría el mensaje de error. Si desvías la salida a un fichero ¿qué ocurre?

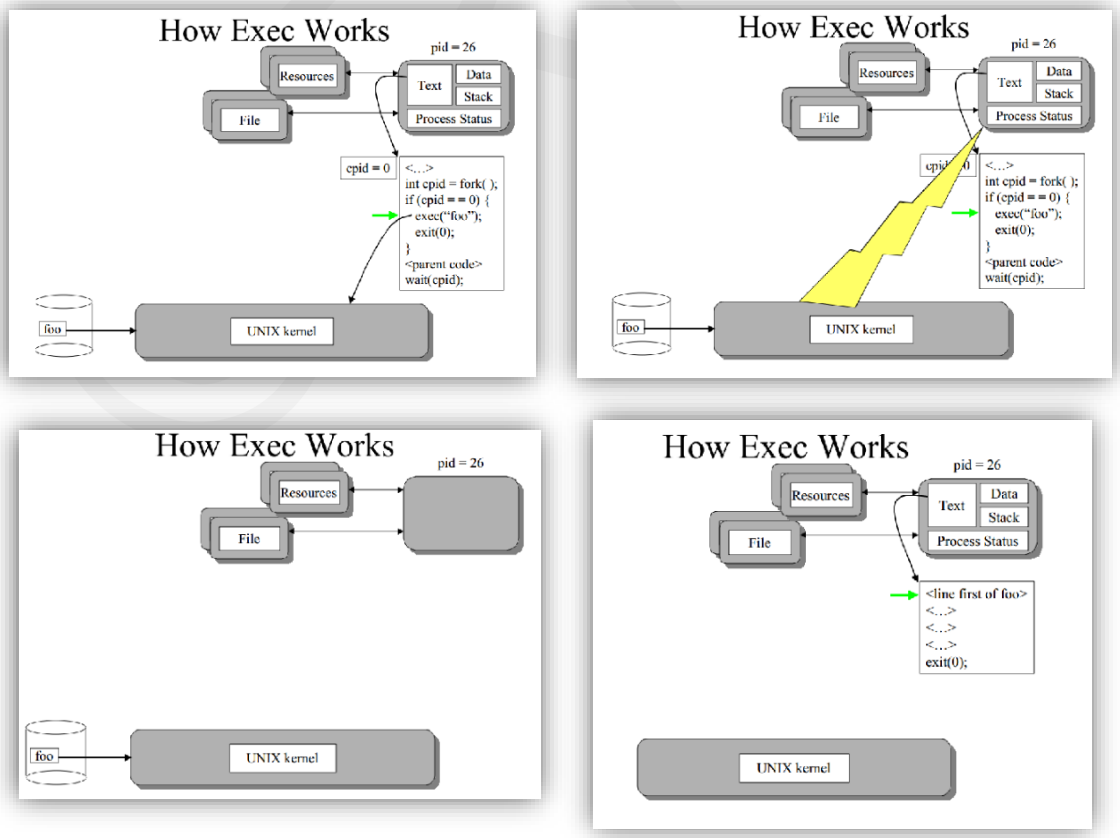
5) Realiza un programa en que un proceso padre genera N procesos hijo iguales (hermanos entre ellos). Crea otro programa que contenga tres generaciones de procesos: Abuelo padre y nieto. De forma opcional haz una que tenga N generaciones.

Ejecutando nuevos programas: *exec*

El valor esencial de `fork` realmente aparece cuando se combina con las otras llamadas al sistema, en particular con la familia `exec`. Las llamadas al sistema `exec` permiten al programador iniciar la ejecución de un nuevo programa sobre uno existente.

Los pasos que provoca esta llamada son:

1. Conseguir **imagen** (formato ELF) del programa a ejecutar:
   - Conseguir i-node (identificación entera) del fichero ejecutable.
   - Verificar número mágico (un código que se coloca al comienzo de todos los ficheros ejecutables) y permiso de ejecución.
   - Lectura de cabeceras. Comprobación de que es “cargable”.



