

	<p style="text-align: center;">UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<p style="text-align: center;"><b>Formato:</b> Guía de Práctica de Laboratorio / Talleres / Centros de Simulación</p>		
<p>Aprobación: 2022/03/01</p>	<p>Código: GUIA-PRLD-001</p>	<p>Página: 1</p>

## GUÍA DE LABORATORIO

INFORMACIÓN BÁSICA					
<b>ASIGNATURA:</b>	Laboratorio de Programación de Sistemas				
<b>TÍTULO DE LA PRÁCTICA:</b>	Procesos y Señales				
<b>NÚMERO DE PRÁCTICA:</b>	Guía 7	<b>AÑO LECTIVO:</b>	2023	<b>NRO. SEMESTRE:</b>	A
<b>TIPO DE PRÁCTICA:</b>	<b>INDIVIDUAL</b>				
	<b>GRUPAL</b>	X	<b>MÁXIMO DE ESTUDIANTES</b>	5	
<b>FECHA INICIO:</b>	14/07/2023	<b>FECHA FIN:</b>	20/07/2023	<b>DURACIÓN:</b>	4 H
<b>RECURSOS A UTILIZAR:</b>					
Consideraciones: Utilizar el Sistema Operativo de Linux en cualquiera de sus versiones					
<b>DOCENTE(s):</b> Mg.					
Edith Giovanna Cano Mamani / Polanco Argüelles Jorge Manuel					

OBJETIVOS/TEMAS Y COMPETENCIAS	
<b>OBJETIVOS:</b>	
La presente práctica de laboratorio tiene como objetivo manejar crear procesos y manejar señales.	
<b>TEMAS:</b>	
La llamada de sistema fork()	
La llamada de sistema exec()	
La llamada de sistema signal()	
La llamada de sistema kill()	
<b>COMPETENCIAS</b>	C.c: Diseña responsablemente sistemas componentes o procesos para satisfacer necesidades dentro de restricciones realistas, económicas, medioambientales, sociales, políticas, éticas, de salud, de seguridad, manufacturación y sostenibilidad
	C.m: Construye responsablemente soluciones siguiendo un proceso adecuado llevando a cabo las pruebas ajustadas a los recursos disponibles del cliente.
	C.p: Aplica la forma flexible, técnicas, métodos, principios, normas, estándares y herramientas de ingeniería necesarias para la construcción de software e implementación de sistemas de información.

## CONTENIDO DE LA GUÍA

## **I. MARCO CONCEPTUAL**

### **Creación de procesos mediante fork**

En Unix, un proceso es creado mediante la llamada al sistema `fork()`. El proceso que realiza la llamada se denomina proceso padre (parent process) y el proceso creado a partir de la llamada se denomina proceso hijo (child process). La sintaxis de la llamada, efectuada desde el proceso padre, es:

`valor = padre()`

La llamada `fork()` devuelve un valor distinto para los procesos padre e hijo: al proceso padre se le devuelve el PID del proceso hijo, mientras que al proceso hijo se le devuelve "0".

Las acciones que implican ejecutar `fork()` son llevadas a cabo por el núcleo (kernel) del sistema operativo Unix. Dichas acciones son las siguientes:

1. Asignación de un hueco en la tabla de procesos para el nuevo proceso (hijo).
2. Asignación de un identificador único (PID) al proceso hijo.
3. Copia de la imagen del proceso padre al proceso hijo (con excepción de la memoria compartida).
4. Asignación al proceso hijo del estado "preparado para ejecución".
5. Devolución de los dos valores de retorno de la función: al proceso padre se le entrega el PID del proceso hijo, y al proceso hijo se le entrega "0".

### **Inicio de programas mediante exec.**



La llamada `exec()` sustituye el programa que las invoca por un nuevo programa. Mientras que `fork()` cree nuevos procesos, `exec()` sustituye la imagen en memoria del proceso que por otra nueva (es decir, que sustituye todos los elementos del proceso: código del programa, datos, pila y heap). El PID del proceso es el mismo que antes de realizar la llamada `exec()`, pero ahora ejecuta otro programa. El proceso pasa a ejecutar el nuevo programa desde el inicio y la imagen en memoria del antiguo programa se pierde al verse sobrescrita. La imagen en memoria del antiguo programa se pierde para siempre, es decir, todo el código que escribíamos posteriormente a la ejecución con éxito de la llamada `exec()` será inalcanzable.

