DISEÑO DE PROTOCOLOS. MÁQUINAS DE ESTADO

Hemos llegado, por fin, a las etapas finales de nuestro viaje. ¡Y vaya viaje! Por el camino, hemos aprendido que nuestros programas pueden y deben reaccionar ante la llegada de eventos (¡esas señales y temporizadores, como nos pueden dar la gloria!). También hemos manipulado canales de comunicaciones y, de forma sencilla, hemos aprendido a enviar información en bloques, y teniendo cuidado de convertir los datos enteros en el transmisor para que se interpreten correctamente en el receptor. Por supuesto, también nos hemos familiarizado con el concepto de servicio, y de cómo es posible dárselo a múltiples "clientes". Y hemos asimilado que para dar un servicio correcto entre dos procesos que se comunican, hace falta una sincronización entre ellos (espero recibir y luego envío una respuesta, esperamos los dos a estar listos para poder empezar a transmitir datos, etc.).

Espero que todo esto te haya ido calando ya de una forma natural, y sin ir a marchas forzadas o metiéndote conceptos con calzador (si te ha dado esta impresión, he hecho el ridículo espantoso como profesor tuyo y merezco lo peor en el limbo de los docentes a olvidar). Pero si todos los conceptos han cuajado ya en tu cabeza, y si miras hacia atrás y piensas en el punto de partida, te puede dar hasta vértigo de lo maduros que estamos. No sé si coincides conmigo en que ya no solo tenemos en nuestra cabeza el que nuestros programas compilen y echen a andar, sino que ahora, casi sin darnos cuenta, estamos muy pendientes de cómo vamos a utilizar el canal de comunicaciones, como vamos a montar el "diálogo" entre los procesos, cuántos bytes se envían y se reciben, qué significa cada mensaje que llega al otro extremo...

Lo que quiero decir es que estamos listos para empezar a hablar de diseños e implementaciones de protocolos de comunicaciones. Ya sabes, ese conjunto de reglas que definen cómo tienen que dialogar dos o más entidades. En nuestro caso, esas entidades son procesos, y nos da igual que estén corriendo en el espacio de memoria del usuario o dentro del kernel.

En esta sesión quiero hablarte un poco más de cómo se diseña un protocolo, para que entiendas mejor lo que a veces se presenta de forma abstracta en otras asignaturas. Así, en un futuro, podrás pertenecer a un equipo de compañeros y profesionales que se encarguen de transmitir a código fuente una norma internacional o documento que describe un protocolo concreto. Quizá el objetivo de ese software sea su ejecución en terminales y equipos de la red de telefonía móvil. O quizá lo metáis dentro de un router de Internet, o en una máquina virtual en la nube, o en un sensor, cualquiera sabe.

Tampoco te voy a engañar: diseñar un protocolo desde cero puede resultar una tarea compleja, que necesita de muchas pruebas, refinamiento y mucha experiencia. Y es complejo porque hay concurrencia implicada, canales y redes con distinto grado de fiabilidad, servicios que hay que cumplir en un tiempo determinado...

Por eso aquí nos vamos a centrar, si te parece bien, a entender cómo descifrar un diseño o norma de protocolo (nos vale con un enunciado de un ejercicio), y llevarlo al código fuente sin errores (es decir, cumpliendo todos los requisitos del servicio que se quiere dar).

ELEMENTOS DE UN PROTOCOLO

Parece totalmente necesario poder entender, entonces de qué consta un protocolo. Dicho de otra manera, ¿qué características tiene que lo hacen un software tan especial? Siempre que hablemos de protocolos de comunicaciones nos vamos a encontrar que es necesario definir perfectamente estos elementos:

1. Qué **servicio** da. O lo qué es lo mismo, qué se espera de él cuando lo utilizamos ¿Para qué sirve? ¿Qué es lo que hace? Cualquier documento donde se defina un protocolo os tiene que dejar claro esto, para poder probarlo y poder certificar que funciona correctamente antes de montarlo en un equipo y venderlo (o liberarlo para uso gratuito de la comunidad).

2. El entorno o **contexto** en el que se tiene que utilizar. Esto es fundamental, porque no es lo mismo un protocolo como TCP o UDP, que necesitan colas y gestionan datos provenientes desde y hacia procesos, que un protocolo como IP, que no está en contacto directo con los procesos de los usuarios de ninguna manera, sino con TCP, UDP y la/s tarjeta/s de red del equipo. El protocolo IP, por ejemplo, no espera que haya procesos de los usuarios encima de él, así que para él no tiene sentido utilizar un puerto para identificarlos. Y así con todo.