2. Modificar **contexto** del proceso:
   - Salvar temporalmente los parámetros de `exec()`.
   - Liberar las regiones de memoria del proceso.
   - Reservar espacio en memoria para las nuevas regiones.
   - Cargar los contenidos del fichero ejecutable.
   - Modificación de algunos campos del PCB: Contador de programa, puntero de pila, memoria ...

3. **Devolver** control:
   - Cargar variables de entorno y parámetros de la llamada `exec()` a la pila del proceso.
   - Poner al proceso en estado preparado.
   - Llamar al planificador (s*cheduler)*.

El formato de los principales miembros de la familia `exec` es el siguiente:

```
char *camino, *fichero;
char *arg0, *arg1, ..., *argn;
char *argv[];
int ret;
ret = execl(camino, arg0,arg1,.., argn, (char *)0);
ret = execv(camino, argv);
ret = execlp(fichero, arg0, arg1,.., argn, (char *)0);
ret = execvp(fichero, argv);
```

Todas las variedades de `exec` (familia) realizan la misma función: transforman al proceso llamador, cargando un nuevo programa en su espacio de memoria. De hecho, son formas (*front-end*) de hacer una única llamada a `execve` que es la verdadera llamada al sistema. El resultado es un nuevo proceso que tiene el mismo PID que el llamador, es decir `exec` no crea un nuevo proceso para correr concurrentemente con el llamador, sino que éste es sustituido. Por lo tanto, el proceso llamador no sigue ejecutando el código antiguo si la llamada funciona bien.

El primer argumento de la llamada (todos son *strings*), *camino*, debe suministrar una ruta a un fichero ejecutable (el sistema lo sabe a través de la lectura de los primeros bytes del fichero: número mágico). El segundo argumento, *arg0*, es el nombre del fichero. El resto de los argumentos dependerá de los datos que necesite el programa (al ser un número variable, el último argumento será un puntero a cero como se ve en el *type cast*), por ejemplo, si ejecutáramos el comando de la *shell* `ls` podríamos darle como argumentos las opciones del mismo como `-l` y el nombre de algún directorio: `ls -l dir`

Esto se puede observar en el código que usa la llamada en formato de lista (las llamadas que terminan en p son versiones más modernas de las anteriores ya que usan la variable PATH).

```
#include <stdlib.h>
#include <unistd.h>
int main()
{
        puts("ejecutando ls");
        execlp("/bin/ls", "ls", "-l", (char *)0);
        /* si se ejecuta la siguiente orden la llamada ha fallado */
        perror("execl fallo al intentar correr ls");
        /* se pone una condición de error por ejemplo 1 */
        exit(1);
}
```

Otra forma de hacer algo equivalente, pero dando los argumentos en forma de vector es `execv` (al igual que su versión moderna `execvp`):

```
#include <stdlib.h>
#include <unistd.h>
int main()
{
        char *av[3];
        av[0] = "ls";
        av[1] = "-l";
        av[2] = (char *)0;
        execvp("/bin/ls", av);
        /* si llega aquí hay error */
        perror("fallo de execv");
        exit(1);
}
```

Como hemos dicho, `execlp` y `execvp` acabadas en p son versiones modernas de las anteriores (sin p), el primer argumento apunta a un fichero y no a un pathname, ya que usan la variable de la *shell* PATH para encontrar el ejecutable, además de permitir ejecutar macros, no sólo ejecutables.

6) Descárgate los programas (`ejem_exec1.c y 2`) de la página, entiéndelos, compílalos y ejecútalos para que compruebes cómo desde tu propio programa se ejecuta un comando del sistema. Modifícalo para ejecutar otro programa tuyo anterior o del sistema (comando).

El cometido de las llamadas `exec` es reproducir la línea de comando para ejecutar algo como si se hubiera ejecutado desde la *shell*, pero conviene saber que desde un programa en C podemos acceder a estos argumentos que nos pasan a través de los argumentos de *main*. Incluso podemos acceder al entorno de ejecución (variables de entorno de la *shell*). Hay tres maneras (en POSIX escoger la primera):

1. Con la llamada `getenv (char *)`.
2. Con la variable externa `environ`.
3. Con el tercer argumento del *main* en formato de doble puntero.

