TRABAJANDO CON PROCESOS EN UNIX

Esta semana entramos de lleno en el campo de la **concurrencia**, donde estudiamos la interacción entre procesos que se ejecutan de forma simultánea ("concurrentemente, a la vez"). Efectivamente, Unix/Linux fue diseñado para dar soporte a la multitarea, de tal modo que puede haber en el sistema más procesos ejecutándose que número de procesadores tenga el hardware del ordenador. ¿Queréis ver cuántos procesos hay ahora mismo corriendo en vuestra máquina? Ya conocemos el comando *ps aux*, pero existe una utilidad que se asemeja más al administrador de tareas de Windows, a pesar de ejecutarse en modo texto en el terminal. Si tecleas este comando:

top

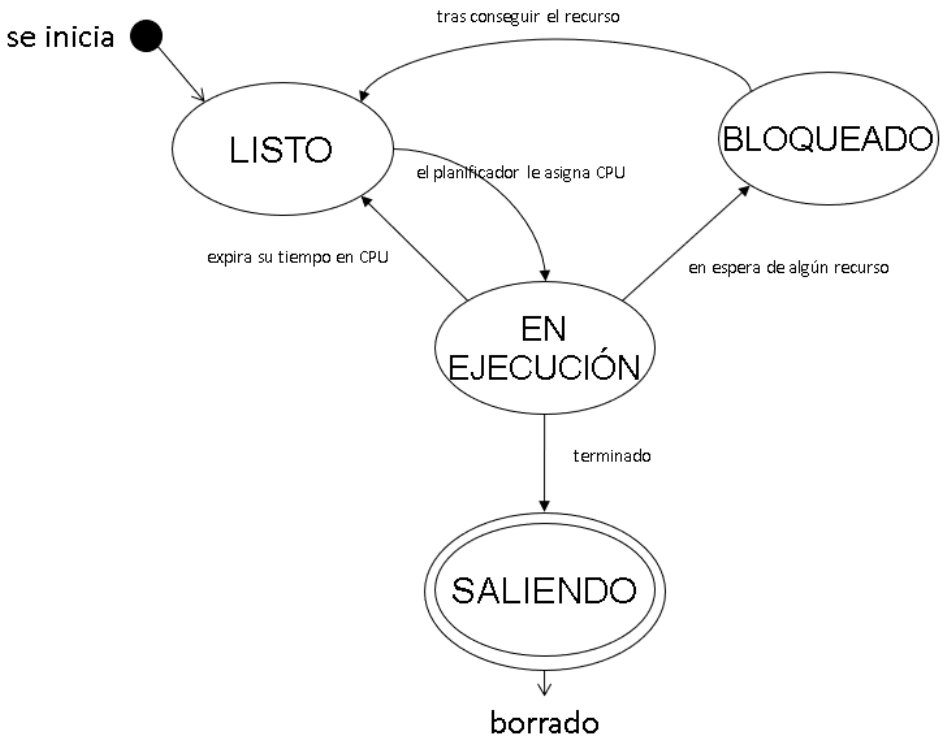
verás que aparece información exhaustiva sobre los procesos y su consumo de memoria, CPU, etc. Mira con atención una línea donde se enumeran las Tasks (procesos) del sistema. Hay procesos que están ejecutándose en CPU (running), otros que están esperando sin gastar CPU (sleeping, stopped), e incluso existe la posibilidad de que haya zombies ¡sí, has leído bien, eso lo veremos más adelante!

```
top - 12:27:25 up  2:51,  4 users,  load average: 4.37, 3.64, 3.44
Tasks: 194 total,   2 running, 192 sleeping,   0 stopped,   0 zombie
Cpu(s): 57.0%us,  1.3%sy,  0.0%ni, 41.1%id,  0.0%wa,  0.4%hi,  0.1%si,  0.0%st
```

El sistema operativo cumple una máxima "solo puede haber un proceso/tarea ejecutándose en una CPU en un momento dado". Esto tiene una implicación importantísima: debe haber un mecanismo en el sistema operativo que decida qué procesos (de entre todos los que aparecen listados en el comando top), ocupan en un momento dado cada CPU. A este mecanismo se le denomina **planificador de tareas** (scheduler en inglés), y es el responsable de que a nosotros nos dé la sensación de que todos los procesos se están ejecutando al mismo tiempo (no es verdad, ya sabemos que en cada momento solo se están ejecutando en realidad un número de procesos que corresponde al número de CPUs de nuestra máquina).

Para dar la sensación de multitarea, el planificador encola a todos los procesos que están listos para ejecutarse, y va sacando de la cola a procesos para asignarles un procesador determinado. Sin embargo, y esta es la clave, no deja que un proceso ocupe la CPU indefinidamente o hasta que tenga a bien terminar, sino que sólo le permite un tiempo máximo de ocupación de CPU, que se denomina ranura de tiempo (time slice en inglés). Cuando pasa su tiempo, aunque no haya terminado su ejecución, lo manda de nuevo a la cola de procesos hasta que le llegue de nuevo su turno. De esta manera, el planificador consigue que todos los procesos del sistema vayan progresando, y de ahí que a nosotros nos parezca que se están ejecutando simultáneamente.

