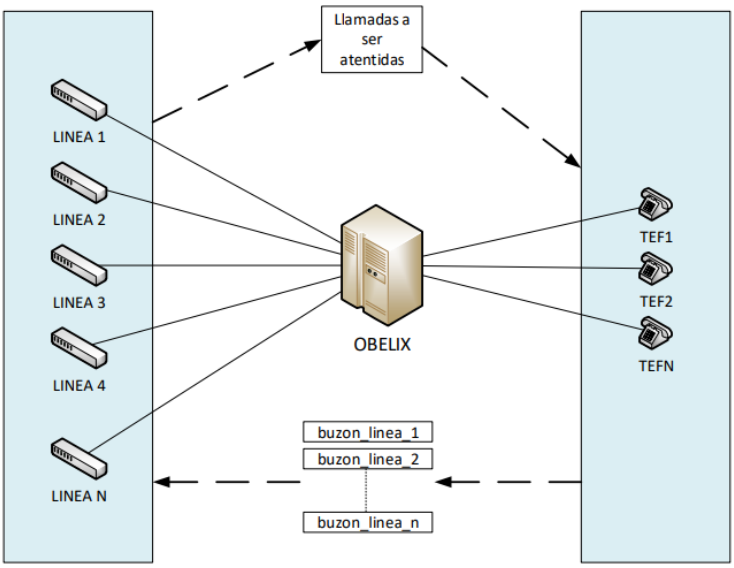
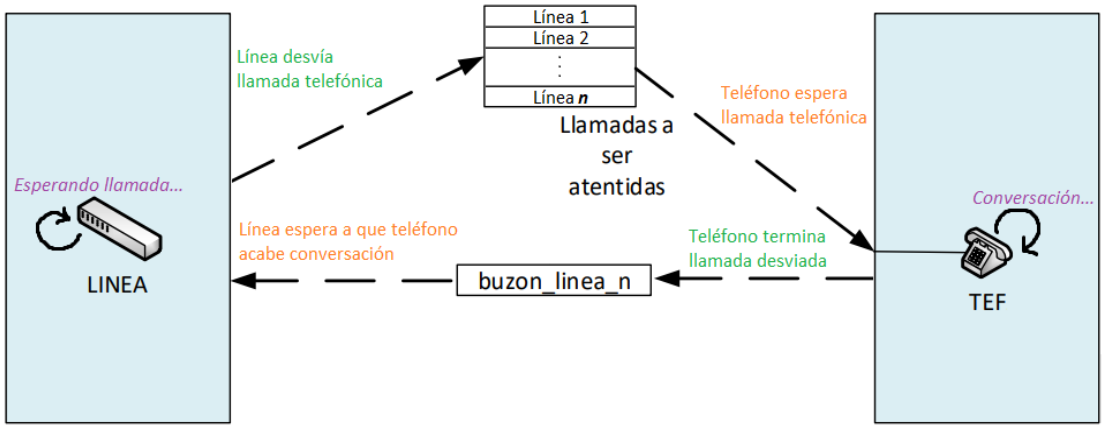
Planteamiento

A pesar de que esta práctica es muy similar a la anterior, pues seguimos contando con nuestro ecosistema de llamadas telefónicas dirigido por la central Obelix, debemos hacer un replanteamiento de la implementación, ya que en este caso contamos con las colas de mensajes del estándar POSIX, y el paso de mensajes como método de sincronización.

Entonces, el funcionamiento primitivo de nuestro programa variará, por otro mucho más intuitivo:



En este caso contaremos con dos tipos de buzones, el buzón donde se alojarán las llamadas a ser atendidas, y el buzón de una línea telefónica, con su respectiva llamada en curso. De este último tipo de buzón habrá en total ***n*** buzones. **Cabe destacar que, debido a limitaciones técnicas, el número máximo de líneas tendrá que ser n = 10**. Esto se debe a que el número máximo de mensajes que puede alojar una cola de mensajes es de 10 elementos (definido en Linux en la ruta `/proc/sys/fs/mqueue/msg_max`), con un tamaño de mensaje de, en nuestra práctica, 64 bytes. Por otro lado, **seguiremos empleando 3 teléfonos**:



La sincronización que se plantea en este problema es la siguiente:

1. Tras un tiempo de espera de llegada de una llamada (SIMULADO), la línea desvía la llamada entrante a un teléfono. Para ello, Envía un mensaje al buzón de llamadas que atenderá un teléfono, y esperará a que termine la conversación para seguir con su funcionamiento, bloqueándose el proceso hasta entonces.