En el siguiente ejemplo se describe cómo se puede usar:

```
/* se le debe pasar el nombre de alguna variable de shell */
#include <stdio.h>
#include <stdlib.h>
/* main con tres argumentos: numero de argumentos desde la linea de
comando, sus valores en formato string, el entorno de shell */
int main(int argc, char *argv[], char **entorno)
{
    int i;
    char *valor;

    for (i = 0; i < argc; i++)   // se pintan los argumentos
        puts(argv[i]);
    valor = getenv(argv[1]);    // lo devuelto es puntero a char
    if (valor)             // si valor es NULL no puede escribirse
        printf("El valor de %s es %s \n",argv[1],valor);
    while(*entorno != (char *)0)
        puts(*entorno++);      // se pinta el entorno (env)
}
```

7) Descárgate el programa de la página (`param.c`), entiéndelo, compílalo y ejecútalo con un argumento (variable) que pertenezca al entorno del programa. Si no conoces ninguno prueba con cualquiera (`printenv`) y fíjate lo que sale.

Sincronizando procesos con *wait*

Cuando un proceso termina o quiere terminar, llama a la función `exit` (aunque no lo haga el sistema lo hace por él con un argumento cero). Esta función (llamada al sistema) es de tipo `void` y toma como argumento un entero que es precisamente un código de salida de 0 a 255, en el caso de que no haya habido error, este código será cero.

```
#include <stdlib.h>
void exit(int status);
```

El código de salida de un proceso hijo puede ser obtenido por el padre a través de la llamada al sistema `wait`, además, esta llamada suspende la ejecución del padre. Cuando el hijo haya acabado su ejecución, el padre es activado. Si el padre ha creado más de un hijo, `wait` se despierta cuando acabe el primero y retorna su valor `exit` (existe una función: `waitpid`, que espera a un proceso determinado).

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Como la función debe devolver dos datos (identificador y código estatus de salida) el identificador lo devuelve de forma natural y el estatus es pasado por referencia para ser cambiado. El uso del estatus no es inmediato, ya que el retorno de `exit` del hijo estará en el byte más significativo de un entero de dos bytes. En el menos significativo tendremos el código de error producido en caso de que no haya sido posible llamar a `exit`. Para obtener ambos bytes hay que utilizar las facilidades que el C nos da para trabajar con bits.

Primero deberemos detectar que en el byte menos significativo no aparecen unos (esto indicaría que el hijo está parado o abortado), para ello se puede hacer la operación AND (**&**) con el hexadecimal $FF. Comprobado esto, debemos desplazar este entero hasta quedarnos con el byte más significativo con el operador de desplazamiento de bits "**>>**". Se puede ver en este fragmento de código C:

```
…
ret = wait(&status);
/* cuando se despierta */

printf("Los valores son: %d  %d\n", ret, status);
/* el byte menos significativo tiene que ser 0 */

if ((status & 0xFF) != 0)
       puts("Algo falla");

/* ahora tomo el byte mas significativo, desplazo 8 bits*/
status = status >> 8;
printf("El hijo salio con %d\n", status);
```

La salida podría ser:

```
Los valores son: 3687  768
El hijo salio con  3
```

Advertid que 768 en binario es: 00000011 0000 0000 y que 00000011 es 3.

Como esto es engorroso, existen una serie de macros que hacen este trabajo por nosotros, esas macros están en la cabecera `wait.h` y su funcionamiento puede comprobarse con `man -S2 wait`. En el siguiente ejemplo tienes las dos más importantes `WEXITSTATUS` y `WIFEXITED` (puedes ver el resto en man wait):

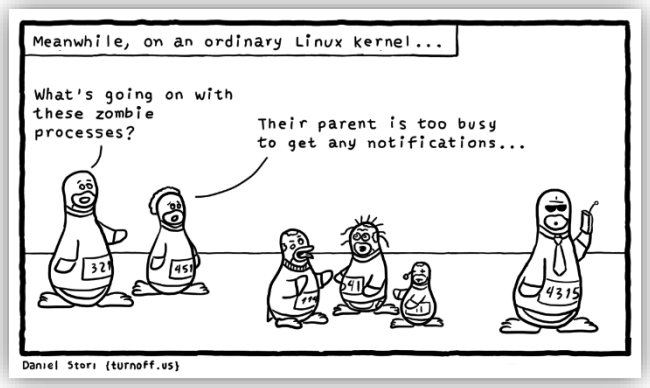
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>   /* aqui estan las macros */
int main()
{
    int codigo,pid;   /* pis puede ser también pid_t */
    puts("Solo un proceso, despues llamo a fork");
    pid = fork();
    if (pid == 0)
    {
         puts("Soy el hijo");
         exit(3);    /* un código de error */
    }
    else if (pid > 0)
    {
         printf("Soy el padre, el hijo es %d \n",pid);

