Índice:

Introducción:

El objetivo de esta guía es primero presentar al alumno herramientas básicas para programar dentro de un ambiente MINIX en C (refrescando algunos conceptos de este lenguaje), para hacer luego hincapié en las distintas llamadas al sistema y sus usos reales, que serán demostrados con ejemplos.

Se recomienda (aunque no es requisito) tener una maquina virtual corriendo Minix mientras se sigue la guía, de esta manera el alumno podrá ir probando los distintos ejemplos propuestos.

Programación C en Minix

Creación del Programa

Para la creación del programa se puede usar cualquier editor de textos ordinario con le que se este familiarizado. En Minix están disponibles los editores vi y mined.
Por convención el nombre del archivo debe terminar con .c, por ejemplo: prueba.c. El contenido del archivo debe obedecer la sintaxis del lenguaje C.

Compilación

Existen muchos compiladores de C, en minix el compilador estándar es el cc. El comando deberá ser seguido por el nombre del programa en C que se quiere compilar. Un determinado número de opciones del compilador pueden ser indicadas también. Sin embargo, la mejor referencia de cada compilador es a través de las páginas en línea, del manual del sistema. Por ejemplo: *man cc*.

Por lo tanto, el comando básico de compilación es:

cc programa.c

Donde programa.c es el nombre del archivo.

Cuando el compilador ha terminado con éxito, la versión compilada o ejecutable, es dejado en un archivo llamado a.out.
Si la opción -o es usada con el compilador, el nombre después de -o es el nombre del programa compilado. Se recomienda usar la opción -o con el nombre del archivo ejecutable, como se muestra a continuación:

cc -o programa programa.c

De esta forma el archivo compilado queda con el nombre “programa” en lugar de a.out.

Ejecución

El siguiente paso es correr el programa ejecutable. Para correr un ejecutable en Minix simplemente se escribe el nombre del archivo precedido de ./ . Por ejemplo para correr el programa *ordenar* que se encuentra en la carpeta actual escribimos:

./ordenar

Programación en C (refrescando algunos temas)

Uso de argumentos en nuestros programas:

Cuando uno ejecuta un comando siempre suele pasar algún parámetro, como por ejemplo, el nombre de un archivo, un valor, etc.
Luego naturalmente nos surge la siguiente pregunta

¿Es posible trabajar con parámetros en nuestros programas (en C)? La respuesta es **SI**

¿Cómo? Veamos de que forma se puede trabajar con parámetros con un ejemplo:

```
int main(int argc, char *argv[])
{
        int x,y;

        x = atoi(argv[1]);
        y = atoi(argv[2]);

        if (argc != 3)
        {
                printf("Cantidad de parámetros incorrecta.\n");
                return 0;
        }

        printf("La suma de los parámetros es: %d\n", x + y);

        return 0;
}
```

¿Y esto qué significa? La función main recibe los parámetros en una estructura llamada argv[], donde cada posición de este vector representa un parámetro (La posición 0 representa el nombre del programa, la 1 el primer parámetro, la 2 el segundo y así). También disponemos de un entero llamado argc, el valor de argc nos informa cuantos parámetros recibió el programa.

Punteros a funciones:

¿Qué son? Los punteros a funciones son como su nombre lo indica “punteros”. Es decir variables que guardan como dato la dirección de memoria de una función dada.

¿Cómo se definen? La sintaxis para definir un puntero a una función en C es la siguiente:

*tipo-devuelto (*nombre-puntero)(parámetros);*

Veamos un ejemplo: Supongamos que disponemos de una función “sumar” que tiene el siguiente prototipo:

int sumar(int a, int b);

entonces podemos definir un puntero a función y asignarle la dirección de “sumar”:

```
int (*ptrSumar)(int, int);
…
ptrSumar = &sumar;
```

Ahora bien, para llamar a la función “sumar” podemos utilizar tanto la función en si como también utilizar el puntero “ptrSumar”. ¿de que forma? De la siguiente manera:

sumar(1,2);

o

*(*ptrSumar)(1,2);*

¿Cómo se pueden pasar como argumentos? Los punteros a funciones como toda variable se pueden pasar como argumentos a una función. Veamos un ejemplo:

Supongamos que tenemos una función llamada “operación_aritmetica” con el siguiente prototipo:

```
Int operación_aritmetica(int a, int b, int (*ptrFunc)(int,int));
```

Y se implementa de la siguiente manera:

```
Int operación_aritmetica(int a, int b, int (*ptrFunc)(int,int))
{
```

```
  return (*ptrFunc)(a,b);
}
```

Esta función puede ser llamada usando un puntero a “sumar” (del ejemplo anterior):

```
Int a;
a = operación_aritmetica(1,2,ptrSumar);
```

o sin necesidad de haber creado el puntero:

```
Int a;
a = operación_aritmetica(1,2,&Sumar);
```