La combinación de las llamadas `fork()` y `exec()` es el mecanismo que ofrece Unix para la creación de un nuevo proceso (`fork()`) que ejecute un programa determinado (`exec()`).

Existen seis posibles llamadas tipo `exec()`:

- int `execv`
- int `execl`
- int `execve`
- int `execvp`
- int `execlp`

Zona de datos entre procesos vinculados por `fork`

	<p style="text-align: center;"><b>UNIVERSIDAD NACIONAL DE SAN AGUSTIN</b>  <b>FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS</b>  <b>ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</b></p>	
<p style="text-align: center;"><b>Formato:</b> Guía de Práctica de Laboratorio / Talleres / Centros de Simulación</p>		
<p><b>Aprobación:</b> 2022/03/01</p>	<p><b>Código:</b> GUIA-PRLD-001</p>	<p><b>Página:</b> 3</p>

Dos procesos vinculados por una llamada fork() (padre e hijo) poseen zonas de datos propias, de uso privado (no compartidas). Obviamente al tratarse de procesos diferentes, cada uno posee un espacio de direcciones independientes e inviolable.

### **Ficheros abiertos entre procesos vinculados por fork**

Los descriptores de ficheros (que indican qué ficheros tiene abierto un proceso) en el momento de hacer la llamada fork() son compartidos por los dos procesos que resulten de dicha llamada. Dicho de otra forma, los descriptores de ficheros en uso por el proceso padre son heredados por el proceso hijo generado

### **Manejo de señales**

Una señal es un "aviso" que puede enviar un proceso a otro proceso. El sistema operativo Unix se encarga de que el proceso que recibe la señal la trate inmediatamente. De hecho, termina la línea de código que esté ejecutando y salta a la función de tratamiento de señales adecuada. Cuando termina de ejecutar esa función de tratamiento de señales, continua con la ejecución en la línea de código donde lo había dibujado.

El sistema operativo envía señales a los procesos en determinadas circunstancias. Por ejemplo, si en el programa que se está ejecutando en una shell nosotros apretamos Ctrl-C, se está enviando una señal de terminación al proceso. Este la trata inmediatamente y sale. Si nuestro programa intenta acceder a una memoria no válida (por ejemplo, accediendo al contenido de un puntero a NULL), el sistema operativo detecta esta circunstancia y le envía una señal de terminación inmediata, con lo que el programa "se cae".

Las señales van identificadas por un número entero. Las señales son interrupciones de software, permiten el manejo de eventos asíncronos. Cada señal tiene un nombre. Ellos comienzan con SIG, no se detallarán todas, pero si algunas como por ejemplo:

**SIGALRM:** generada cuando el timer asociado a la función alarm expira. También cuando el timer de intervalo es configurado (setitimer)

**SIGCHLD:** Cuando un proceso termina o para, el proceso envía esta señal a su padre. Por defecto esta señal es ignorada. Normalmente el proceso padre invoca la función wait para obtener el estatus de término del proceso hijo. Se evita así la creación de procesos "zombies".

**SIGCONT:** es enviada para reanudar un proceso que ha sido parado (suspendido) con SIGSTOP.

**SIGINT:** generada con DELETE o Control-C

**SIGKILL:** Permite terminar un proceso.

**SIGTSTP:** generada cuando presionamos Control-Z. Puede ser ignorada.

**SIGSTOP:** similar a SIGTSTP pero no puede ser ignorada o capturada.

**SIGUSR1:** Es una señal definida por el usuario para ser usada en programas de aplicación.

**SIGUSR2:** Otra como la anterior.

## II. EJERCICIO/PROBLEMA RESUELTO POR EL DOCENTE

### EJERCICIO 1.

Utilizando la llamada al sistema fork imprimir dos veces el mensaje “hola mundo”

```
//codigo01.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();

    printf("Hola Mundo\n");
    return 0;
}
```

