



# **Universidad de Mendoza - Computacion 2**

Guía de ejercicios de  
laboratorio

## Temario

Consideraciones:	3
Ejercicio 1 - Getopt	4
Ejercicio 2 - Getopt	4
Ejercicio 3 - Fork	4
Ejercicio 4 - Fork	4
Ejercicio 5 - Fork	4
Ejercicio 6 - Señales	5
Ejercicio 7 - Señales	5
Ejercicio 8 - Señales	5
Ejercicio 9 - Señales	5
Ejercicio 10 - Señales / Pipes	5
Ejercicio 11 - Pipes	6
Ejercicio 12 - FIFO / pipes	6
Ejercicio 13 - Multiprocessing	7
Ejercicio 14 - Multiprocessing / Pipe	7
Ejercicio 15 - Multiprocessing / Pipe / FIFO	7
Ejercicio 16 - Mini Chat	7
Ejercicio 17 - Fork / MQ	8
Ejercicio 18 - Multiprocessing / MQ	8
Ejercicio 19 - Pool of Workers / MQ	8
Ejercicio 20 - Pool of workers	8
Ejercicio 21 - Threading / Pipe / FIFO	8
Ejercicio 22 - Threading	9
Ejercicio 23 - Threading / MQ	9
Ejercicio 24 - Sockets INET/DGRAM	9
Ejercicio 25 - Sockets INET/STREAM	9
Ejercicio 26 - Sock Inet Stream / fork	10
Ejercicio 27 - Sock Inet Str / multiprocessing	10
Ejercicio 28 - Sock Inet Str / threading	10
Ejercicio A (opcional) - Stress Test	10
Ejercicio 29 - Sock Inet Str / threading	11
Ejercicio 30 - Lock (multiprocessing)	11
Ejercicio 31 - Lock (threading)	12
Ejercicio 32 - ProcessPoolExecutor	12
Ejercicio 33 - ThreadPoolExecutor	12
Ejercicio B (Opcional) - Performance	12
Ejercicio C (opcional) - (minichat improved)	13

### Consideraciones:

Los ejercicios de la presente guía práctica deben resolverse siguiendo estos lineamientos generales:

- **Copiar y pegar código:** Si bien reutilizar código está bien, copiar un fragmento de código solamente "porque en X ejercicio funcionaba", sin saber qué es lo que hace ese código, no está bien. Deben poder **justificar** el uso de cada herramienta. Esto también es válido para cuando copian código de Internet, que saquen algo de *stackoverflow* no me sirve si no saben qué es lo que hace.
- **Código inútil:** Un poco derivado del anterior, me he encontrado con que a veces copian un código completo y lo modifican para resolver otra tarea. Si hacen eso tengan en cuenta eliminar todo el código que no usen en el original. Por ejemplo, encontrarme una práctica que resuelve, por decir algo, un problema usando sockets, y dentro del código define pipes y fifos que nunca utilizan. dejar ese código me da a pensar que no saben para qué sirven esas líneas, y las dejaron "*por las dudas, porque así funciona*".
- **Mezclar código:** Si bien en Python se pueden escribir funciones intercaladas con el código principal, no es una buena práctica. Por favor definan todas las funciones que van a usar en la parte superior, o como módulos externos al programa, y luego hagan las correspondientes llamadas. Recuerden siempre utilizar una función como `main()` verificando el contenido de la variable de entorno `__name__`.
- **Código redundante:** Si una sección de código va a utilizarse varias veces, no copien y peguen esas líneas! Utilicen funciones!! Para eso existen.
- **¿Resuelve el problema?:** Aunque parezca tonto, me he encontrado este año y en años anteriores programas que no resuelven la consigna que fue planteada. Por favor no pierdan de vista la consigna que deben resolver, el programa debe solucionar un problema puntual, no "*hacer algo parecido a lo que se pide*".
- **Soluciones complejas:** Hay veces que me he encontrado código muy complejo y *rebuscado* para resolver algo simple. Siempre tengan en mente el principio [KISS \("Keep It Simple, Stupid"\)](#), no por resolver un problema de manera rebuscada y compleja van a ser mejores programadores.
- **Repositorios GIT:** Todos los ejercicios propuestos deben ser resueltos y publicados en el repositorio Git personal. Junto con los documentos de cátedra se encuentra una guía para crear y trabajar los pasos básicos de GIT con GitLab. No considero válidos códigos remitidos en archivos comprimidos, tarballs, etc.