¿Se pueden devolver? Si, como toda variable pueden ser devueltos por una función. Veamos la sintaxis con un ejemplo:

En este ejemplo desarrollaremos una función “devolverSumar()” que devuelve un puntero a la función “sumar” (del ej. Anterior).

```
Int (*devolverSumar(void))(int, int)
{
  return &sumar;
}
```

Luego podemos utilizar:

```
(*devolverSumar())(1,2);
```

que es lo mismo que invocar

```
sumar(1,2);
```

Como el alumno a esta altura este posiblemente un poco mareado, conviene definir un nuevo tipo “tipoSumar” para aclarar la sintaxis:

```
typedef int (*tipoSumar)(int, int);
```

Ahora podemos escribir “devolverSumar” como:

```
tipoSumar devolverSumar(void)
{ … }
```

Llamadas al sistema Minix

Las llamadas al sistema proveen la interfase entre los procesos de usuario y el sistema operativo.
Este conjunto de llamadas se pueden dividir en seis grupos:

- Gestión de procesos

pid = fork()	Crea un proceso hijo idéntico al proceso padre
s = wait(&status)	Espera a que un proceso hijo termine y determina su condición de salida
s = execve(name,argv,env)	Sustituye la imagen de un proceso
exit(staus)	Pone fin a la ejecución de un proceso y establece la condición de salida
size = brk(addr)	Fija el tamaño del segmento de datos que se direccionará en addr
pid = getpid()	Retorna el ID del proceso solicitante
pid = getppid()	Retorna el ID del proceso padre

- Gestión de archivos

fd = creat(name, mode)	Crea un nuevo archivo o trunca uno existente
fd = mknod(name, mode, addr)	Crea un nodo i
fd = open(file, how)	Abre un archive para lectura escritura o ambos
s = close (fd)	Cierra un archivo abierto
n = read (fd, buffer, nbytes)	Lee datos de un archivo en un buffer
n = write (fd, buffer, nbytes)	Escribe datos en un archive desde un buffer
pos = lseek(fd, offset, whence)	Mueve el puntero del archivo a una nueva posición
s = stat(name, &buff)	Devuelve información de estado del archivo
s = fstat(fd, buff)	Devuelve información de estado del archivo
fd = dup(fd)	Duplica el descriptor de archivo
s = pipe(&fd[0])	Crea una tubería
Ioctl(fd, request, argp)	Realiza operaciones especiales sobre archivos

- Gestión de señales

oldfunc = signal(sig, func)	Indica que la señal sig sea capturada y sea tratada por la rutina func
s = Kill(pid, sig)	Envía una señal sig al proceso pid
residual = alarm(seconds)	Planifica una señal SIGALARM después de cierto tiempo
s = pause()	Suspende la ejecución hasta la siguiente señal

- Gestión de directorios y sistema de archivos

s = link(name1, name2)	Crea una nueva entrada de directorio name2 para name1 (enlace)
s = unlink(name)	Elimina un enlace
s = mount(special, name, wflag)	Monta un sistema de archivo
s = umount(special)	Desmonta un sistema de archivo
s = sync()	Limpia los bloques reservados en la memoria del disco
s = chdir(dirname)	Cambia el directorio de trabajo
s = chroot(dirname)	Cambia el directorio raíz

- Protección

s =chmod(name,mode)	Cambia los bits de protección
uid = getuid()	Determina el user id del solicitante
gid = getgid()	Determina el group id del solicitante
s = setuid(uid)	Fija el user id
s = segid(gid)	Fija el group id
s = chown(name, owner, group)	Cambia el propietario y el grupo del archivo
oldmask = umask(complmode)	Fija una máscara que se utiliza para descubrir bits de protección

- Gestión del tiempo

seconds = time(&seconds)	Determina el tiempo transcurrido en segundos desde el 1/1/1970
s = stime(tp)	Fija el tiempo transcurrido desde el 1/1/1970
s = utime(file,timep)	Fija la hora del ultimo acceso al archivo
s = times(buffer)	Fija los tiempos de usuario y del sisema que se han usado hasta ahora

Llamada al sistema FORK()