2. El teléfono estará a la espera de que le desvíen una llamada telefónica, por lo que el proceso estará bloqueado hasta que no haya una línea en el buzón de llamadas con una llamada entrante. En el momento en el que haya una disponible, el teléfono atiende la llamada y mantiene una conversación (SIMULADO). Cuando acaba, envía un mensaje a la línea, avisando de que la conversación terminó, liberando la línea.

Como podemos notar, existe un patrón de bloqueos y desbloqueos de procesos línea y teléfono de manera constante, propio de una existente sincronización gracias a las colas de mensajes que se emplean. De hecho, para poder sincronizar a la perfección ambos procesos, **debe enviarse y recibirse el nombre del buzón de la línea llamante y la conversación**. Entonces, podemos interpretar así el funcionamiento primitivo de los procesos correspondientes a la línea y el teléfono:

Proceso Línea

```
while (1){
/*Esperando llegada llamada…*/
   sleep(1 a 30s);
/*Desvío llamada telefónica…*/
   send(BUZON_LLAMADAS,buzon_linea);
/*Esperando finalización…*/
   receive(MI_LINEA,conversacion);
}
```

Proceso Teléfono

```
while (1){
/*Esperando llamada entrante…*/
   receive(BUZON_LLAMADAS,buzon_linea);
/*Entablando conversación…*/
   sleep(10 a 20s);
/*Finalización llamada…*/
   send(MI_LINEA,conversacion);
}
```

Es, a partir de aquí, cuando comenzamos a construir nuestro programa correspondiente a la práctica 3 de laboratorio.

Headers

Para el funcionamiento completo del programa, se han considerado un solo header o interfaz, además de las ya existentes pertenecientes a la librería estándar ANSI C. Esta es: “definitions.h”. Con respecto a la práctica anterior, hay presentes numerosos cambios.

En este header incluimos en primer lugar una librería de tipos de sistema que define una colección de símbolos y estructuras typedef. También se definen las rutas de los ejecutables de los procesos “telefono” y “linea” junto con su tipo de proceso (CLASE). Se definen el número de lineas y telefonos, el nombre de los buzones que vamos a emplear y los tamaños máximos en bytes de cada mensaje. Por último, definimos una cadena de caracteres que nos indica el fin de la conversación y una estructura de TProceso compuesta por un pid y un tipo de proceso o clase.

Manager.c

En esta clase es donde se va a ejecutar todo lo necesario para obtener el resultado deseado del programa, es decir, contiene desde la creación de los buzones hasta las llamadas a las clases linea.c y telefono.c, pasando por la instalación de la señal de interrupción (Control+C) y la inicialización de las correspondientes tablas de procesos. Finalmente, el programa terminará mediante la señal de interrupción instalada, donde se procederá a terminar los procesos existente y liberar los recursos cargados en memoria.

En este punto vamos a indagar concretamente en aquellos métodos que han supuesto cierta dificultad una vez comenzada la fase de programación, los cuales son: *`crear_buzones()`* y *`lanzar_proceso_linea()`*.

- **crear_buzones()**. Este método abre las colas de mensajes que se emplearán para la ejecución y correcto funcionamiento del programa. En primer lugar, declaramos una serie de variables que usaremos durante la creación de los buzones, como son un iterador para las colas de mensaje, dos estructuras de atributos `mqAttrA y mqAttrB` y una cadena de caracteres auxiliar con un tamaño igual a `TAMANO_MENSAJES`. Una vez creadas, procedemos a crear un buzón de llamadas y una serie de buzones linea.