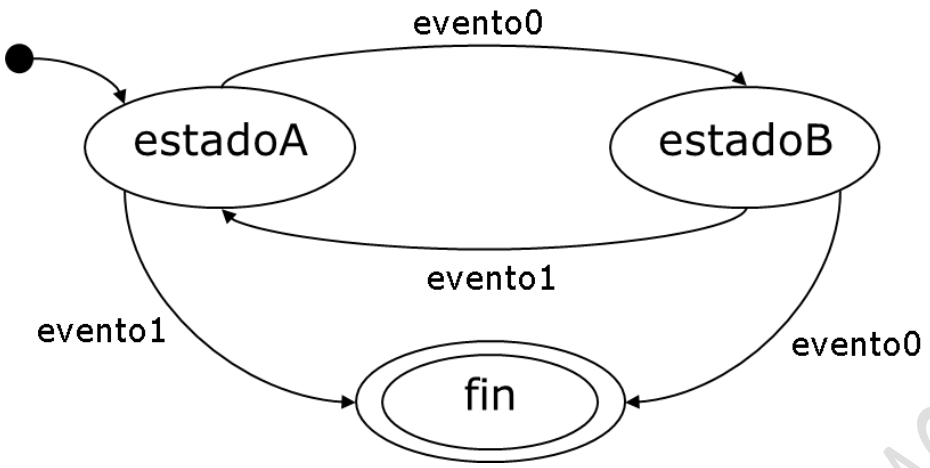
3. Su **vocabulario** de mensajes. Este concepto es fácil. Se refiere a los tipos de mensajes que utiliza el protocolo. Por ejemplo, puede haber un mensaje de solicitud de conexión, otro de datos, otro de desconexión, uno de aceptación a un mensaje recibido, otro de rechazo...

4. El **formato de cada mensaje**. Cuando al programador le indican que tiene que enviar un mensaje solicitando una conexión, tiene que ver cómo se construye dicho mensaje, cuántos campos tiene, de qué tipo son y cómo se rellenan, si hay que hacer alguna conversión o no a big endian, el orden en el que se montan los campos en el mensaje para enviarlos y luego interpretarlos en el receptor... Esto ya lo hemos hecho antes nosotros cuando hemos enviado primero un dato con la longitud de la cadena, y a continuación la cadena en sí. Así, el formato de este mensaje de ejemplo tendría dos campos, en este orden: campo longitud (tamaño prefijado en big endian) + campo datos (tamaño variable).

5. Por último, y también muy importante, las **reglas** de cómo proceder, de cómo programarlo. Tiene que quedar clara la secuencia de mensajes que se intercambia cada entidad participante, quién empieza a hablar, qué pasa si se excede un tiempo sin que aparezca un mensaje esperado, cómo reacciona el protocolo cada vez llega un mensaje u otro evento, qué hacer si no son están correctos o no se esperaban en ese momento...

Es en este último punto donde vamos a trabajar hoy. Asumimos que el protocolo debe ir avanzando en su ejecución y, según la situación en la que esté en cada momento, debe reaccionar de una forma o de otra.

La manera de describir este comportamiento, en software de comunicaciones, es clásica para utilizar las llamadas máquinas de estados ¿recuerdas que este concepto ya se introdujo para entender cómo funcionaba el ciclo de vida de un proceso? Pues vamos otra vez con a ello.



(ACABO DE ENTRAR EN MODO COPIA-PEGA TOTAL. ¡SIENTE EL DÉJÀ VU!) 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.

... (El concepto de máquina de estado) 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).

IMPLEMENTANDO NUESTRA PRIMERA MÁQUINA DE ESTADOS

La forma que os presento para implementar las máquinas de estado es una bastante habitual. Vayamos paso a paso:

Lo primero que tiene que quedar claro en el momento que veamos una máquina de estados es el número de estados que tiene y el número de eventos a los que reacciona.