De hecho, el ciclo de vida (y muerte) de un proceso es el siguiente:



El diagrama que estás viendo en la figura, está descrito utilizando una notación visual muy común en software de comunicaciones. Se denomina máquina o diagrama de estados, y permite describir de manera formal y no ambigua el comportamiento de un sistema o proceso. Cada forma ovalada representa un **estado** estable del programa, en el que sus variables tienen un valor fijo y no cambian hasta que se produce determinado evento, que lo hace seguir ejecutándose hasta el siguiente estado. Los eventos y su efecto, en este lenguaje visual se representan con flechas, denominadas **transiciones**, que indican un estado de partida y un estado destino. El pequeño punto negro de la figura y el óvalo doble son estados especiales: el primero indica el estado inicial de la máquina (dónde empieza la ejecución), y el segundo indica su estado final.

Vete familiarizando con esta forma de dibujar el comportamiento de un proceso que evoluciona a partir de eventos de entrada, puesto que es una forma habitual de especificar protocolos (normas) en programas de telecomunicaciones. Es tan importante en nuestro ámbito por su naturaleza no ambigua. Esto quiere decir que, a partir de una especificación expresada como una máquina de estados, existe una única forma de programar nuestro sistema para que cumpla dicha especificación, sin la posibilidad de que se produzcan errores de interpretación humana (qué es lo que nos han pedido que hagamos). En próximas sesiones aprenderemos a generar código C a partir de una máquina de estados, puesto que esta asignatura se llama Fundamentos de Software de Comunicaciones. Sin embargo, debes saber que existen herramientas visuales que se utilizan en la industria, en las que se puede dibujar una máquina y luego, de forma automática, se puede obtener el esqueleto de código en el lenguaje de programación que se necesite, que representa exactamente la especificación dibujada.

Pero volvamos al ciclo de vida del proceso. El proceso, cuando se carga en memoria y el sistema operativo le asigna su PID, entra en el estado LISTO, por el que puede ocupar un lugar en la cola de procesos que el planificador irá seleccionando para que, progresivamente, se ejecuten en ranuras de tiempo. A partir de aquí puede pasar que el planificador le asigne, efectivamente,

una CPU, por lo que el proceso transita hacia el estado EN EJECUCIÓN y permanece allí hasta que se produzca alguno de estos eventos:

-se ha alcanzado el tiempo máximo en el que se le permite ocupar la CPU. Esto significa que ha agotado su ranura temporal, pero como el proceso no ha terminado su ejecución, debe volver a la cola de procesos esperando, por lo que pasa de nuevo al estado LISTO

-el proceso ha terminado su ejecución (el programa ha finalizado). En este caso se pasa al estado final SALIENDO, y el sistema operativo ya se encarga de liberar su PID, descargarlo de memoria, cerrar los descriptores de fichero que dejó abiertos (si fuera necesario), etc.

-el proceso no puede continuar su ejecución normal porque está esperando un recurso. ¿Qué es esto de un recurso? Es un elemento que falta para poder continuar la ejecución normal del programa. Podemos pensar en situaciones que ya hemos practicado en el curso, como estar esperando que llegue una señal con pause(), o estar esperando a que el usuario introduzca algo por teclado con scanf() o read() del descriptor 0. En este caso, el sistema operativo hace que el proceso abandone inmediatamente la CPU, llevándolo al estado BLOQUEADO.

¿Cuándo se sale del estado BLOQUEADO? En el momento que se consigue el recurso necesario (llega una señal, el usuario introduce un texto y pulsa la tecla Intro...). Fíjate, y esto es muy importante, que el proceso no vuelve directamente a la CPU, sino que va de nuevo a figurar como LISTO y, por tanto, pasa a la cola de procesos listos para ser ejecutados. Esto implica un posible retardo en su reanudación.

Recuerda que un temporizador periódico implementado con alarm() es más sensible a este tipo de retardos, denominados derivas de tiempo. Por eso es tan importante utilizar setitimer() para resolver ese problema. También es importante recordar que ITIMER_REAL, como primer argumento de setitimer() permite tener en cuenta el tiempo total de transcurrido (incluido el que no es de ejecución en CPU, el de permanencia en la cola de procesos, o bloqueado). Si se quisiera medir solamente el tiempo que el proceso está ejecutándose en la CPU, se utilizaría ITIMER_VIRTUAL como primer argumento.

NOTA: setitimer() en modo ITIMER_VIRTUAL, al medir tiempo solo cuando el proceso está en la CPU, no contabiliza tiempo cuando se está en pause() ¡puesto que está bloqueado, no en ejecución! En ese caso, no esperes que el sistema genere la señal SIGVTALRM.