- Buzón de llamadas: Antes de abrir su cola de mensajes, vamos a necesitar asignar a la estructura de atributos `mqAttrA` un número máximo de mensajes, el cual va a ser el contenido de `NUMLINEAS` que equivale a 10, y un tamaño máximo de los mensajes que va a ser `TAMANO_MENSAJES`.
  Tras esto vamos a comprobar que al abrir la cola de mensajes para qHandlerLlamadas empleando mq_open no sucede ningún problema, donde en caso afirmativo lanzamos un aviso, liberamos los recursos y finalizamos con código de error.
- Buzones linea: Para abrir la cola de mensajes vamos a proceder de forma similar al buzón de llamadas, solo que en la estructura de atributos `mqAttrB` asignamos el número máximo de mensajes a 1, ya que en realidad se van a crear tantos buzones como valor tenga `NUMLINEAS`. Una vez hecho esto, mediante un bucle `for` iremos comprobando si cada cola de mensaje se abre correctamente, donde en caso contrario se procede de la misma forma que en el buzón de llamadas, lanzamos aviso, liberamos recursos y finalizamos el programa con código de error.

- **lanzar_proceso_linea()**. Para lanzar un proceso linea necesitaremos declarar un pid de proceso y una cadena de caracteres con un tamaño correspondiente a `TAMANO_MENSAJES` llamada `nombre_completo_buzon`. Tras declarar ambas variables vamos a insertar un `switch` donde se dan dos casos al crear un proceso a partir del "pid":
  - `case -1`: trata cuando se ha producido un **error** al crear el proceso. Aquí lanzamos un aviso y procedemos a llamar a los métodos *terminar_procesos()* y *liberar_recursos()* y finaliza el programa con código de error.
  - `case 0`: aquí se da el caso de que sí se haya **creado correctamente** el proceso. Entonces, llamamos a ejecutar a la clase linea pasándole su ruta, tipo de proceso, el nombre completo del buzón y un parámetro `NULL`, para evitar errores, dentro de la condición de un if para comprobar si la ejecución se ha realizado con éxito. En caso de error, es decir, que la ejecución devuelva -1, imprime en pantalla un mensaje de error y finaliza el programa con código de error.

  Tras esto, una vez fuera de la sentencia del `switch`, se procede a almacenar en la tabla de procesos de linea (g_process_linea_table) el "pid" y la clase (su tipo de proceso).

Linea.c

El proceso línea, tal y como sucede en la práctica anterior, se compone de un único módulo, el módulo principal. Está estructurado en dos partes: en la declaración de variables locales y en el bucle infinito de funcionamiento.

En la declaración de variables podemos encontrar un entero "pid", que, como indica su nombre, almacena el pid del proceso. También se declaran dos objetos descriptor de colas de mensajes POSIX. Para emplearlas, requerimos de los headers `<mqueue.h>` y nuestro `<definitions.h>`. Entonces, declaramos dos, uno para las llamadas a ser atendidas (`qHandlerLlamadas`) y otro para la línea en cuestión (`qHandlerLinea`, 1 de ***n***). Por último, tenemos dos búferes, uno para el nombre del buzón de la línea y otro que simulará ser la conversación. Ambos serán del mismo tamaño del tamaño correspondiente a las colas de mensajes: `TAMANO_MENSAJES`.

Entonces, se establece la semilla de números aleatorios del proceso línea. A partir de aquí, la línea comprobará que el número de argumentos sea correcto y que se escribe bien el nombre del buzón de la línea en `buzonLinea`. Entonces, abrimos las colas de mensajes en modo lectura escritura. Para el descriptor de llamadas abrimos el `BUZON_LLAMADAS`, y para línea el nombre del buzón de la línea, el cual recordemos que llegó como argumento al proceso.