Y esta es la palabra clave: "reacciona". Una máquina de estados implementada en un programa NO ejecuta nada (no avanza, no hace ningún gasto de CPU) hasta que no se recibe un evento. En ese momento, y según el estado en el que esté, ejecutará una serie de acciones y luego transitará hacia otro estado, o se quedará en el que está. Y, de nuevo, esperará sin hacer nada hasta que se reciba el siguiente evento).

En este ejemplo identificamos dos estados: estadoA y estadoB. No considero el estado fin, porque como tiene el doble circulo, implica que ahí acaba la máquina, y por tanto al llegar ahí se sale de la máquina y ya no se tiene que pensar en qué hacer si llegan nuevos eventos. Utiliza macros de preprocesado para definirlos:

```
#define estadoA 0
#define estadoB 1
```

Puedes poner los valores que quieras, siempre que sean distintos para cada etiqueta de estado. Aquí hemos decidido empezar a numerar por cero y seguir en orden ascendente, una estrategia habitual.

Después identificamos los eventos. Aquí también usaremos macros para definirlos:

```
#define evento0 0
#define evento1 1
```

Y lo mismo aquí, pon el valor que quieras, siempre que no se repita en otros eventos. Nota que la identificación de eventos y estados es independiente, no pasa nada si se repite un valor identificativo de un estado para un evento. No tienen nada que ver.

Ahora vamos a montar la máquina en sí, que puede estar en un main(), o dentro de una función.

```
1  int fin=0;
2  int estado = estadoA; /*estado inicial*/
3  while(!fin){
4  /*bloquea hasta que llegue un evento (podria utilizar un read/select/pause, etc.)*/
5          int evento = espera_evento();
6          /*chequea que el evento es valido*/
7          if(evento <0){
8                  printf("Evento no reconocido\n");
9                  break; /*sale INMEDIATAMENTE de la maquina*/
10         }
11         switch(estado){
12                 case estadoA:
13                         switch(evento){
14                                 case evento0:
15                                         printf("conmutaA->B\n");
16                                         estado = estadoB;
17                                         break;
18                                 case evento1:
19                                         printf("fin\n");
20                                         fin=1;
21                                         break;
22                         }
23                         break;
24                 case estadoB:
25                         switch(evento){
26                                 case evento0:
27                                         printf("fin\n");
28                                         fin=1;
29                                         break;
30                                 case evento1:
31                                         printf("conmutaB->A\n");
32                                         estado = estadoA;
33                                         break;
34                         }
35                         break;
```

```
36                      default: printf("Estado no reconocido\n"); fin=1; break;
37              }
38  }
```

La estructura es sencilla de entender, la máquina se ejecuta dentro de un bucle hasta que se alcance la condición de fin. Aquí hay dos variables principales a considerar, la variable estado almacena (¡qué sorpresa!) el estado actual en el que se encuentra la máquina. Por eso al principio, antes de entrar en el bucle, el estado inicial se asigna a la variable (obviamente, esta información la tenemos mirando el dibujo de la máquina, en el ejemplo el estado inicial es estadoA). La otra variable fundamental es la variable evento, que almacena (¡qué maravilla!) el evento que se haya recibido, y que hará que la máquina deba reaccionar.

En la línea 5 está LA CLAVE de cómo funciona la máquina. Esa función la tenemos que montar nosotros. Ahí es donde se realizará un bloqueo, la espera, hasta obtener uno de los eventos definidos. Más adelante os pongo ejemplos de este invento conveniente para nosotros. Por ahora es más interesante seguir estudiando esta estructura de código.

Una vez que se obtiene un evento, se hace un doble switch. Comenzamos a razonar de esta forma"si estoy en el estado tal y llega el evento pascual entonces haz...". El primer switch sirve para chequear en qué estado estás y, por cada estado, se chequea con otro switch el evento recibido, de forma que se hacen sus acciones correspondientes. En este ejemplo simple no se ha concretado qué hay que hacer ante la llegada de un evento determinado en un estado concreto, y nos limitamos a imprimir un mensaje por pantalla. En otros casos puede que se diga que se active un temporizador, se deshabilite la recepción de alguna señal, se actualice el valor de un contador...

Fíjate que, tras la realización de las acciones asociadas a la llegada de un evento en un estado concreto, puede que haga falta cambiar de estado, y eso SOLO tendrá efecto para la siguiente iteración de la máquina. ¡Es muy importante esto, como puedes ver en las líneas 16 y 32!