PROCESOS BLOQUEADOS Y SEÑALES

A partir de ahora, vamos a catalogar a las llamadas al sistema como bloqueantes o no bloqueantes. En el caso de que una llamada sea bloqueante, sabemos que se saca al proceso de la CPU y éste no puede volver a ejecutarse hasta obtener el recurso solicitado.

Sin embargo, se considera un recurso válido la llegada de una señal, lo que puede alterar el comportamiento de la llamada. En efecto, sabemos que en modo Unix System V (opciones de compilación -ansi, -std=c99 o -std=c11), un proceso que está bloqueado por una llamada al sistema, va a hacer que ésta devuelva -1 si llega una señal. La recepción de una señal en un proceso hace que pase de estado BLOQUEADO a LISTO y posteriormente, en algún momento, a EJECUCIÓN. Y es en ese momento cuando la llamada al sistema devolverá -1, poniendo la variable global *errno* al valor EINTR. No queda otra que volver a ejecutar de nuevo la llamada y volver a ejecutar el proceso para solicitar de nuevo el recurso que el programador está esperando de ella.

En la asignatura, por tanto, para cada llamada al sistema que identifiquemos como bloqueante, debemos estar preparados para reintentar su ejecución. Este es el bucle que debemos implementar:

```
int leidos;
char buffer[TAM_BUFFER];

do{
  errno = 0;
  leidos = read(fd, buffer, TAM_BUFFER);
}while( (leidos < 0) && (errno == EINTR) );
```

Nótese cómo se ha de reiniciar *errno* a cero (que significa que no hubo error) en cada posible iteración del bucle, para no confundir al siguiente chequeo de la variable.

LANZAR PROCESOS DESDE PROGRAMAS EN C: CLONES Y MUTANTES

En software de comunicaciones es habitual trabajar con dos conceptos relacionados con los procesos: la clonación y la mutación. El caso más frecuente es el de un proceso de servicio que espera una petición, a través de un canal de comunicaciones, por parte de un cliente. Si esta petición llega y el proceso se pone a atenderla, deja en cola a otros clientes que estén pidiendo el servicio. Si los clientes esperan mucho en cola, puede que se cansen y se vayan. Esto no ayuda a la buena reputación de nuestro software (como tampoco pasaría en la vida real).

La solución para atender a más de un cliente de forma simultánea implica que el proceso de servicio genere otros procesos, denominados hijos, que se encarguen de atender a cada cliente de forma separada, mientras el proceso original (padre) es capaz de volver a estar pendiente de la llegada de una nueva petición (¡sin esperar a que el hijo haya terminado su tarea!). Así sí que estarían trabajando de forma concurrente estos procesos padre e hijos.

El código clásico que nos vamos a encontrar es siempre así:

```
mientras(normalmente bucle infinito){
    espero_una_peticion_de_servicio (espero leer datos de un canal/descriptor de fichero)
    genero un hijo -> es el hijo quien atiende al cliente
}
```

La clave aquí es la generación de un hijo. Habitualmente, el proceso padre se clona y prepara una zona de código especial que debe ejecutar el clon (el hijo). En algunos casos, además, el hijo atiende al cliente usando ese código programado por nosotros, o bien muta y se convierte en otro proceso (para ejecutar su propia funcionalidad) de entre los disponibles en el sistema operativo.

A partir de aquí, se van a explicar paso a paso estos conceptos. Por su importancia, pero también por su complejidad, te ruego que prestes mucha atención.

CLONES

¡En Unix no existe el debate ético sobre la clonación que sí tenemos en nuestra sociedad! Aquí se va a trabajar a lo bestia: un proceso genera un clon de sí mismo para que le haga trabajo que él NO quiere hacer. Tal cual. Es un clásico ejemplo de arquitectura maestro/esclavo. Clones y esclavos, no te asustes y sigue leyendo.

¿En qué consiste generar un proceso clon? Cuando un proceso, con PID X, ejecuta una llamada al sistema especial para clonar, el sistema operativo crea un clon. Este clon es un proceso exactamente idéntico al original, que se carga en memoria y pasa a la cola de procesos LISTOS para ser ejecutados. El clon tiene su propio PID único, distinto al del original, pongamos que tiene valor Y. Pero, y aquí se empieza la cosa a poner interesante, si ejecutamos getppid() en el clon, obtenemos el valor X. Es decir, el proceso original es PADRE y el clon es HIJO, si atendemos a la jerarquía de procesos que implementa el sistema operativo.

Recuerda que en Unix todos los procesos son familia, derivando de un proceso padre de todos llamado init (que tiene el PID de valor 1). De hecho, cada vez que ejecutamos un comando o un programa utilizando el terminal, el proceso cargado en memoria y ejecutado es hijo del terminal.

Un clon, por tanto, es un duplicado en memoria del proceso original. Y con la memoria, se copia el contenido de la pila y del heap del proceso original al clon. También se copia la zona de código, las variables globales ¡Todo! Incluso la tabla de descriptores de ficheros se copia completa, por lo que el clon tiene acceso a los ficheros a través de los mismos descriptores abiertos que el original. Hasta el contador de programa se pone en la misma posición en el clon que donde está en el original.