Para compilar se usa el siguiente comando:

```
$ gcc -Wall -o codigo01 codigo01.c
```

Para ejecutarlo usar el siguiente comando:

```
$ ./codigo01
```

### EJERCICIO 2.

Utilizando el código a seguir determine el número de veces que saldrá la palabra “hola” en el caso que pongamos tres llamadas al sistema fork()

```
//codigo02.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hola\n");
    return 0;
}
```

Para compilar se usa el siguiente comando:

```
$ gcc -Wall -o codigo02 codigo02.c
```

Para ejecutarlo usar el siguiente comando:

```
$ ./codigo02
```

### EJERCICIO 3.

En un programa de c, mostrar como se crea un proceso hijo asociado a un proceso padre utilizando la llamada a sistema fork().

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    // El proceso hijo se imprime porque hay un valor de cero
    if (fork() == 0)
        printf("Hola desde el hijo!\n");

    // El proceso padre retorna por el valor no es cero
    else
        printf("Hola desde el padre!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

Para compilar se usa el siguiente comando:

```
$ gcc -Wall -o codigo03 codigo03.c
```

Para ejecutarlo usar el siguiente comando:

```
$ ./codigo03
```

### EJERCICIO 4.

Un proceso que ya no tiene más un padre porque ha terminado su ciclo de ejecución o ha terminado su ejecución sin esperar a que su proceso hijo haya terminado su ejecución se llama proceso huérfano, utilizando un programa C mostrar que esta situación puede ocurrir.

```
//codigo04.c
#include<stdio.h>
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
int main()
{
    // Crea un proceso hijo
    int pid = fork();

    if (pid > 0)
        printf("en el proceso padre\n");

    // Note que el pid es 0 en el proceso hijo
    // y negativo si el proceso fork() falla
    else if (pid == 0)
    {
        sleep(30);
        printf("en el proceso hijo\n");
    }

    return 0;
}
```

Para compilar se usa el siguiente comando:

```
$ gcc -Wall -o codigo04 codigo04.c
```

Para ejecutarlo usar el siguiente comando:

```
$ ./codigo04
```

### EJERCICIO 5.

Un proceso que ha finalizado la ejecución, pero todavía tiene una entrada en la tabla de procesos para informar a su proceso padre este proceso es conocido como proceso zombi, escribir un programa en C para mostrar un proceso zombi el proceso hijo se torna zombi

```
//codigo05.c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork retorna el id del proceso
    // en el proceso padre
    pid_t child_pid = fork();

    // Proceso padre
```

```
if (child_pid > 0)
    sleep(50);

// Proceso hijo
else
    exit(0);

return 0;
}
```

Para compilar se usa el siguiente comando:

```
$ gcc -Wall -o codigo05 codigo05.c
```

Para ejecutarlo usar el siguiente comando:

```
$ ./codigo05
```

Verificar la ejecución del programa em otra terminal utilizando el siguiente comando

```
$ ps -l
```

### EJERCICIO 6.

Mostrar el uso de la llamada al sistema fork() con el siguiente programa en c, mostrar su seguimiento con el uso del comando ps y su opción "la"

```
// codigo06.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main() {
    pid_t rf;
    rf = fork();
    switch (rf) {
        case -1:
            printf("No he podido crear el proceso hijo \n");
            break;
        case 0:
            printf("Soy el hijo, mi PID es %d y mi PPID es %d \n", getpid(),
                getppid());
            sleep(20); //suspende el proceso 20 segundos
            break;
        default:
            printf("Soy el padre, mi PID es %d y el PID de mi hijo es %d \n", getpid(), rf);
            sleep(30); //suspende el proceso 30 segundos. Acaba antes el hijo.
    }
}
```

```
}  
printf("Final de ejecución de %d \n", getpid());  
exit(0);  
}
```

Para compilar se usa el siguiente comando:

```
$ gcc -Wall -o codigo06 codigo06.c
```

Para ejecutarlo en segundo plano usar el siguiente comando:

```
$ ./codigo06 &
```