NOTA: ¡no te olvides de poner break al terminar cada case del switch porque, si no, sigues ejecutando el siguiente case!

Pues eso es todo, si sigues esta estructura, puedes implementar cualquier especificación visual que te den en formato máquina de estados. Todo esto no es nada rígido, ojo. Puede que no quieras poner una función aparte para hacer el bloqueo y decidas hacerlo directamente dentro del while (pero antes de los switch, claro). También puedes pasarle argumentos a espera_evento(), si los necesitaras. Tú lo vas viendo, lo importante es tener claro el concepto.

Por ejemplo, alguien que no ha entendido cómo funciona una máquina de estados, tampoco va a entender la implementación propuesta, y va a estar tentado a aprenderse aquí cosas de memoria. Eso lleva a la ruina más absoluta. Algunas veces me he encontrado con este disparate (pero nunca entre mis estudiantes, que sois gloria bendita):

```
int estado = estadoA; /*estado inicial*/
int evento = espera_evento();
while(!fin){
        switch(estado){
            /*aqui vienen los case y dentro de cada uno el switch de eventos*/
        }
}
```

La maldición del MS-DOS recaerá sobre ti y dos generaciones de tus descendientes si no esperas un evento cada vez que iteras en la máquina. Esta otra perla también se ha visto por ahí alguna que otra vez:

```
while(!fin){
        int estado = estadoA; /*estado inicial en cada iteracion????????*/
        int evento = espera_evento();
        switch(estado){
              /*aqui vienen los case y dentro de cada uno el switch de eventos*/
        }
}
```

Horroroso. En cada iteración se reinicia el estado al estado inicial. A tomar viento lo que se implementa en el doble switch anidado... ¡No lo hagas nunca así de mal!.

Ejemplos de espera_evento()

Te propongo un primer ejemplo donde se esperan eventos que son señales. Mira:

```
int ctrl_c=0;
int timeout=0;

void manejadora_sigint(int signum){
        ctrl_c = 1;
        signal(signum, manejadora_sigint);
}

void manejadora_sigalrm(int signum){
        timeout = 1;
        signal(signum, manejadora_sigalarm);
}

int espera_evento(){
        pause();
        if(ctrl_c){
                ctrl_c=0;
                return evento0;
        }else if(timeout){
                timeout=0;
                return evento1;
        }else
                return -1; /* ERROR evento no reconocido*/
}

/*asume que en main() se han registrado las manejadoras para las señales
ANTES de entrar en la máquina de estados, obviamente*/
```

La función espera_evento() anterior cumple el requisito de BLOQUEAR al proceso hasta que se reciba un evento. Al desbloquearse, el siguiente requisito es identificar qué es lo que se ha recibido y traducirlo a uno de los eventos válidos que tenemos definidos.

Por ejemplo, en el enunciado del protocolo se puede decir que la llegada de una señal SIGALRM se debe identificar como el evento0, y que la llegada de SIGINT se identifica como el evento1. Cómo veis, la traducción suele ser muy sencilla. También se admite devolver un -1 si hay algún fallo dentro de espera_evento(). En este caso, recibir un evento que no se sabe traducir, genera un valor de retorno de -1, un error.

No sé si habéis caído, pero hay un ERROR GARRAFAL con este espera_evento(). Imagina que estás bloqueado en pause() y que llega un SIGINT, por lo que sales del bloqueo para devolver evento0 y que avance la máquina. Pero resulta que antes de volver a bloquearte en pause(), llega otro SIGINT. Pues bien, cuando finalmente llegas al pause(), ¡te quedas bloqueado hasta que aparezca otra nueva señal! ¿Lo has entendido? ¡El segundo SIGINT se ha perdido! La máquina se ha quedado sin un evento y no funciona correctamente (debería haber terminado al pulsar dos veces Ctrl-C en la consola).