A ver si se entienden bien las implicaciones de esto:

-El proceso clon tiene el mismo código que el proceso original, así que ejecutarán las mismas instrucciones.

-Si el proceso original tenía en su pila una variable x con valor 5 antes de hacer la clonación, al clonar, el proceso clon también tiene una variable x en su pila de valor 5.

-Siguiendo el ejemplo anterior, si después de la clonación el proceso original cambia el valor de su variable de pila a x = 6, el clon sigue teniendo su variable x con valor 5. Recuerda que cada proceso se ejecuta en su propia zona de memoria independiente, aunque ejecuten el mismo programa. Lo mismo sucede si el clon hace cambios a sus variables, crea nuevas, etc. Al original no le repercuten dichas acciones.

-En el momento en que se crea el clon, éste pasa a la cola de procesos LISTOS esperando ser ejecutados, donde también puede acabar el proceso original. El planificador puede decidir que el clon entre antes en CPU que el original, y se ejecute antes. Da igual que sea "el último en llegar", lo que tiene una implicación severa en el diseño de estos programas padre/hijo, maestro/esclavo o como se quieran denominar: JAMÁS SE HA DE ASUMIR EL ORDEN RELATIVO EN QUE SE VAN A EJECUTAR UNO Y OTRO. Simplemente, no está en manos del programador. Hablaremos sobre esto con más cuidado en la siguiente sesión, en el que comunicaremos a padres con hijos y viceversa, y lo deberemos tener en cuenta.

-Si el proceso original hizo una apertura de un fichero o canal de comunicaciones antes de la clonación, tras ésta, el clon también tiene acceso al fichero o canal, a través de su descriptor de fichero copiado.

-Clonar no equivale a ejecutar un programa dos veces seguidas en la consola. Efectivamente, en ese caso aparecen dos procesos A y B en el sistema que ejecutan un mismo código, pero ambos son hijos del terminal, y no se cumple entonces que B sea hijo de A. Tampoco comparten ficheros abiertos, etc.

LLAMADA AL SISTEMA PARA CLONAR: fork()

Un programa en C, en el momento que quiera hacer un clon de sí mismo, necesita ejecutar la llamada al sistema fork(). Veamos el programa más simple posible:

```
1  int main(){
2          int x = 5;
3          pid_t pid = fork();
4          if(pid < 0){
5                  perror("falla fork");
6                  exit(1);
7          }
8          printf("Hola\n");
9          return 0;
10 }
```

Al ejecutar este programa, se convierte en proceso en el sistema y su contador de programa comienza en la primera línea (es el comportamiento normal, nada nuevo). En el momento que se ejecuta fork (línea 3), y si no falla devolviendo -1, aparece un proceso idéntico a este original que se estaba ejecutando. Recuerda, el contador de programa del proceso original está en la línea 3, ¿y el del clon? También en la línea 3 ¡No empieza a ejecutarse por el principio! Vamos a recalcarlo con esta figura:

```
1  int main(){                          1  int main(){
2          int x = 5;                   2          int x = 5;
3          pid_t pid = fork();          3          pid_t pid = fork();
4          if(pid < 0){                 4          if(pid < 0){
5                  perror("falla fork"); 5                  perror("falla fork");
6                  exit(1);             6                  exit(1);
7          }                            7          }
8          printf("Hola\n");            8          printf("Hola\n");
9          return 0;                    9          return 0;
10 }                                    10 }
```

Ahora hay dos procesos en el sistema, cada uno con su propio PID, los dos tienen el mismo código, y van por la línea 3, puesto que siguen ejecutándose (cuando les toque y en el orden que decida el planificador) al salir de la llamada fork().

Y en la pantalla del terminal, como no podía ser de otra manera, va a aparecer dos veces el mensaje "Hola\n".

Pero ¿no habías dicho hace un momento que el proceso original (padre) creaba al clon (hijo) para hacer trabajos que él no quisiera hacer? De eso nada, los dos comparten el mismo código así que hacen exactamente lo mismo.

Efectivamente, el programa es el mismo, pero ¿y si pudiéramos de alguna forma distinguir en el código si estamos ejecutando el padre o el hijo? ¿Podríamos, en función de eso, decidir hacer unas acciones u otras?

La respuesta está en ese misterioso valor de retorno de fork() que he pasado por alto a posta hasta llegar aquí. Resulta que fork() es la ÚNICA LLAMADA AL SISTEMA QUE DEVUELVE DOS VALORES AL MISMO TIEMPO. **En el proceso padre, al salir del fork(), esta llamada devuelve un valor positivo. En el proceso hijo, al salir del fork(), esta llamada devuelve cero**.

