❌ Minitalk ✅

Objetivo del proyecto:

En este ejercicio deberemos crear un programa llamado **server** que nos muestre su PID. Después uno llamado **client** al que le daremos la PID del servidor como argumento y un texto entre comillas. El servidor debe mostrar el texto facilitado. En el bonus debe soportar caracteres ASCII extendidos y el cliente debe mostrar un mensaje cuando el servidor haya mostrado su mensaje.

El problema es que sólo podemos usar las señales SIGUSR1 y SIGUSR2 que son señales definidas por el usuario. La única manera de trasladar el código ASCII al servidor con sólo 2 señales es hacerlo en binario.

Ejecución:

Ejecutamos el comando make y tendremos que tener confirmación del estado **ok de la librería, del server y del client.** Ejecutamos **./server** tal como nos indican las instrucciones. Antes de ejecutar el cliente veamos qué va ha hecho er server hasta ahora.

Analicemos el código del **server** por partes:

```
int main(void)
{
  int         pid;
  struct sigaction  data; /* Se define la estructura de sigaction para recibir
                             una señal y saber de quien la recibe. */

  pid = getpid(); /* Esta variable tendrá el PID del servidor, el cual debemos
                  introducir en el cliente para que llegue la señal. */
  ft_printf("Ejecuta el cliente en otra ventana con el comando ./client");
  ft_printf(" seguido del PID: %d y el mensaje entre comillas", pid); /* Aquí
                  se nos muestra el PID del servidor. */
  ft_printf(" que quieras que muestre el servidor\n");
  ft_printf("Ejemplo: ./client %d \"hola\"\n", pid); /* Ejemplo de los argumentos
                  a introducir en el cliente. */
  data.sa_flags = SA_SIGINFO; /* Con esta variable sabemos quien nos manda la
                  señal. */
  data.sa_sigaction = signal_control; /* Con esta variable ejecutamos la
                  respuesta a esta señal con la función signal_control. */
  sigaction(SIGUSR1, &data, NULL); /* En espera de recibir SIGUSR1 */
  sigaction(SIGUSR2, &data, NULL); /* En espera de recibir SIGUSR2 */
  while (1)
    pause();
}
```

Ahora sí ejecutamos **./client PID “mensaje”** en una ventana nueva como nos indican las instrucciones y veamos que va haciendo el client:

```
int main(int argnum, char **arg)
{
  size_t  i;
  char  *str;

  if (argnum != 3 || !ft_strlen(arg[2])) /* Si hay argumentos de más o de menos
                                        o algún argumento vacío. */
    return (ft_printf("Te faltan o te sobran argumentos."));
  signal(SIGUSR1, signal_control); /* Esperando a que llegue la señal SIGUSR1 y
                                ejecutarle la función signal_control. */
  signal(SIGUSR2, signal_control); /* Esperando a que llegue la señal SIGUSR2 y
                                ejecutarle la función signal_control. */
/* En este caso usamos signal y no sigaction porque no necesitamos saber quien
nos la envía. */
  i = 0;
  str = arg[2];
  while (ft_strlen(str) >= i) /* Este bucle está hecho así para que envíe
                              también el carácter '\0' */
  {
    send_signal(arg[2][i], ft_atoi(arg[1])); /* Ejecutamos la función send_signal
                              caracter por carácter al PID que lo convertimos en int */
    i++;
  }
  while (1)
    pause();
}
```

Veamos qué hace la función **send_signal en el client:**

```
/* Aquí está la manera de mandar los caracteres sólo con 2 señales, el SIGUSR1
es el 0 y el SIGUSR2 va a ser 1. Así un caracter lo enviaremos en modo binario.
ejemplo 'A' == 65 (en decimal) == 01100001 (en binario) */
void  send_signal(char c, int pid)
{
  int i;
  int bit;

  i = 7;
  while (i >= 0)
  {
    g_signal_ready = 0; /* Variable global para gestionar los tiempos de espera. */
    bit = (c >> i) & 1; /* Bits desplazados "i" veces para capturar 1 u 0. */
    if (bit == 0)
      kill(pid, SIGUSR1); /* Manda 0 a la PID. */
    else
      kill(pid, SIGUSR2); /* Manda 1 a la PID. */
    while (g_signal_ready == 0) /* Hasta que el server no tenga la señal no
                              cambiará a 1 y se queda en espera. */
      usleep(1);
    i--;
```

```
  }
}
```