         /* se utilizaran para realizar condiciones de if */
         wait(&codigo);   /* solo un hijo, no recojo el pid */
         printf("El codigo es %d \n",WEXITSTATUS(codigo));
         printf("El codigo de true es %d \n",WIFEXITED(codigo));
    }
    else
         perror("El fork no ha funcionado");
}
```

8) Descárgate el programa de la página (`wait.c`), entiéndelo, compílalo y ejecútalo. Indica para qué sirven las macros del programa. Edita el programa para hacer uso de las macros dentro de sentencias `if`. Usa también de `WIFSIGNALED` y `WTERMSIG` que vienen descritas en el manual.

Zombis

Se ha supuesto hasta aquí que `exit` y `wait` se utilizan de manera ordenada, es decir, que cada subproceso tiene a alguien que le espera. Sin embargo, hay dos situaciones que merece la pena señalar:

1. Un hijo llama a `exit` cuando su padre no está en ese momento esperando un `wait`.
2. Un padre llama a `exit`, mientras uno o varios hijos aún están ejecutándose.



En el primer caso, el proceso es colocado en una especie de limbo y se vuelve un ***zombi*** (aparece al hacer un `ps` como `defunct`). Un proceso *zombi* ocupa una entrada en la tabla

mantenida por el kernel para el control de procesos, pero no usa ningún otro recurso del sistema. Será borrado de esa tabla cuando el padre le reclame mediante la ejecución de `wait`.

En el segundo caso, al padre se le permite salir normalmente y los hijos (incluidos los *zombis*) huérfanos son adoptados por el proceso de inicialización del sistema. Los procesos ***huérfanos zombis*** se eliminan de la tabla de procesos en la adopción.
En el siguiente ejemplo (podría ser una "nano" *shell*) se muestra como pueden ejecutarse dos procesos a la vez, sincronizándose a través de la llamada a `wait` :

```
/* corre ls como un subproceso */
int main()
{
      int creado;

      creado = fork();

      /* si es el padre espera hasta que el hijo acabe */
      if (creado > 0)
      {
            wait((int *)0);  /* no importa lo que devuelve */
            puts("ls completado");
            exit(0);
      }
      /* si es el hijo, ejecuta exec */
      if (creado == 0)
      {
            execl("/bin/ls", "ls", "-l", (char *)0);
            perror("si llega aqui error");
      }
}
```

Cuando la *shell* ejecuta un comando normalmente, usa `fork`, `exec` y `wait` como en el ejemplo anterior. En cambio, cuando el comando se ejecuta en *background*, se omite la llamada `wait` y por tanto ambos procesos (la *shell* y el comando) se ejecutan concurrentemente.

9) Descárgate el programa `minishell.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. Comenta la línea del `wait` y observa qué pasa. Realiza cambios en el programa para que veas la existencia de zombies (*defunct*), recuerda cómo se ejecutan procesos en *background* y que la llamada a `sleep` duerme a un proceso varios segundos que son puestos como argumento.

Compartición de ficheros y datos

Los procesos hijo que se crean tienen "una copia" de las mismas variables declaradas por el proceso padre. Si cualquiera de los dos procesos cambia estas variables, cambiará su copia, pero no la del otro proceso. Esto puede verse con el siguiente ejemplo:

```
#include <stdio.h>
int main()
{
     int creado;
     int comun = 0;   // es realmente comun ?