¿Cómo podemos arreglar este problema? Cuando espera_evento() depende de señales, lo mejor es encolarlas para que no se pierda nada. Y te propongo que eso lo hagas con una pipe. ¿Cómo con una pipe? ¿Pero eso no se utiliza para comunicar padres e hijos? Sí, pero no solo para eso. Mira la solución, a ver si te convence:

```
int fd_pipe[2];

void manejadora_sigint(int signum){
	uint8_t evento = evento0;
	writen(fd_pipe[1], &evento, 1);
	/*chequea que ha funcionado*/
	signal(signum, manejadora_sigint);
}

void manejadora_sigalrm(int signum){
	uint8_t evento = evento1;
	writen(fd_pipe[1], &evento, 1);
	/*chequea que ha funcionado*/
	signal(signum, manejadora_sigalarm);
}

int espera_evento(){
	uint8_t evento;
	int leidos = readn(fd_pipe[0], &byte, 1);
	if(leidos < 0)
		return -1;
	else
		return evento;
}

int main(){
	int resultado = pipe(fd_pipe);
	if(resultado < 0){
		perror("Error en pipe");
		exit(1);
	}
	signal(SIGINT, manejadora_sigint);
```

```
        signal(SIGARLM, manejadora_sigalrm);

        int fin = 0;
        int estado = estadoA; /*estado inicial*/
        while(!fin){
                int evento = espera_evento();
                /*la implementacion de la maquina sigue como antes*/

        /*...*/
        }
        return 0;
}
```

Fíjate qué curioso. El array fd_pipe[2], es una variable global, porque los manejadores escriben en la tubería y espera_evento() lee de la tubería. Así hemos conseguido que todos los eventos se encolen en orden y no se pierdan.

Por supuesto, la creación de la pipe hay que hacerla en el main(), y lo tienes que hacer ANTES de registrar las manejadoras, no vaya a ser que se intenten utilizar los descriptores de la tubería antes de que esté creada (las casillas del array tiene basura antes de hacer la llamada al sistema pipe()).

NOTA: Este diseño con la pipe es un poco cutre salchichero, y no sirve para dejarlo así en un programa profesional. La verdad es que a nosotros sí que nos sirve perfectamente para practicar con las máquinas de estado en nuestros ejercicios, así que seguimos adelante con esta idea. No leas más si no quieres, pero aquí dejo alguna pincelada de por qué el diseño con la pipe sigue siendo regular, y posibles soluciones (fuera del alcance de nuestra asignatura):

Chapuza 1: si la pipe se llenara de eventos, puede que te quedes bloqueado en el writen, y el proceso se queda así para siempre. ¡No hay otro proceso diferente que lea y deje sitio en la tubería, ya que eres tú mismo el que lo debería hacer, pero no puedes por estar bloqueado! Solución: puedes decirle al kernel que, en vez de enviarle las señales a tu proceso, las vaya guardando en una cola especial suya. Tú puedes acceder a esa cola a través de un descriptor que se te proporciona, y con un read() normal, puedes ir leyendo las señales que haya en dicha cola. Para obtener el descriptor, necesitar utilizar una llamada al sistema que se llama signalfd() (man 2 signalfd).

Chapuza 2: siempre hemos dicho que lo primero que hay que hacer en el programa son las llamadas a signal(), para asegurarnos rápidamente de que, cuando lleguen las señales, se manejarán y no se terminará el programa (que es lo que pasa, por defecto, en la mayoría de casos). Pero aquí tenemos que hacer primero la llamada a pipe() para crear la tubería. Eso hace que haya más riesgo de que lleguen señales en ese tiempo y el programa termine. Solución: existe una forma de decirle al kernel que, en una zona de código, bloquee la llegada de señales. El mecanismo se llama sigprocmask() (man 2 sigprocmask) y, básicamente, se utiliza ejecutándolo para esto: "a partir de ahora bloquea la llegada de la señal tal y la pascual". Tras esta operación, ya puedes ejecutar pipe(). Mientras, si aparece alguna señal, el kernel la encola (no se pierden). Cuando estés listo, ejecutas de nuevo sigprocmask() para esto: "a partir de ahora desbloquea la llegada de la señal tal y la pascual". Como el mundo es muy pequeño, hay un ejemplo de cómo se usa sigprocmask() en la página del manual para signalfd() ¡porque se utilizan juntas!