## Ejercicio 1 - Getopt

Crear una calculadora, donde se pase como argumentos luego de la opción -o el operador que se va a ejecutar (+,-,\*,/), luego de -n el primer número de la operación, y de -m el segundo número.

Ejemplo:

```
./calc -o + -n 5 -m 6
5 + 6 = 11
```

Considerar que el usuario puede ingresar los argumentos en cualquier orden. El programa deberá verificar que los argumentos sean válidos (no repetidos, números enteros, y operaciones válidas).

## Ejercicio 2 - Getopt

Escribir un programa que reciba dos nombres de archivos por línea de órdenes utilizando los parámetros “-i” y “-o” procesados con getopt().

El programa debe verificar que el archivo pasado a “-i” exista en el disco. De ser así, lo abrirá en modo de solo lectura, leerá su contenido, y copiará dicho contenido en un archivo nuevo cuyo nombre será el pasado a “-o”. Si el archivo nuevo ya existe, deberá sobrescribirlo.

## Ejercicio 3 - Fork

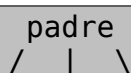
Escribir un programa que en ejecución genere dos procesos, uno padre y otro hijo. El hijo debe escribir "Soy el hijo" 5 veces. El padre debe escribir "Soy el padre" 2 veces. El padre debe esperar a que termine el hijo y mostrar el mensaje "Mi proceso hijo termino".

## Ejercicio 4 - Fork

Escribir un programa en C que en ejecución genere 2 procesos (padre e hijo). Este programa recibirá como único argumento del intérprete de órdenes un entero. El padre incrementará ese valor en 4 y lo mostrará por pantalla. El hijo decrementará ese valor en 2 y lo mostrará en pantalla.

## Ejercicio 5 - Fork

Realice otro programa que genere la siguiente configuración de procesos, y cada proceso al iniciar deberá mostrar “Soy el proceso N, mi padre es M” N será el PID de cada hijo, y M el PID del padre.



hijo1 hijo2 hijo3

## Ejercicio 6 - Señales

Hacer un programa que capture la señal que se envía al presionar Ctrl+c y la ignore la primera vez, mostrando un mensaje por pantalla "Esta vez me saldré, inténtalo nuevamente". Establecer los valores por defecto para que al segundo intento se realice lo esperado.

## Ejercicio 7 - Señales

Desde un proceso padre crear un hijo y enviarle una señal SIGUSR1 cada 5 segundos. El proceso hijo estará a la espera y escribirá un aviso en pantalla cada vez que llega la señal de su padre.

## Ejercicio 8 - Señales

Realice un programa que genere dos procesos. El proceso hijo1 enviará la señal SIGUSR1 al proceso padre y mostrará la cadena "ping" cada 5 segundos. El proceso padre, al recibir la señal SIGUSR1 enviará esta misma señal al proceso hijo2. El proceso hijo2, al recibir dicha señal mostrará la cadena "pong" por pantalla.

```
Soy proceso hijo1 con PID=1545: "ping"
Soy proceso hijo2 con PID=1547: "pong"
```

```
[... 5 segundos mas tarde ...]
Soy proceso hijo1 con PID=1545: "ping"
Soy proceso hijo2 con PID=1547: "pong"
[... y así sucesivamente ...]
```

## Ejercicio 9 – Señales

Escribir un programa que genere 3 procesos hijos. El proceso padre enviará una señal a cada uno de sus hijos (SIGUSR1). Cada hijo mostrará por pantalla el mensaje: "Soy el PID xxxx, recibí la señal yyy de mi padre PID zzzz", donde xxxx es el pid del hijo en cuestión, yyy el número de señal, y zzzz el pid del proceso padre inicial.

## Ejercicio 10 – Señales / Pipes

Escribir un programa que genere un proceso hijo (fork), y a su vez, éste genere un nuevo proceso hijo (fork) (ver esquema abajo). Todos los procesos estarán comunicados mediante un pipe.

A → B → C

El proceso B escribirá en el pipe el mensaje "Mensaje 1 (PID=BBBB)\n" en el momento

en que el proceso A le envíe la señal USR1.