Con esta nueva y valiosa información ahora podemos asignar tareas distintas al proceso padre y al hijo así (nótese la variación al código original):

```
1  int main(){
2          int x = 5;
3          pid_t pid = fork();
4          if(pid < 0){
5                  perror("falla fork");
6                  exit(1);
7          }
8          if(pid == 0){
9                  printf("Hola, soy el hijo\n");
10         }else{
11                 printf("Hola, soy el padre\n");
12         }
13         return 0;
14 }
```

```
1  int main(){
2          int x = 5;
3          pid_t pid = fork();
4          if(pid < 0){
5                  perror("falla fork");
6                  exit(1);
7          }
8          if(pid == 0){
9                  printf("Hola, soy el hijo\n");
10         }else{
11                 printf("Hola, soy el padre\n");
12         }
13         return 0;
14 }
```

En la figura, aún sin especificar qué proceso es el padre y cuál el hijo, se puede determinar fácilmente por el valor de cada contador de programa en un momento futuro de la ejecución (el contador de programa aparece en la columna del número de línea, con un color sombreado). La condición pid == 0 solo es verdad en el hijo, así que solo el hijo puede llegar a imprimir el texto "Hola, soy el hijo\n". El padre no entra en esa zona de código puesto que, para él, pid == 0 es falso, así que imprime "Hola, soy el padre\n". ¡Hemos conseguido que el proceso original y el clon puedan hacer tareas distintas!

Por lo tanto, tras el chequeo de que fork() ha funcionado, haremos siempre una ZONA EXCLUSIVA, mediante un if/else, para garantizar líneas de código a ejecutar únicamente por el proceso original y por el clon. Es muy común ver algún tipo de alternativa como:

```
if(pid >0 ){
    //zona exclusiva del padre
}else{
    //zona exclusiva del hijo
}
```

Hay un detalle que he dejado para el final, pero que también tiene importancia. Te habrás dado cuenta que el resultado de fork() se lo he asignado a una variable a la que, en este ejemplo, he llamado pid, de tipo pid_t. Que se llame así, y no de otra manera, no es casual. Recuerda que hemos dicho que, en el proceso padre, la llamada a fork() devuelve un valor positivo (siempre que no falle, claro). Pues bien, da la interesante circunstancia que ese valor positivo corresponde

al PID del proceso hijo que ha generado. ¡Es una forma de que el padre sepa hablar con su hijo, por ejemplo, mandándole una señal!

En la zona exclusiva de ejecución del padre, tras un calentón, se podría escribir:

kill(pid, SIGKILL);

convirtiéndolo así en asesino de su propio clon, y sin responsabilidad penal alguna.

Aparte de la broma, ¿piensas que no hay duda de que esa señal llegará al hijo? La respuesta es "puede que no llegue", si el hijo termina su ejecución antes de que el padre llegue a ejecutar ese kill(). Recuerda que nadie te asegura el orden de ejecución del planificador y, evidentemente, cuando un proceso termina, ya no puede recibir señales.

Ya que estamos, ¿podría el proceso hijo comunicarse con su proceso padre utilizando también señales? Espero que no pienses que en su variable pid ha almacenado el identificador de proceso de su padre. En el hijo, fork() devuelve cero, y eso no es un valor de PID válido (el proceso init, el padre de todos, tiene el menor PID, y es 1). Para encontrar el PID de su padre, tendrá que recurrir a

pid_t pid_de_mi_padre = getppid();

Debes seguir preguntándote: ¿y qué pasa si el proceso padre acaba su ejecución antes de que el hijo? ¿que devuelve ahora getppid()? ¿Un PID que ya no es válido en el sistema operativo? No, en caso de que el padre acabe antes que el hijo, el proceso hijo queda HUÉRFANO (tal cual en terminología Unix), y es adoptado por el init, que pasa a ser su nuevo padre. De todas formas, no es habitual que un proceso padre termine antes que sus hijos, al menos en software de comunicaciones puesto que, en nuestra forma de diseñar, los clones realizan un trabajo para el proceso original, y éste, de alguna manera se ha de beneficiar de él. Lo normal es que el padre sea el último proceso en acabar. Veremos cómo asegurarnos de esto un poco más adelante.

GENERANDO MÁS DE UN CLON

Supón que un proceso tiene que crear dos clones, en vez de uno solo. Espero que este código te genere el mismo sarpullido que a mí, solo con verlo:

```
int main(){
        pid_t pid1 = fork();
        if(pid1 < 0){
                perror("falla fork");
                exit(1);
        }
        pid_t pid2 = fork();
        if(pid2 < 0){
                perror("falla fork");
                exit(1);
        }
        return 0;
}
```

¿Entiendes qué está pasando? Tras el primer fork(), aparece un clon que está ejecutando el mismo programa, así que el original y la copia ejecutan cada uno por separado el segundo fork(). De esta forma, el proceso original genera un segundo hijo, pero el clon también genera otro hijo. Como resultado, el proceso original tiene dos hijos, ¡pero también un nieto!