Verificar la ejecución del programa en otra terminal utilizando el siguiente comando

```
$ ps -l
```

### **EJERCICIO 7.**

Escribir los programas codigo07.c y codigo08.c como se muestra a continuación, ejecutar el codigo07 en background y verificar su ejecución con el comando ps y la opción "la"

```
//codigo07.c  
#include <stdio.h>  
#include <unistd.h>  
#include <string.h>  
#include <stdlib.h>  
int main (int argc, char *argv[]) {  
    int i;  
    printf("Ejecutando el programa invocador (codigo07). Sus argumentos son:\n");  
    for (i = 0; i < argc; i++)  
        printf("argv[%d]: %s\n", i, argv[i]);  
    sleep(10);  
    strcpy(argv[0], "codigo08");  
    if (execv("./codigo08", argv) < 0) {  
        printf("Error en la invocacion a codigo08\n");  
        exit(1);  
    };  
    exit(0);  
}
```

```
//codigo08.c  
#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
int main (int argc, char *argv[]) {
```



```
int i;
char a[2000000];
printf("Ejecutando el programa invocado (codigo08). Sus argumentos son:\n");
for (i = 0; i < argc; i++)
    printf("argv[%d]: %s\n", i, argv[i]);
sleep(10);
exit(0);
}
```

Compilar cada uno de los programas con los siguientes comandos:

```
$ gcc -Wall -o codigo07 codigo07.c
```

```
$ gcc -Wall -o codigo08 codigo08.c
```

Ejecutar el programa ejercicio2 em background como se muestra a continuación

```
$/codigo07 arg1 arg2 ... argN &
```

Verifique cada uno de los procesos lanzados con el comando ps y la opción "la"

```
$ps la
```

### **EJERCICIO 8.**

Comprobar con el siguiente programa que los procesos vinculados por una llamada fork() (padre e hijo) poseen zonas de datos propias, de uno privado (no compartidas), cada uno posee un espacio de direcciones independiente e inviolable, en el siguiente programa se asigna valor distinto a una misma variable según se trate de la ejecución del proceso padre o del hijo.

```
//codigo09.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i;
    int j;
    pid_t rf;
    rf = fork();
    switch (rf) {
        case -1:
            printf("\nNo he podido crear el proceso hijo");
            break;
        case 0:
            i = 0;
```

```
printf("\nSoy el hijo, mi PID es %d y mi variable i (inicialmente a %d) es par",
getpid(), getppid());
    for (j = 0; j < 5; j++) {
        i++;
        i++;
        printf("\nSoy el hijo, mi variable i es %d", i);
    };
    break;
default:
    i = 1;
    printf("\nSoy el padre, mi PID es %d y mi variable i (inicialmente a %d) es impar",
getpid(), rf);
    for (j = 0; j < 5; j++) {
        i++;
        i++;
        printf("\nSoy el padre, mi variable i es %d", i);
    }
}
printf("\nFinal de ejecucion de %d\n", getpid());
exit(0);
}
```

Para compilar el programa se requiere la siguiente sentencia:

```
$ gcc -Wall -o codigo09 codigo09.c
```

Para su ejecución se requiere la siguiente línea

```
$ ./codigo09
```

### **EJERCICIO 9.**

Escribir un programa en c que permita a dos procesos escribir distintas cadenas de caracteres en los mismos ficheros.

```
// codigo10.c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
int main() {
    int i;
    int fd1, fd2;
    const char string1[10]= "*****";
    const char string2[10]= "-----";
```

```
pid_t rf;
fd1 = creat("ficheroA", 0666);
fd2 = creat("ficheroB", 0666);
rf = fork();
switch (rf) {
    case -1:
        printf("\nNo he podido crear el proceso hijo");
        break;
    case 0:
        for (i = 0; i < 10; i++) {
            write(fd1, string2, sizeof(string2));
            write(fd2, string2, sizeof(string2));
            usleep(1); /* Abandonamos voluntariamente el procesador */
        }
        break;
    default:
        for (i = 0; i < 10; i++) {
            write(fd1, string1, sizeof(string1));
            write(fd2, string1, sizeof(string1));
            usleep(1); /* Abandonamos voluntariamente el procesador */
        }
}
printf("\nFinal de ejecucion de %d \n", getpid());
exit(0);
}
```