Te pongo ahora otro ejemplo de espera_evento(), para que veas que se puede ir un poco más allá. En este caso, vamos a esperar que llegue un mensaje con una cadena de texto y esto es el evento0. También podría llegar una señal (por ejemplo SIGALRM, por un temporizador), interrumpiendo la lectura del canal, y entonces lo interpretamos como evento1. A ver qué te parece esto:

```
int espera_evento(int fd, char *buffer, size_t max_a_leer){
        alarm(5); /*activa la cuenta atras, pensado para compilar en modo System V*/
        int leidos = read(fd, buffer, max_a_leer);
        alarm(0); /*cancela la alarma*/
        if(leidos < 0){
                if(errno == EINTR)
                        return evento1;
                else
                        return -1; /*error*/
        }else if(leidos == 0){
                return -1;
                /*error (en este caso se ha decidido que
                  un cierre de canal es un error, pero podría ser un
                  evento en otro ejemplo)*/
        }else{
                buffer[leidos] = '\0'; /*lo convierto en cadena*/
                return evento0;
        }
}

int main(){
        signal(SIGARLM, manejadora_sigalrm); /*supon que existe la manejadora*/
        int fd = open("/tmp/fsc_fifo", O_RDONLY); /*para ser lector de una fifo*/
        /*chequea si ha ido bien*/
        char buffer[512];
        int fin=0;
        int estado = estadoA; /*estado inicial*/
        while(!fin){
          int evento = espera_evento(fd,buffer,512-1); /*ojo, deja espacio para el '\0'*/
                /*ahora viene el doble switch estado-evento
                  ¡y puedes utilizar el mensaje que hay en buffer para inspeccionarlo y
                  tomar decisiones, enviarlo por un canal, guardarlo en un fichero,
                  imprimirlo por pantalla...
                */
        }
}
```

Es la flexibilidad a la que me refería antes, espera_evento() recibe argumentos, y devuelve el mensaje en caso de que se reciba, para que luego se pueda manipular en la máquina de estados.

Te he puesto este ejemplo a posta para que hagamos una reflexión:

Supón que tu máquina de estados es parte de un protocolo que hace de retransmisor: mensaje que le llega por un canal, mensaje que retransmite por otro. Eso se especificaría así:

"Si la máquina está en el estadoA y le llega un evento0 (que representa la llegada de un mensaje por el canal de entrada), debe enviarlo por el canal de salida, y luego pasar al estadoB".

¿Dónde harías ese envío del mensaje?

Si estás pensando en hacerlo directamente en espera_evento() es que no has entendido bien cómo funciona una máquina de estados (perdón por soltártelo así). Recuerda que cualquier acción que realice la máquina al reaccionar a un evento SIEMPRE, SIEMPRE, ha de hacerse en el "case" del evento recibido, para cada estado concreto. ¡En esa zona, y solo en esa!

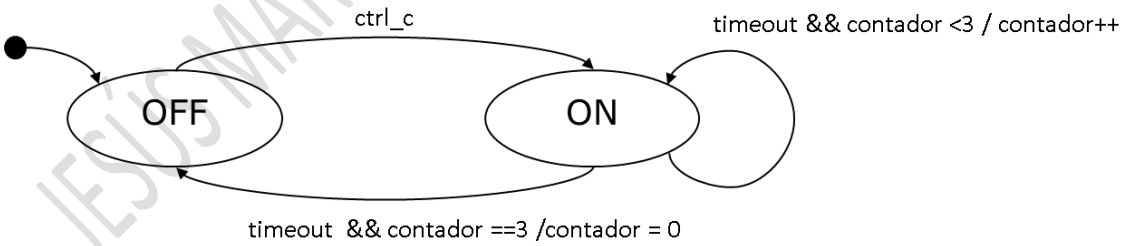
```
switch(estado){
        case estadoA:  switch(evento){
                                case evento0:
                                        int escritos = writen(1,buffer,strlen(buffer));
                                        /*chequea que ha ido bien*/
                                        estado = estadoB;
                                        break;
/*...*/
```

Ni se te ocurra tampoco cambiar de estado dentro de espera_evento(). Esa acción debe quedar exclusivamente en la parte de acciones del "case" correspondiente al evento recibido para el estado actual.

Si te saltas estas reglas, te habrás metido en un lío de cuidado, y es muy posible que tu programa termine siendo un ejemplo lamentable de código espagueti[1], imposible de entender y mantener y, por supuesto, que seas la vergüenza de tu familia.

SUBIENDO EL NIVEL: MAQUINA DE ESTADOS CON TEMPORIZADORES

Mira esta otra máquina, con alguna que otra variante interesante:



El enunciado es este: "una máquina se encuentra continuamente en estado OFF hasta que le llega una señal SIGINT (evento ctrl_c) y pasa al estado ON. Allí permanece hasta que llegan tres señales SIGALRM, con un intervalo de un segundo entre ellas. Después, regresa al estado OFF. Durante la estancia en el estado ON, se deshabilita la llegada de señales SIGINT".