Claro, se nos ha olvidado poner la ZONA EXCLUSIVA, para conseguir que el original y sus clones ejecuten tareas distintas. Esta es la solución, que ya te estabas imaginando:

```
int main(){
	pid_t pid1 = fork();
	if(pid1 < 0){
		perror("falla fork");
		exit(1);
	}
	if(pid1 > 0){ //zona exclusiva padre
		pid_t pid2 = fork();
		if(pid2 < 0){
			perror("falla fork");
			exit(1);
		}
		if(pid2 > 0){ //zona exclusiva padre

		}else{ //zona exclusiva hijo 2

		}
	}else{ //zona exclusiva hijo 1

	}
	return 0;
}
```

Como en nuestros programas de servicio lo normal será tener un bucle para ir creando hijos a medida que lleguen nuevas peticiones de clientes, vamos a generalizar el razonamiento anterior para la creación de N hijos. De nuevo, este programa esconde un error nefasto:

```
int main(){
	while(1){ //bucle infinito de servicio
		//espera una peticion de un cliente
		//...
		//y hace fork para atenderla en un clon
		pid_t pid = fork();
		if(pid < 0){
			perror("falla fork");
			exit(1);
		}
		if(pid == 0){ //zona exclusiva del hijo
			//atiende al cliente
			//...
		}
		//se vuelve al bucle, a esperar una nueva peticion
```

```
        }
        return 0;
}
```

¿Has visto el fallo antes de compilarlo y ejecutarlo por ti mismo? Espero que sí porque, de nuevo, está mal implementada la ZONA EXCLUSIVA. Fíjate que es el padre el que debería volver al bucle para seguir esperando peticiones y generando el siguiente clon. Sin embargo, cuando el proceso hijo acaba sus tareas, ¡también vuelve al bucle! Y también genera hijos que a su vez generarán hijos... Dependiendo de en qué consista la espera de petición de servicio, podría darse el caso de que el sistema se llene de procesos, que ocupan sus propios recursos en el sistema, y deje de funcionar correctamente. Es un fallo grave de diseño. La solución, claro está, consiste en asegurarte de que, cuando un proceso hijo termine, NO SIGA EJECUTANDO MÁS CÓDIGO. En el ejemplo anterior, la zona exclusiva de código del hijo debe finalizar con una sentencia que termine el proceso. Quizá así:

```
if(pid == 0){ //zona exclusiva del hijo
        //atiende al cliente
        //...
        exit(0);
}
```

EL CONTRATO ENTRE PADRES E HIJOS

Unix nos ha demostrado no tener ningún tipo de escrúpulo ni ética en el modo en que un proceso genera y gestiona a sus hijos: clones, esclavos, huérfanos... Pero, en este océano de iniquidad, se puede encontrar una isla de responsabilidades que se asemejan más a las normas que cumplimos en nuestra sociedad. Es lo que vamos a denominar el contrato.

En esta asignatura vamos a hacer que, tanto procesos padres como hijos, cumplan cada uno un contrato obligatorio.

El contrato del hijo:

-los hijos siempre deben acabar su ejecución dentro de su ZONA EXCLUSIVA. Podemos usar la función de biblioteca exit() para obligar a cumplirlo.

El contrato del padre:

-por diseño, un padre siempre tiene que acabar más tarde que todos sus hijos. Además, tiene que evitar que se conviertan en ZOMBIES. ¡Por fin aparece este concepto!

Cuando un hijo termina, el sistema operativo lo saca de la memoria y de la lista de procesos ejecutables, pero no libera del todo los recursos asociados a él, hasta que el padre da su consentimiento. Mientras tanto, el hijo queda aún en el sistema en un estado DIFUNTO (defunct en inglés) lo que también se denomina ZOMBIE. Cuando hagas tus propios programas, puedes forzar a que el padre termine más tarde que el hijo de alguna forma (por ejemplo, poniendo un pause() en su zona exclusiva), y en otro terminal, puedes investigar con alguno de estos comandos

top

ps -el

si los hijos se quedan zombies: en ambos comandos hay una columna denominada S (de State), donde los procesos aparecen como R (running), S (sleeping) o D (defunct).

Para que el padre pueda cumplir este contrato (esperar a sus hijos para que no queden zombies en el sistema, ocupando recursos), es obligatorio que utilice la llamada al sistema wait(0). Esta llamada bloquea al padre hasta que el sistema operativo le notifique que su hijo ha terminado. ¿Y cómo se da esta notificación? Cuando el sistema operativo detecta que un proceso ha terminado, le envía a su padre la señal SIGCHLD. La condición de desbloqueo de wait(0) es, por tanto, la recepción de esa señal.