Prototipo:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void)
```

Paginas del manual:

man 2 fork

Descripción:

El propósito de la llamada al sistema fork() es la creación de un nuevo proceso, el cual se convierte en el proceso hijo de quien realiza la llamada. Una vez que el hijo ha sido creado, ambos procesos ejecutarán paralelamente la instrucción que sigue inmediatamente a la llamada a fork(). Por ello, se debe distinguir a partir de este punto el comportamiento del hijo del comportamiento del padre. Esta diferencia se nota se nota en el valor que fork() devuelve al proceso padre e hijo. En el caso del Padre fork() devuelve el PID del proceso hijo, mientras que al hijo le devuelve el valor 0. En caso de fallar la llamada al sistema devuelve el valor -1.

Ejemplo del uso:

" Cree un proceso padre, que cree un proceso hijo y donde ambos impriman en pantalla 200 veces el mensaje "This line is from child/parent, value = 'nro. linea' ". Luego al finalizar mostraran respectivamente "*** Child/Parent process is done ***" ".

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>


#define MAX_COUNT 200

void ChildProcess(void); /* child process prototype */
void ParentProcess(void); /* parent process prototype */

int main(void)
{
        pid_t pid;
```

```
        pid = fork();
        if (pid == 0)
        {
                ChildProcess();
        }
        else
        {
                ParentProcess();
        }

        return 0;
}

void ChildProcess(void)
{
        int i;
        for (i = 1; i <= MAX_COUNT; i++)
                printf(" This line is from child, value = %d\n", i);

        printf(" *** Child process is done ***\n");
}

void ParentProcess(void)
{
        int i;
        for (i = 1; i <= MAX_COUNT; i++)
                printf("This line is from parent, value = %d\n", i);

        printf("*** Parent process is done ***\n");
}
```

Llamada al sistema WAIT()

Prototipo:

#include <sys/types.h>
#include <sys/wait.h>

*pid_t wait(int *status)*

Paginas del manual:

man 2 wait

Descripción:

La llamada al sistema wait() pausa a el proceso que la llama hasta que una señal es recibida o uno de sus procesos hijos termina. Esta llamada al sistema es muy útil a la hora de coordinar la ejecución del proceso padre e hijo en especial cuando el padre debe realizar una tarea luego de que culmina la ejecución de su hijo.

Ejemplo del uso:

"Hacer un programa padre que cree un programa hijo, espere que este muestre en pantalla "soy el hijo" y luego muestre "soy el padre""

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
        pid_t pid;
        int *i;

        pid = fork();

        if (pid == 0)
        {
                printf("soy el hijo\n");
        }
        else
        {
                wait(i);
                printf("soy el padre\n");
        }

        return 0;
}
```

Llamadas al sistema GETPID() y GETPPID()

Prototipo:

#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void)
pid_t getppid(void)

Paginas del manual:

man 2 getpid
man 2 getppid

Descripción:

Getpid() devuelve al proceso que llama su pid (Process ID)
Getppid() devuelve al proceso que llama el pid de su padre.

Ejemplo del uso:

"Hacer un programa padre que cree un programa hijo, espere que este muestre en pantalla "soy el hijo de: 'pid del padre' " y luego muestre "soy el padre de: 'pid del hijo' ""

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
        pid_t pid;
        int *i;

        pid = fork();

        if (pid == 0)
        {
                printf("soy el hijo de: %d\n", getppid());
        }
        else
        {
                wait(i);
                printf("soy el padre de: %d\n", pid);
        }

        return 0;
}
```

Llamada al sistema EXECVE()

Prototipo:

#include <sys/types.h>
#include <unistd.h>

*int execve (const char *filename, const char *argv [], const char *envp[])*

Paginas del manual:

man 2 execve

Descripción:

Execve() ejecuta el programa indicado por el parámetro filename. Este parámetro debe ser bien un binario ejecutable, o bien un shell script comenzando con una línea de la forma "#! intérprete". En el segundo caso, el intérprete debe ser un nombre de camino válido para un ejecutable que no sea él mismo un script y que será ejecutado como intérprete. El parámetro argv es un array de cadenas de argumentos pasados al nuevo programa. El parámetro envp es un array de cadenas, convencionalmente de la forma clave=valor, que se pasan como entorno al nuevo programa. Tanto argv como envp deben terminar en un puntero nulo. El vector de argumentos y el entorno pueden ser accedidos por la función "main" del programa invocado cuando se define como int main(int argc, char *argv[], char *envp[]).
Execve() no regresa en caso de éxito, y el código, datos y la pila del proceso invocador se reescriben con los correspondientes del programa cargado. El programa invocado hereda el PID del proceso invocador y cualquier descriptor de fichero abierto que no se haya configurado para "cerrar en ejecución" (close on exec). Las señales pendientes del proceso invocador se limpian. Cualquier señal capturada por el proceso invocador es devuelta a su comportamiento por defecto.