Cuando esto ocurra, el proceso B le enviará la señal USR1 al proceso C, y C escribirá en el pipe el mensaje "Mensaje 2 (PID=CCCC)\n".

Cuando el proceso C escriba este mensaje, le enviará al proceso A la señal USR2, y el proceso A leerá el contenido del pipe y lo mostrará por pantalla, en el formato:

```
A (PID=AAAA) leyendo:  
Mensaje 1 (PID=BBBB)  
Mensaje 2 (PID=CCCC)
```

### Notas:

- AAAA, BBBB, y CCCC son los respectivos **PIDs** de los procesos A, B y C.
- Inicialmente el proceso B envía un mensaje al pipe cuando A le hace llegar la señal USR1, por lo que B deberá quedar esperando al principio una señal.
- El proceso C también deberá quedar esperando inicialmente una señal que recibirá desde el proceso B para poder realizar su tarea.
- El proceso A inicialmente envía la señal USR1 al proceso B, y automáticamente debe quedar esperando una señal desde el proceso C para proceder a leer desde el pipe.
- Los saltos de línea en los mensajes ("\\n") son necesarios para que el pipe acumule varias líneas, y no concatene los mensajes en la misma línea.
- El pipe es uno solo en el que los procesos B y C deberán escribir, y el proceso A deberá leer.
- **Pista:** la utilización de señales es para sincronizar los procesos, no necesariamente los handlers deben gestionar el pipe.

## Ejercicio 11 – Pipes

Escribir un programa que lance dos procesos hijos (fork).

Uno de los procesos hijos se encargará de leer desde la entrada estándar líneas de texto, y en la medida en que el usuario escriba, el proceso las irá enviando por un pipe que compartirá con el otro proceso hijo.

El segundo proceso se encargará de leer desde el pipe las líneas que el primer proceso escriba, y las irá mostrando por pantalla en el formato "Leyendo (pid: 1234): mensaje", donde 1234 es el pid de este segundo proceso, y "mensaje" es el contenido leído desde el pipe.

## Ejercicio 12 – FIFO / pipes

Crear un sistema Productor-Consumidor (Escritor-Lector) donde un proceso productor almacene un mensaje de saludo en una tubería FIFO. Ese mensaje será enviado mediante línea de comandos como argumento del programa. Ejemplo

```
./saludofifo HOLA
```

Otro programa (consumidor) deberá leer el mensaje desde la tubería FIFO, generar un proceso hijo (fork) y enviarle por PIPE el mensaje. El proceso hijo mostrará por pantalla el mensaje recibido.

```
Proc1_fifo_escritor → FIFO → Proc2_fifo_lector → pipe → Proc2hijo
```

## Ejercicio 13 – Multiprocessing

Escribir un programa que genere tres procesos hijos (fork) cuyos nombres serán “Hijo1”, “Hijo2” e “Hijo3”.

El proceso padre debe mostrar por pantalla el mensaje “Lanzando hijo NN” en la medida que ejecuta cada uno de los procesos.

Cada hijo que se lance mostrará por pantalla su pid, el pid de su padre, esperará un segundo, y luego terminará.

El padre debe esperar a la terminación de sus hijos.

## Ejercicio 14 – Multiprocessing / Pipe

Replicar el comportamiento del ejercicio 11 utilizando el módulo Multiprocessing.

## Ejercicio 15 – Multiprocessing / Pipe / FIFO

Replicar el comportamiento del ejercicio 12 utilizando el módulo Multiprocessing para la creación del proceso hijo del programa consumidor.

## Ejercicio 16 – Mini Chat

Escribir dos programas que interactúen entre si como si fuesen clientes peer-to-peer. El funcionamiento será el siguiente:

- Llamemos AA al primer programa en ejecutarse, y BB al segundo.
- AA quedará escuchando un texto desde el otro programa.
- BB permitirá al usuario escribir por entrada estándar un mensaje. Al dar *enter* se enviará dicho mensaje a AA, y automáticamente BB quedará esperando un mensaje remoto.
- AA al recibir el mensaje lo mostrará por pantalla, y permitirá al usuario ingresar desde entrada estándar un texto.
- Se repite el comportamiento anterior, pero en sentido inverso.
- La idea es que se produzca una sucesión de mensajes de ida y vuelta entre AA y BB, donde BB enviará el primer mensaje, y todos los mensajes serán leídos desde la entrada estándar.