Este es un ejemplo de código donde se crea un clon y se cumplen los contratos para el padre y para el hijo de una manera correcta. Debe servir de modelo para todos vuestros programas a partir de ahora:

```
int main(){
	pid_t pid = fork();
	if(pid < 0){
		perror("falla fork");
		exit(1);
	}
	if(pid > 0){ //zona exclusiva padre
		//...
		wait(0); /* contrato: espera a que termine su hijo */
	}else{ //zona exclusiva hijo
		//...
		exit(0); /* contrato: siempre termina antes de abandonar la zona exclusiva */
	}
	return 0;
}
```

NOTA: Si miras el prototipo de wait() en el manual (man 2 wait), verás que es así:

```
pid_t wait(int *wstatus);
```

Nosotros no vamos a utilizar nunca el argumento de wait(), y como necesita una dirección de memoria, directamente podemos utilizar wait(NULL) o wait(0), que resulta equivalente.

Falta un último detalle, y ya terminamos con las guerras clon. Si hemos dicho que wait() es una llamada al sistema BLOQUEANTE, estamos haciendo que nuestro bucle de servicio FALLE ESTREPITOSAMENTE:

```
int main(){
	while(1){ //bucle infinito de servicio
		//espera una peticion de un cliente
		//...
		//y hace fork para atenderla en un clon
		pid_t pid = fork();
		if(pid < 0){
			perror("falla fork");
			exit(1);
		}
```

```
		if(pid == 0){ //zona exclusiva del hijo
			//atiende al cliente
			//...
			exit(0); //cumple contrato
		}else{
			wait(0); //cumple contrato ¡ pero se bloquea!
		}
		//si llegas aqui, tu hijo ya ha terminado
		//como padre, vuelvo al bucle a esperar una nueva peticion
	}
	return 0;
}
```

Sigue el razonamiento. Todo este lío de los clones lo hemos montado por una única razón: queremos que se pueda atender de manera simultánea a los clientes que vayan requiriendo un servicio, aprovechando las capacidades multitarea del sistema operativo. Para ello, hemos decidido que cada petición de servicio se va a delegar a un proceso hijo, para que el padre pueda volver inmediatamente a esperar a un nuevo cliente. Esto implica que el padre y sus hijos tienen que trabajar de manera CONCURRENTE. El código anterior incumple esa premisa: ¡si el padre ha de esperar bloqueado en wait(0) hasta que termine su hijo, no se está ejecutando concurrentemente con él! Dicho de otra manera, si el proceso padre tiene que esperar hasta que un cliente sea completamente servido antes de poder atender una nueva petición, para eso no hacen falta clones, todo el servicio podría hacerse en el proceso original. Eso sí, haciendo que los clientes en cola desesperen y se vayan del sistema, hartos de esperar.

Pero parece que hemos llegado a un callejón sin salida: ¿cómo permitir la concurrencia, evitando ese bloqueo en wait(0), pero cumplir a la vez el contrato para que los clones no queden zombies? Como experto en programación con señales, creo que ya estás adivinando posible solución. Puesto que el sistema operativo manda al padre una señal SIGCHLD cuando un hijo termina, podemos decir en nuestro programa que, ante la llegada de dicha señal, se ejecute una función manejadora, y sea allí donde se realice la llamada a wait(0), ¡que en este caso ya NO será bloqueante! Este es el código correcto utilizando signal():

```
void manejadora(int numero_senal){
	wait(0); //cumple contrato, no hay bloqueo
	signal(numero_senal, manejadora);
}

int main(){
	signal(SIGCHLD, manejadora);

	while(1){ //bucle infinito de servicio
		//espera una peticion de un cliente
		//...
		//y hace fork para atenderla en un clon
		pid_t pid = fork();
		if(pid < 0){
			perror("falla fork");
			exit(1);
		}
```

```
                if(pid == 0){ //zona exclusiva del hijo
                        //atiende al cliente
                        //...
                        exit(0); //cumple contrato
                }
                //como padre, vuelvo al bucle a esperar una nueva peticion
        }
        return 0;
}
```

En Linux, pero no en todos los Unix, ojo, también se le puede decir al sistema operativo que no queremos recibir la señal SIGCHLD. Cuando hacemos en el programa esta llamada:

signal(SIGCHLD, SIG_IGN);

el sistema operativo detecta que no te interesa lo que le pase a tu hijo y, en ese caso, libera inmediatamente los recursos asociados al clon cuando éste termina. Es decir, ya no lo deja zombie, y libera también al padre de la necesidad de hacer wait(0). Sin embargo, esta opción hace el código NO PORTABLE, así que preferimos utilizar el código que registra una manejadora donde se llame a wait(0).

MUTACIONES DE PROCESOS

Existe una familia de funciones que empiezan por exec*(), que permiten que un proceso se convierta en otro. El término mutación es la forma que utilizamos en esta asignatura para referirnos a este hecho. ¿Qué pasa cuando un proceso muta y se convierte en otro? Que sigue manteniendo su PID y sigue siendo hijo del mismo padre, pero (y aquí está la gracia), el proceso original se descarga de memoria completamente y se carga la memoria del nuevo, que lo reemplaza, para ser ejecutado desde el principio de su main.