Volvamos al **server** a ver que hace con ese número binario que le ha llegado. Recordemos que en el main del server nos quedamos esperando unas señales en estas líneas de código:

```
sigaction(SIGUSR1, &data, NULL);
sigaction(SIGUSR2, &data, NULL);
```

Dentro del “&data” viene definido que se usará la función signal_control (que la explicamos a continuación) y los datos de quien la envía para volver a enviarle datos.

```
/* Tomamos de base que los 8 números de binario son ceros, c = 0, así que sólo
hay que cambiar si nos llega la SIGUSR2 que es un 1. */
void  signal_control(int signal, siginfo_t *info, void *param)
{
  static char c = 0;
  static int  bit = 7;
  static int  pid = 0;

  (void)param; /* Esto se usa para que no ponga pegas al compilar aunque
               realmente no haga nada. */
  if (signal == SIGUSR2) /* Si la señal que recibimos es un 1 (SIGUSR2) pues
                         desplazamos el uno bit veces. */
    c = c | (1 << bit);
  if (pid == 0) /* Obtenemos la PID del cliente para mandarle una señal. */
    pid = info->si_pid;
  kill(pid, SIGUSR2); /* Mandamos la señal. */
  bit--;
/* Dejamos de leer aquí para volver a saltar al cliente y ver como reacciona
al mandarle la SIGUSR2. */
  if (bit < 0)
  {
    bit = 7;
    if (c == '\0')
    {
      c = 0;
      kill(pid, SIGUSR1);
      ft_printf("\n");
      pid = 0;
      return ;
    }
    ft_printf("%c", c);
    c = 0;
  }
}
```

El servidor ha recibido 0 o 1 , lo ha puesto en su orden y manda al cliente la señal SIGUSR2, veamos qué hace ahora el **cliente.**

Con este código está esperando las señales:

```
signal(SIGUSR1, signal_control);
signal(SIGUSR2, signal_control);
```

Se ejecuta la función signal_control en el **cliente** cuando recibe la señal SIGUSR2.

```
void  signal_control(int signal)
{
  if (signal == SIGUSR2) /* Nos llega SIGUSR2. */
  {
    g_signal_ready = 1; /* Cambiamos el valor de esta variable global. */
  }
  else
    exit(ft_printf("El servidor ha recibido el mensaje.\n"));
}
```

Anteriormente en el **cliente**, la función send_signal se quedó en pausa por este bucle:

```
while (g_signal_ready == 0)
      usleep(1);
```

Al cambiar a 1 la variable global g_signal_ready ya salimos del bucle y continuamos. Pongo la función send_signal del **cliente** para poder comprenderla mejor:

```
void  send_signal(char c, int pid)
{
  int i;
  int bit;

  i = 7;
  while (i >= 0)
  {
    g_signal_ready = 0;
    bit = (c >> i) & 1;
    if (bit == 0)
      kill(pid, SIGUSR1);
    else
      kill(pid, SIGUSR2);
    while (g_signal_ready == 0)
      usleep(1);
    i--;
```

```
  }
}
```

¿Qué ocurre cuando mandamos el caracter ‘\0’ que nos indica el fin de la string?. Pues veamos qué hace el **server:**

```
/* El caracter '\0' en binario es 00000000, así que una vez le llega el
último 0 saltamos al siguiente comentario. */
void  signal_control(int signal, siginfo_t *info, void *param)
{
  static char c = 0;
  static int  bit = 7;
  static int  pid = 0;

  (void)param;
  if (signal == SIGUSR2)
    c = c | (1 << bit);
  if (pid == 0)
    pid = info->si_pid;
  kill(pid, SIGUSR2);
  bit--;
  if (bit < 0)
  {
    bit = 7;
    if (c == '\0') /* Aquí detecta que el caracter es '\0' y manda la SIGUSR1. */
    {
      c = 0;
      kill(pid, SIGUSR1);
      ft_printf("\n");
      pid = 0;
      return ;
    }
    ft_printf("%c", c);
    c = 0;
  }
}
```

Volvemos a saltar al **cliente** a ver qué hace cuando recibe la SIGUSR1.

```
/* Al recibir SIGUSR1 imprime el mensaje que el servidor ya ha terminado. */
void  signal_control(int signal)
{
  if (signal == SIGUSR2)
  {
    g_signal_ready = 1;
  }
  else
    exit(ft_printf("El servidor ha recibido el mensaje.\n"));
}
```