- El mecanismo de comunicación entre procesos a utilizar queda a decisión del diseñador y programador :)

## Ejercicio 17 – Fork / MQ

Escriba un programa que cree diez procesos hijos (fork).

Cada proceso al iniciar debe escribir por pantalla "Proceso xx, PID: xxxx"

A su vez, cada proceso debe esperar  $i$  segundos, donde  $i$  es el número de proceso hijo, y luego escribir en un formato "xxxxx\t" el pid en una cola de mensajes.

El padre debe esperar a que todos los hijos terminen, y luego leer el contenido de la cola de mensajes y mostrarlo por pantalla.

## Ejercicio 18 – Multiprocessing / MQ

Reescriba el programa anterior pero esta vez haciendo uso del módulo multiprocessing.

## Ejercicio 19 – Pool of Workers / MQ

Reescriba el programa anterior pero esta vez haciendo uso de pools de procesos.

## Ejercicio 20 – Pool of workers

Escriba un programa que haga uso de un pool de procesos para calcular las raíces cuadradas de los números impares comprendidos entre los valores  $N$  y  $M$  pasados por argumentos al programa mediante los modificadores  $-n$  y  $-m$  (validar las entradas con `getopt`).

Ejemplo de ejecución:

```
python3 calculador.py -n 12 -m 20
Raíz cuadrada de 13: 3.605
Raíz cuadrada de 15: 3.872
Raíz cuadrada de 17: 4.123
Raíz cuadrada de 19: 4.358
```

Escriba dos versiones del programa:

- Una haciendo uso de la función **map**
- Una haciendo uso de la función **apply**

## Ejercicio 21 – Threading / Pipe / FIFO

Reescribir el programa consumidor del ejercicio 15 para que haga uso de multithreading.



## Ejercicio 22 – Threading

Adaptar el ejercicio 13 para que haga uso de hilos con threading. Los hijos, además de mostrar su PID y el PID del proceso padre, también deberán mostrar el identificador de hilo de ejecución.

## Ejercicio 23 – Threading / MQ

Reescriba el programa del ejercicio 17 pero esta vez haciendo uso del módulo threading.

## Ejercicio 24 – Sockets INET/DGRAM

Escriba un programa cliente-servidor con sockets INET/DGRAM que tenga el siguiente comportamiento.

1. El usuario ejecutará el programa servidor pasándole dos argumentos:
  1. -p: El puerto en el que va a atender el servicio (por defecto debe atender en todas las direcciones de red configuradas en el sistema operativo).
  2. -f: Una ruta a un archivo de texto en blanco.
2. El servidor creará el socket con los datos especificados, y creará, si no existe, el archivo de texto.
3. El cliente recibirá dos argumentos por línea de comandos: la dirección IP (-a) y el puerto en el que atiende el servidor (-p).
4. El cliente comenzará a leer desde STDIN texto hasta que el usuario presione Ctrl+d.
5. El cliente enviará todo el contenido por el socket al servidor.
6. El servidor leerá todo el contenido desde el socket hasta que encuentre un EOF.
7. El servidor almacenará todo el contenido en el archivo de texto creado.

NOTA: los parámetros serán pasados por argumento y parseados usando getopt.

Ejemplo de carga del servidor:

```
python3 servidor.py -p 1234 -f /tmp/archivo.txt
```

Ejemplo de carga del cliente:

```
python3 cliente.py -a 192.168.0.23 -p 1234
```

## Ejercicio 25 – Sockets INET/STREAM

Replique el comportamiento del ejercicio anterior, esta vez utilizando protocolo de transporte confiable (TCP).

## Ejercicio 26 – Sock Inet Stream / fork

Escribir un programa servidor que levante un socket INET STREAM en todas las direcciones locales, y en un puerto específico que será pasado por parámetros de línea de comandos.

El servidor debe escuchar en el puerto especificado, y debe permitir, mediante subprocesos (fork) atender a varios clientes simultáneamente.