Lo tenemos todo para empezar a trabajar. Lo primero es identificar estados: OFF y ON. Después, los eventos: ctrl_c y timeout (llegada de SIGINT y SIGALRM).

```
#define OFF 0
#define ON 1
```

1

```
#define ctrl_c 0
#define timeout 1
```

¿Y qué es ese contador que aparece en el dibujo? Es una variable de la máquina, una condición cuyo valor hay que tener en cuenta para continuar en el mismo estado o cambiar. Es típico incluir esta información extra cuando con el evento en sí (timeout en este caso) no es suficiente para saber si permaneces en ON o cambias a OFF ¡Pero no es un evento, es una condición! No te va a llegar en la función espera_evento().

A veces, también te puedes encontrar que aparece una barra / después del evento (+condición). Si es así, contiene información de las acciones a realizar cuando llega dicho evento. Esta acción o acciones no tienen por qué estar completas en el dibujo, lo importante es lo que diga la especificación o enunciado de la máquina, pero se ponen ahí por ser suficientemente representativas para entender de un vistazo que son muy importantes.

Claro, hay más acciones que hacer y que no aparecen en el dibujo: hay que activar un temporizador periódico **antes de pasar a ON**, y también habrá que desactivarlo al pasar de nuevo a OFF.

En este ejemplo, tampoco hay estado final. Vale, puede que la máquina esté dando servicio continuamente, eso no nos molesta a nosotros.

Os pongo aquí la implementación, sin entrar en detalles de espera_evento(), que podemos resolver como en el apartado anterior, usando una pipe. Fíjate como tienes que activar el temporizador ANTES de cambiar al estado en el que se van a esperar los eventos de tipo SIGALRM (porque si no, no vas a recibir eventos de timeout en la máquina). Y tienes que desactivarlo ANTES de pasar al estado OFF. Lo mismo te digo con el registro de la señal manejadora de SIGINT, tienes que hacerlo ANTES de volver al estado OFF porque, si no, en la vida vas a volver a recibir la señal.

```
signal(SIGALRM,manejadora_sigalrm);
signal(SIGINT, manejadora_sigint);

int fin = 0;
int contador = 0;
int estado = OFF; /*estado inicial*/
while(!fin){
        /*bloquea hasta que llegue un evento (podria utilizar un read/select/pause, etc.)*/
        int evento = espera_evento();
        switch(estado){
                case OFF:
                        switch(evento){
                                case ctrl_c:
                                        struct timeval uno = {1,0};
                                        struct itimerval timer = {uno, uno};
                                        /*activa temporizador*/
                                        setitimer(ITIMER_REAL, &timer, NULL);
                                        signal(SIGINT,SIG_IGN); /*desactiva llegada de Ctrl-C*/
                                        printf("OFF->ON\n");
                                        estado = ON;
                                        break;
                                case timeout:
```

```
                                        printf("algo va mal, no lo esperaba aqui\n");
                                        fin=1;
                                        break;
                                }
                                break;
                        case ON:
                                switch(evento){
                                        case ctrl_c:
                                                printf("algo va mal, no lo esperaba aqui\n");
                                                fin=1;
                                                break;
                                        case timeout:
                                                contador++;
                                                if(contador == 3){
                                                        struct timeval cero = {0,0};
                                                        struct itimerval timer = {cero, cero};
                                                        /*desactiva temporizador*/
                                                        setitimer(ITIMER_REAL, &timer, NULL);
                                                        contador = 0; /*pone el contador a cero*/
                                                        signal(SIGINT,manejadora_sigint);
                                                        printf("ON->OFF\n");
                                                        estado = OFF;
                                                }
                                                break;
                                }
                                break;
        }
}
```

Pues ya estamos listos para hacer cualquier implementación que nos pongan por delante. Os recomiendo que hagáis algún ejercicio de examen interesante (zona de exámenes del campus) para afianzar aún más estos conceptos. Por ejemplo:

Septiembre 2016 (¡hay que hacer un filtro de control de groserías (tacos) manipulando cadenas!)

Junio 2018 (¡hay que utilizar select en espera evento!)

¡Al ataque!