Existen diversas variantes de exec*(), dependiendo de los argumentos que se utilicen, pero todas indican el proceso al que mutar y, si los tiene, los argumentos que hay que pasarle a este nuevo proceso. Pongo aquí un ejemplo con la llamada execl()

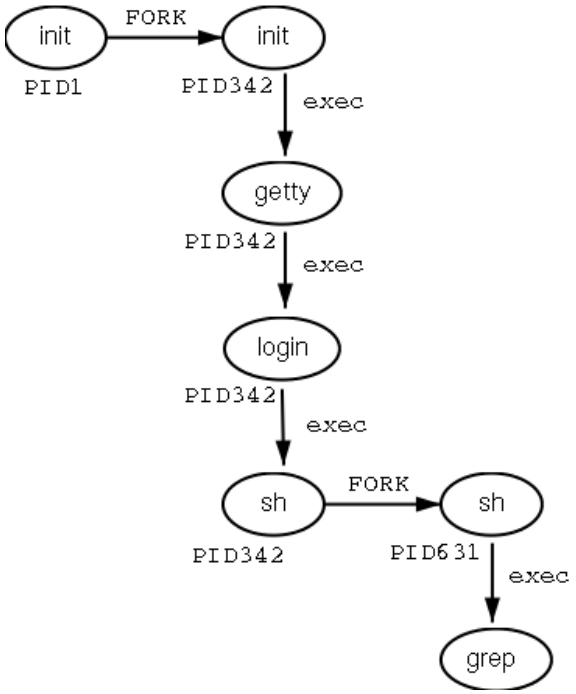
```
int resultado = execl("/usr/bin/man", "man", "2", "signal", NULL);
/* si execl() ha tenido exito este codigo que viene a continuacion YA NO SE EJECUTA (ya que
este proceso es reemplazado por el nuevo) */
if(resultado == -1){
        /* pero si ha fallado la mutacion, se sigue por aqui */
}
```

La llamada a execl() tiene una cantidad de argumentos variable, como mínimo dos, que son cadenas de caracteres:
-el primer argumento es la cadena con el comando o programa ejecutable. Nótese que en el ejemplo se ha utilizado la ruta absoluta al comando man (esta ruta se puede consultar a su vez utilizando el comando de consola: which ls), pero podría haberse utilizado una ruta relativa al directorio en el que se está ejecutando el proceso original (con más cuidado, claro).
-los demás argumentos son cadenas que se quieren pasar al nuevo proceso, empezando, por supuesto, por el nombre del programa y después enumerando cada argumento. Esto es lo que

el sistema operativo copiará en el argumento char *argv[] del main del nuevo proceso. Nótese cómo, cuando no se necesiten más argumentos, se pondrá un último argumento a NULL (o a cero).

Esta forma de trabajar es muy habitual en Unix para reutilizar programas y no reinventar la rueda. Y, además, casi siempre se utiliza combinado con la clonación. Por ejemplo, supón que necesitas un clon cuyo único cometido es obtener la fecha al sistema operativo y mostrarla por pantalla. Desde luego que en su ZONA EXCLUSIVA puedes implementar este código en C, pero si ya existe un programa externo que lo hace, ¿por qué no usarlo? De esta forma, el clon se convierte (muta) en el programa date (comprueba que existe en el terminal) y cumple su cometido.

De hecho, esto es lo que hace el terminal cada vez que le decimos que ejecute un programa. El terminal se clona a sí mismo, y luego muta al proceso que haya solicitado el usuario. Mira esta figura:



El proceso init se clona y luego muta al proceso de control de terminal de arranque, que a su vez muta en el proceso de login para que un usuario pueda entrar en el sistema. Si el login es correcto, el proceso login muta a una terminal (aquí es una shell de tipo sh), y ya se pueden teclear comandos en ella. Si, por ejemplo, el usuario introduce el comando grep y le da a la tecla Intro, la shell se clona y luego muta hasta ejecutar dicho comando. Este ejemplo no es ajeno a cómo funciona vuestro Linux, lo que pasa es que normalmente se tiene configurado un arranque gráfico, y no en modo texto. Así, el login, inmediatamente mutaría al proceso que arranca el entorno de ventanas.

LA FUNCIÓN DE BIBLIOTECA system()

Existe una función muy útil de la biblioteca de C (no es una llamada al sistema), que se llama system() (man 3 system). Es estupenda para ejecutar comandos del sistema operativo desde nuestros programas. Ejemplo:

```
int main(){
        printf("Hoy es:\n");
        system("date"); //muestra la fecha
        system("clear"); //limpia la pantalla
        printf("Que gusto da una pantalla limpia, oiga\n");
        return 0;
}
```

Nota que una llamada a system() no reemplaza tu programa por el que le pasas como argumento, y que tu programa se sigue ejecutando después, como si tal cosa. Sin embargo, para poder comportarse así, esta función, internamente, tiene que hacer uso de la clonación y de la mutación (mira la página del manual: man 3 system, para ver cómo se efectúa la mutación). ¿Serías capaz de crear tu propia función que emulara a system() y donde utilizaras todos los conceptos aprendidos en esta sesión: fork, wait, execl?