A partir de aquí se ejecutará el bucle infinito, denotado en ANSI C como `while(1)`:

1. Se simula esperar una llamada, paralizando el proceso un número aleatorio de entre 1 y 30 segundos.

2. Informa de que se ha recibido una llamada, entonces, se envía un mensaje a la cola de mensajes de las llamadas a ser atendidas, utilizando el descriptor que tiene abierto `BUZON_LLAMADAS`, y le pasamos la cadena `buzonLinea` (nombre del buzón), con su tamaño exacto y prioridad 0.
3. Espera a que se reciba un mensaje en el buzón de la línea (denotado con el descriptor de la línea, que tiene abierto `buzonLinea`). Tiene que llegar el mensaje al buffer, que simula ser la conversación, con su tamaño exacto. Establecemos la prioridad con NULL; no es relevante.

Telefono.c

A diferencia de la práctica anterior, el proceso teléfono se compone únicamente del módulo principal. Como en proceso linea, el modulo del proceso telefono está estructurado en dos partes: declaración de variables y bucle infinito de funcionamiento del programa.

En la declaración de variables nos encontramos con una variable entera "pid", la cual va a almacenar el `pid` del proceso. También tenemos dos búferes, uno para el nombre del buzón y otro que simulará ser la conversación. Ambos presentarán el mismo tamaño, que será el correspondiente a las colas de mensajes: `TAMANO_MENSAJES`.

Así pues, se establece la semilla de números aleatorios del proceso telefono con el pid de proceso y se crea y abre una cola de mensajes en modo de lectura y escritura con descriptor `BUZON_LLAMADAS`.

A partir de aquí se ejecuta el bucle infinito `while(1)`:

1. Bloqueamos el telefono hasta que le desvíe una llamada, es decir, hasta que le llegue un mensaje desde `mq_receive()`.
2. El telefono recibe un mensaje proveniente de la cola de mensajes de las llamada donde encontramos la cadena buzonLinea con su tamaño y prioridad `NULL` e informa de ello. Aquí el proceso se paraliza un número aleatorio de entre 10 y 20 segundos.
3. Abrimos la cola de mensajes correspondiente al nombre del buzón de la línea a atender en modo lectura y escritura. El resultado lo guardamos en dicho descriptor (qHandlerLinea). El proceso informa de que la llamada ha sido finalizada y envía un mensaje a la cola de mensaje de la linea con `buffer` como descriptor de archivo, su tamaño exacto y prioridad 0.
4. Cerramos la cola de mensajes para linea abierta.

Compilación y ejecución con Makefile

Una vez hemos cesado de programar las clases, vamos a comprobar que pueden ejecutarse sin impedimentos. Para ello usaremos un archivo MAKEFILE. Los archivos MAKEFILE contienen reglas que especifican las acciones a realizar por el comando *make* y directivas que indican a éste algunas formas especiales de interpretación del contenido del archivo MAKEFILE.

El comando *make* funciona basándose en las reglas especificadas en el archivo MAKEFILE. En esas reglas se expresa, para cada archivo que deba construirse, de qué otros archivos dependen.

En este caso, cuando escribamos *make* por terminal, nos va a crear directorios nuevos:

- obj/ Aquí se alojarán los archivos objeto (extensión .o), necesarios para la compilación
- exec/ En este directorio se guardan los ejecutables resultantes de la compilación

En la ejecución de make, se crearán 3 archivos objeto: manager.o, telefono.o y linea.o, con opción para depuración (ggdb), usando como includes la carpeta include del proyecto (por eso creamos archivo objeto de la lista) y visualización de todas las advertencias (Wall), y 3 ejecutables: *manager*, *telefono* y *linea*.

Por último, para ejecutar el resultado de hacer make empleamos `./<directorio_ejecutable> <argumento1> <argumento2>`. En el caso de este programa debemos escribir en la línea de comandos la siguiente sentencia de ejecución: `./exec/manager`

En el archivo MAKEFILE también hay definida una última regla de borrado, denominada clean, para borrar las carpetas creadas anteriormente, incluyendo su contenido. Para llevarlo a cabo, sólo debemos escribir en nuestro espacio de trabajo donde esté MAKEFILE el comando *make clean*.