Ejecutar el servidor, por ejemplo, así:

```
./servidor 1234
```

Escribir un programa cliente que utilice sockets INET STREAM que reciba por parámetros dirección ip del servidor y puerto de esta forma:

```
./cliente 127.0.0.1 1234
```

El cliente con esos datos debe conectarse al servidor al puerto especificado, y luego debe leer desde la entrada estándar (STDIN\_FILENO) y todo lo que el usuario escriba debe ir enviandoselo al servidor. El servidor los recibirá e irá mostrándolos por pantalla junto con el **identificador del proceso hijo** que atiende al cliente conectado, la **dirección del cliente**, y su **puerto** cliente.

Cuando el usuario escriba "exit" el cliente debe terminar la conexión y cerrar. El servidor debe cerrar la conexión con ese cliente.

## Ejercicio 27 – Sock Inet Str / multiprocessing

Modifique el programa anterior (servidor) para que haga uso de multiprocessing.

## Ejercicio 28 – Sock Inet Str / threading

Modifique el programa anterior (servidor) para que haga uso de multithreading.

## Ejercicio A (opcional) – Stress Test

Considerando el cliente desarrollado para los ejercicios anteriores, escriba un nuevo cliente que, en vez de leer desde la entrada estándar y enviar el texto al servidor, genere 20 procesos hijos que trabajen en paralelo, y cada uno realice una conexión automatizada contra el servidor, que le envíe un millón de cadenas de caracteres de 128B al servidor. El último valor será un "exit", lo que producirá que la comunicación con el servidor termine.

Pruebe conectar con este nuevo cliente a cada uno de los tres servidores programados antes (fork, multiprocessing, threading). Realizar cada experimento por separado tomando el tiempo en el que demora el cliente desde que inicia hasta que termina. Compare los tiempos de las tres experiencias.

NOTA: puede usar comandos como "time" para ejecutar el cliente (*man time*).

## Ejercicio 29 – Sock Inet Str / threading

Escriba un servidor tcp que atienda un puerto pasado por argumento (**-p** en getopt), y reciba los siguientes comandos: ABRIR, CERRAR, AGREGAR, y LEER, por parte de los clientes.

Si el cliente envía un comando ABRIR, el servidor deberá solicitarle un nombre de archivo.

Si el cliente envía un comando "AGREGAR" el servidor deberá solicitarle una cadena de texto para agregar al final del archivo.

Si el cliente envía un comando "LEER" el servidor le deberá enviar al cliente el contenido del archivo.

Si el cliente envía un comando "CERRAR" el servidor deberá cerrar el archivo y cerrar la comunicación con el cliente.

El servidor deberá poder mantener conexiones en simultáneo con varios clientes mediante un sistema multihilo.

Deberá también programar a los clientes para que le soliciten comandos al usuario mediante un simple menú de opciones, e interactúe con el servidor para que pueda realizar sus tareas.

## Ejercicio 30 – Lock (multiprocessing)

Escriba un programa que lance 15 procesos hijos. El programa recibirá, mediante el uso del modificador “-f”, un nombre de archivo que deberá crear o, si ya existe, abrir y limpiar. Además, recibirá por argumento un número mediante el modificador “-r”. Ese número representará la cantidad de iteraciones (ver más adelante).

Una llamada típica al programa podría ser:

```
./programa -r 5 -f /tmp/test.txt
```

Cada proceso tendrá asociada una letra del alfabeto (A, B, C, etc.), y deberá escribir “su” letra tantas veces como iteraciones se hayan especificado con “-r”, y con un delay de 1 segundo. Es decir, con la llamada anterior, cada proceso deberá escribir una letra (“A”, “B”, etc) 5 veces con intervalo de un segundo... de esa forma, el programa completo no demorará más de 5 segundos (todos los procs corren en paralelo).

Haga uso de lock para que las N letras (5 en el ejemplo) escritas por cada proceso se mantengan juntas y no intercaladas con los demás.

El resultado en el archivo del ejemplo será algo así:

```
AAAAADDDDDDEEEEEBBBBB ...
```

Y no algo así

```
AABADDEBABEFBFAGGHD...
```

### Ejercicio 31 – Lock (threading)

Reescriba el programa anterior para que haga uso de hilos de ejecución en vez de procesos, y utilice una versión de lock que sea *thread-safe*.

### Ejercicio 32 – ProcessPoolExecutor

Escriba una función que calcule la serie de Fibonacci para un argumento dado. El programa hará uso de ProcessPoolExecutor para que calcule las series de Fibonacci de una lista de números enteros comprendidos entre N y M, siendo estos dos valores números enteros pasados por argumento mediante las opciones “-n” y “-m” al programa.