     puts("Solo un proceso");
     puts("Llamo a fork");
```

```
      creado = fork();
      if (creado == 0)   /* el hijo */
      {
            printf("Soy hijo, comun a 7, ahora vale %d\n",comun);
            comun = 7;
            printf("Soy hijo, comun vale ahora %d \n",comun);
      }
      else if (creado > 0)    /* el padre */
      {
            sleep(1);
            printf("Soy padre, comun a 3, ahora vale %d\n", comun);
            comun = 3;
            printf("Soy padre, comun vale %d \n",comun);
      }
      else
            perror("El fork no ha funcionado");
}
```

Donde ambos procesos tomarán como valor inicial de la variable `comun` el valor de cero. A pesar de que el padre se duerma un segundo (con la llamada a `sleep`), el hijo dejará su variable a 7 y el padre la suya a 3.

10) Descárgate el programa `datos.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. Comprueba el valor de la variable común en ambos procesos.

En la creación de un proceso se comparte la tabla de descriptores (ficheros abiertos) del padre. Si el fichero es abierto antes del `fork` ambos procesos podrán utilizarlo sin necesidad de abrirlo de nuevo, pero en este caso, debido a que el puntero del fichero es almacenado por el sistema, éste se verá compartido por los dos procesos, de tal manera que ambos escribirán en el mismo fichero, pero en distintas posiciones del mismo.

Un ejemplo de esto se puede ver en el siguiente programa (necesitas saber cómo se abren y utilizan ficheros en ANSI C):

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
   pid_t creado;
   FILE *fiche;

   fiche = fopen("pepe", "w");      /* abro un fichero */

   creado = fork();               /* creo dos procesos */

   if (creado == 0)   /* hijo */
   {
        puts("Soy el hijo"); /* no necesita abrir el fichero */
        fputs("Escribe el hijo\n", fiche);
        fclose(fiche);
   }
   else if (creado > 0)   /* padre */
   {
        sleep(4);   /* para asegurar que lo cierra el anterior */
        fputs("Escribe el padre\n", fiche);
        printf("Soy el padre, el hijo es %d \n",creado);
   }
   else
        perror("El fork no ha funcionado");
}
```

El fichero *pepe* será conocido por los dos procesos y no necesitarán abrirlo (cada proceso mantiene su copia del descriptor del fichero). Por lo tanto, una vez realizado el `fork` los cierres de cada uno no afectan al otro. Además, al tener el mismo puntero interno de escritura, no se machacan lo escrito por ambos.

> 11) Descárgate el programa `fichero.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. Muestra el fichero creado en pantalla. Crea una copia y cambia el programa para que ahora tanto padre como hijo abran el fichero después del `fork`. Ejecútalo y comprueba de nuevo el contenido del fichero. ¿cuál es la diferencia?

Otras identificaciones

A parte del PID del propio proceso, existen otros identificadores de interés como son: el número del proceso padre PPID y el número del grupo de procesos GID (al igual que los usuarios, los procesos están agrupados por grupos). El GID es heredado del proceso padre, por lo que el sistema antiguamente usaba este identificador para que todos los procesos de la sesión de un usuario fueran abortados cuando se salía del sistema, ya que todos tienen el mismo GID que la *shell* que los arrancó y ésta se encargaba de matar a sus procesos.

En la actualidad en POSIX se manejan dos conceptos: el grupo de procesos y la sesión de trabajo. La *shell* cuando crea a un proceso, lo nombra líder de esa sesión, y todos sus procesos hijo tendrán el mismo identificador de grupo que vendrá dado por el PID del padre.

Estos identificadores se pueden conseguir a través de las llamadas al sistema (también existe la posibilidad de ponerlos):

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
gid = getpgrp();
```

En el siguiente programa se describe su uso:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
      int ppid, pid, gid;
      pid = fork();

      if (pid == 0)   /* hijo */
      {
            pid = getpid();
            ppid = getppid();
            gid = getpgrp();
            printf("El pid es %d, el ppid es %d, y el gid es %d\n",
                                                      pid, ppid, gid);
      }

      else if (pid > 0)    /* padre */
      {
            pid = getpid();
            ppid = getppid();
            gid = getpgrp();
            printf("El pid es %d, el ppid es %d, y el gid es %d\n",pid,
            ppid, gid);
      }
```

```
        else
        {
                perror("El fork no ha funcionado");
                exit(1);
        }
}
```

12) Descárgate el programa `identifi.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. Indica los resultados que aparecen en pantalla e interprétalos.