Para compilar el siguiente programa es necesario poner la siguiente línea:

```
$ gcc -Wall -o codigo10 codigo10.c
```

```
$ ./codigo10
```

Para verificar el contenido de ficheroA y ficheroB realizamos los siguientes comandos

```
$ cat ficheroA
```

```
$ cat ficheroB
```

### **EJERCICIO 10.**

Utilizando la llamada al sistema wait() hacer que un proceso invocado quede suspendido hasta que termine alguno de sus procesos hijos.

```
// codigo11.c
#include <unistd.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include <wait.h>
int main() {
    pid_t rf;
    rf = fork();
    switch (rf) {
        case -1:
            printf("\nNo he podido crear el proceso hijo");
            break;
        case 0:
            printf("Soy el hijo, mi PID es %d y mi PPID es %d\n", getpid(), getppid());
            sleep(10);
            break;
        default:
            printf("Soy el padre, mi PID es %d y el PID de mi hijo es %d\n", getpid(), rf);
            wait(0);
    }
    printf("\nFinal de ejecucion de %d \n", getpid());
    exit(0);
}
```

Compila el programa con la siguiente linea

```
$ gcc -Wall -o codigo11 codigo11.c
```

Ejecutamos el programa

```
$ ./codigo11
```

Visualizamos los procesos lanzados

```
$ ps la
```

### **EJERCICIO 11.**

Hacer uso de la llamada de sistema `signal()` para cambiar el comportamiento del siguiente programa con respecto a las señales recibidas, en caso de imposibilidad de ser detenido el programa utilice el comando `KILL`

```
// codigo12.c
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#define VUELTAS 1000000000LL
void confirmar(int sig) {
    char resp[100];
```

```
write(1, "Quiere terminar? (s/n):", 24);
read(0, resp, 100);
if (resp[0]=='s') exit(0);
}
int main(void) {
    long long int i;
    signal(SIGINT, SIG_IGN);
    write(1, "No hago caso a CONTROL-C\n", 25);
    for (i=0; i<VUELTAS; i++);
    signal(SIGINT, SIG_DFL);
    write(1, "Ya hago caso a CONTROL-C\n", 25);
    for (i=0; i<VUELTAS; i++);
    signal(SIGINT, confirmar);
    write(1, "Ahora lo que digas\n", 19);
    for (i=0; i<VUELTAS; i++);
    exit(0);
}
```

Compila el programa con la siguiente linea

```
$ gcc -Wall -o codigo12 codigo12.c
```

Ejecutamos el programa

```
$ ./codigo12
```

Para eliminar el proceso se puede realizar el siguiente comandos

```
$ kill -SIGINT procesos
```

## EJERCICIO 12.

Utilizando señales hacer que un proceso envíe señales para matar a un proceso hijo utilizando la llamada al sistema kill() utilizando el siguiente código

```
//codigo13.c
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#define VUELTAS 10000000000LL

void hijo(int sig) {
    printf("\t\t\tPadre! ¿Qué haces?\n");
    printf("\t\t\tFinal de ejecución de %d \n", getpid());
    kill(getppid(), SIGUSR1);
}
```

```
        exit(0);
    }

    void padre(int sig) {
        printf("\tHijo! ¿Qué he hecho?\n");
        printf("\tFinal de ejecución de %d \n", getpid());
        exit(0);
    }


    int main(void) {
        long long int i;
        pid_t rf;
        rf = fork();
        switch (rf) {
            case -1:
                printf("No he podido crear el proceso hijo. \n");
                break;
            case 0:
                printf("\t\tSoy Isaac, mi PID es %d y mi PPID es %d. \n", getpid(), getppid());
                signal(SIGUSR1, hijo);
                for (i=0; i<VUELTAS; i++);
                break;
            default:
                printf("\t\tSoy Abraham, mi PID es %d y el PID de mi hijo es %d. \n", getpid(), rf);
                signal(SIGUSR1, padre);
                sleep(1);//suspende el proceso 1 segundo.
                printf("\t\tVoy a matar a mi hijo.\n");
                sleep(15);//suspende el proceso 15 segundos.
                kill(rf, SIGUSR1);
                for (i=0; i<VUELTAS; i++);
        }
        exit(0);
    }
}
```