Así, un ejemplo de ejecución sería el siguiente:

```
./fibo_pool.py -n 5 -m 18
```

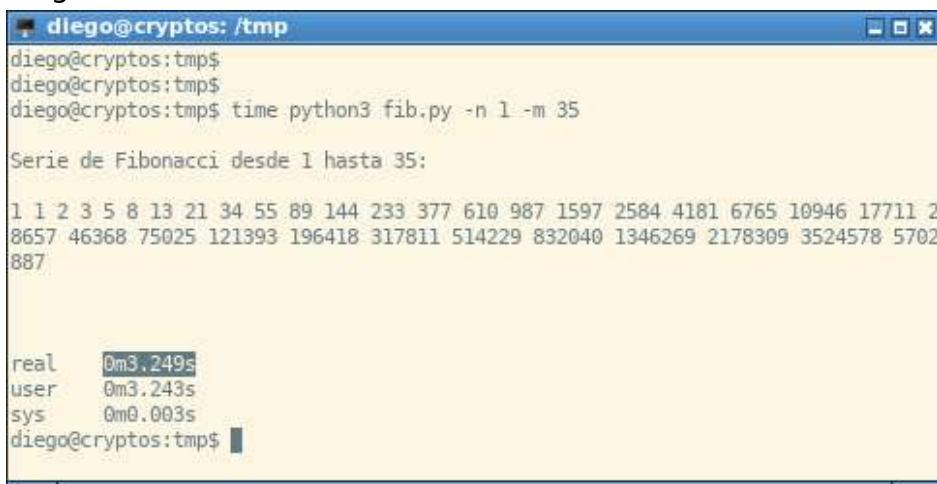
De este ejercicio se deben escribir dos versiones, una que utilice el algoritmo recursivo para calcular la serie de Fibonacci, y otra que utilice el algoritmo iterativo.

### Ejercicio 33 – ThreadPoolExecutor

Reescriba el programa anterior para ejecutarlo con hilos concurrentes mediante ThreadPoolExecutor. Al igual que el ejercicio anterior, escribir una versión que utilice el algoritmo recursivo y otra haciendo uso del algoritmo iterativo.

### Ejercicio B (Opcional) – Performance

Mida el rendimiento en ejecución de los cuatro programas escritos en los dos ejercicios anteriores, y para los mismos argumentos (por ejemplo, calcular las series entre 1 y 35, o entre 1 y 40). Para medir el tiempo de ejecución puede usar el comando time de la shell de la siguiente manera:



```
diego@cryptos: /tmp
diego@cryptos:tmp$
diego@cryptos:tmp$
diego@cryptos:tmp$ time python3 fib.py -n 1 -m 35

Serie de Fibonacci desde 1 hasta 35:

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887

real    0m3.249s
user    0m3.243s
sys     0m0.003s
diego@cryptos:tmp$
```

Suba al repo git un archivo de texto con las salidas de terminal donde se vean los tiempos de ejecución de cada caso, y una breve explicación de por qué alguna ejecución se lleva a cabo en mas o menos tiempo que otra.

Ahora instale el paquete “**pypy3**” y utilice pypy3 en vez de python3 para las ejecuciones, y mida nuevamente los tiempos... ¿qué diferencia nota?

Agregue sus notas al archivo de texto, y una breve explicación del pypy3: ¿qué es? ¿para qué sirve?

## Ejercicio C (opcional) – (minichat improved)

Reescribir el ejercicio 16 (minichat) para que permita a los dos procesos comunicarse en **red tcp/ip**.

Si recuerda, en el ejercicio 16 cada uno de los end-points del chat podía enviar solamente un mensaje. Mejore esta característica para que cada extremo pueda mandar cualquier cantidad de mensajes tomando el control de la comunicación hasta que el usuario escriba una línea con el mensaje “cambio.”.

Así, una conversación de chat típica podría tener esta forma (comienza escribiendo el usuario userAA en este caso):

Programa AA

```
Hola
qué tal?
Cómo estás?
cambio.
userBB: Bien!
userBB: Vos?
```

Programa BB

```
userAA: Hola
userAA: qué tal?
userAA: Cómo estás?
Bien!
Vos?
cambio.
```