Compila el programa con la siguiente línea

```
$ gcc -Wall -o codigo13 codigo13.c
```

Ejecutamos el programa

```
$ ./codigo13
```

	<p align="center">UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<b>Formato:</b> Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		
<b>Aprobación:</b> 2022/03/01	<b>Código:</b> GUIA-PRLD-001	<b>Página:</b> 15

### III. EJERCICIOS/PROBLEMAS PROPUESTOS

#### EJERCICIO 1.

Crear un programa en C++ donde utilizando una llamada a sistema fork() se puedan crear tres hijos.

#### EJERCICIO 2.

Crear un programa en C++ donde utilizando la biblioteca csignal se pueda imprimir un mensaje de "hola" utilizando la señal SIGABRT

#### EJERCICIO 3.

Modifica el código 10.c para que mediante la utilización de la función "sleep()", la frecuencia a la que el proceso hijo escribe en los ficheros sea menor que la del proceso padre. Es decir que realice menos escrituras por unidad de tiempo.

#### EJERCICIO 4.

Crear un programa en C++ o C que permita comprobar que el proceso nieto se ejecuta antes que el proceso hijo y el proceso hijo se ejecuta antes que el proceso padre

### IV. CUESTIONARIO

- 1) ¿Cómo usted definiría la bomba fork?
- 2) ¿Para qué sirve la señal SIGXCPU?

### V. REFERENCIAS Y BIBLIOGRAFÍA RECOMENDADAS:

- [1] P.J. Deitel and H.M. Deitel, "Cómo Programar en C++", México, Ed. Pearson Educación, 2009  
 [2] B. Stroustrup, "El Lenguaje de Programación C++", Madrid, Addison Pearson Educación, 2002  
 [3] B. Eckel, "Thinking in C++", Prentice Hall, 2000

### TÉCNICAS E INSTRUMENTOS DE EVALUACIÓN

<b>TÉCNICAS:</b> Redacción escrita	<b>INSTRUMENTOS:</b> Rúbrica
---------------------------------------	---------------------------------

## CRITERIOS DE EVALUACIÓN

<b>Criterio de evaluación / Niveles de expectativa</b>	<b>Primer Criterio: Insatisfecho</b>	<b>Segundo Criterio: En Proceso</b>	<b>Tercer Criterio: Satisfactorio</b>	<b>Cuarto Criterio: Sobresaliente</b>
Informe  <b>Puntaje máx: 08</b>	El informe es difícil de leer y no cuenta con la información pedida.  <b>Puntaje: 0</b>	El informe incluye la mayor parte de la información solicitada, pero cuesta comprenderlo. <b>Puntaje: 2</b>	El informe incluye la información solicitada y es comprensible.  <b>Puntaje: 4</b>	El informe está claramente detallado e incluye toda la información solicitada <b>Puntaje: 8</b>
Cantidad de información  <b>Puntaje máx: 06</b>	Uno o más de los temas no han sido tratados  <b>Puntaje: 0</b>	Todos los temas han sido tratados y la mayor parte de las preguntas han sido contestadas, como mínimo con una frase. <b>Puntaje: 2</b>	Todos los temas han sido tratados y la mayor parte de las preguntas han sido contestadas, como mínimo dos frases cada una. <b>Puntaje: 4</b>	Todos los temas han sido tratados y todas las preguntas han sido contestadas con tres o más frases cada una.  <b>Puntaje: 6</b>
Calidad de información  <b>Puntaje máx: 06</b>	La información tiene poco que ver con el tema principal.  <b>Puntaje: 0</b>	La información está relacionada con el tema principal.  <b>Puntaje: 2</b>	La información está claramente relacionada con el tema principal.  <b>Puntaje: 3</b>	La información está claramente relacionada con el tema principal y presenta otros ejemplos. <b>Puntaje: 6</b>