Ejemplo del uso:

"Cree un programa al que se la pasan como argumentos nombres de archivos, luego el programa haciendo uso de la llamada al sistema execve() debe llamar a 'cat' para que este los muestre en la salida estándar concatenados"

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[], char *envp[])
{
        execve("/usr/bin/cat", argv, envp);
        printf("Esto no se imprimira!!!");

        return 0;
}
```

Llamada al sistema DUP()

Prototipo:

#include <sys/types.h>
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);

Paginas del manual:

man 2 dup

Descripción:

Dup y dup2 crean una copia del descriptor de fichero oldfd.
Después de una llamada a dup o dup2 con éxito, los descriptores oldfd y newfd pueden usarse indistintamente. Comparten candados (locks), indicadores de posición de ficheros y banderas (flags); por ejemplo, si la posición del fichero se modifica usando lseek en uno de los descriptores, la posición en el otro también cambia.
Dup usa el descriptor libre con **menor numeración** posible como nuevo descriptor. Dup2 hace que newfd sea la copia de oldfd, cerrando primero newfd si es necesario.
Dup y dup2 devuelven el valor del nuevo descriptor, o -1 si ocurre algún error.

"Realice un programa que cree un archivo llamado test.txt, duplique el descriptor del mismo con el menor descriptor disponible y a continuación escriba en él, utilizando el nuevo descriptor, la palabra 'Hola'".

Ejemplo del uso:

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
        int fd, newfd;
        char cadena[4] = "Hola";

        fd = open("test.txt", O_CREAT + O_WRONLY);
        newfd = dup(fd);
        write(newfd, cadena, 4);
        close(fd);

        return 0;
}
```

Llamada al sistema PIPE()

Prototipo:

#include <sys/types.h>
#include <unistd.h>

int pipe(int descf[2]);

Paginas del manual:

man 2 pipe

Descripción:

Pipe crea una tubería o interconexión entre dos procesos. La llamada al sistema pipe crea un par de descriptores de ficheros, que apuntan a un nodo-í de una tubería, y los pone en el vector de dos elementos apuntado por descf. descf[0] es para lectura, descf[1] es para escritura.

En caso de éxito, se devuelve cero. En caso de error se devuelve -1 y se pone un valor apropiado en errno.

Ejemplo del uso:

"Cree un programa padre, el cual crea un programa hijo. Luego el hijo debe pasarle mediante un pipe el mensaje "Hola" al padre, quien lo debe mostrar en pantalla"

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
        pid_t pid;
        int tuberia[2];

        pipe(tuberia);
        pid = fork();

        if (pid)
        {
                char cadena[4];

                close(tuberia[1]);

                read(tuberia[0], cadena, 5);
                printf("%s", cadena);
        }
        else
        {
                close(tuberia[0]);

                write(tuberia[1], "Hola", 5);
        }

        return 0;
}
```

(Nota: se debe tener en cuenta que el mecanismo que proporcionan los pipes es uni-direccional, por ello un proceso solo escribe, mientras que el otro solo lee. A esto se debe que el padre cierre el pipe de escritura y el hijo cierre el pipe de lectura).

Redirección de entrada y salida estándar

Descripción:

En ocasiones resulta útil poder redireccionar la entrada o salida estándar de un proceso a algún archivo u hacia otro proceso. Para ver como esto puede ser logrado nos valdremos principalmente de dos llamadas al sistema: Dup y Pipe.

Cabe aclarar que tanto la entrada estándar de un proceso, como su salida estándar tienen asignados descriptores. Formalmente estos son:

0 -> Entrada Estándar
1 -> Salida Estándar
2 -> Error Estándar

Ejemplo, redireccionando la salida a un archivo:

En este caso la estrategia será la siguiente: Primero cerraremos el descriptor de la salida estándar (1) y haremos un dup del descriptor del archivo al cual queremos escribir. Como dup asigna el menor número disponible y dado que no hemos cerrado la entrada estándar (0), no tenemos dudas en que asignará el descriptor 1.

Veamos la tabla de descriptores del proceso mientras nuestra estrategia se lleva a cabo:

Al principio:

0	Entrada estándar
1	Salida estándar
N	Descriptor archivo

Luego de cerrar la salida estándar (close(1))

0	Entrada estándar
1	(vacío)
N	Descriptor archivo

Luego de duplicar el descriptor del archivo con (dup(descr))

0	Entrada estándar
1	Descriptor archivo
N	Descriptor archivo

Código:

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
        int d;

        d = open("test.txt", O_CREAT + O_WRONLY);

        close(1);
        dup(d);

        printf("Esto se escribirá en el archivo!!!");

        return 0;
}
```

Ejemplo, redireccionando la salida a un proceso hijo:

Nuevamente re-direccionaremos la salida del proceso, pero esta vez lo haremos desde un proceso hijo, hasta un proceso padre. Nuestra lógica será la misma que en el ejemplo anterior, solo que en esta vez trabajaremos con un descriptor de un pipe. Veamos el ejemplo de la sección de la llamada al sistema pipe modificado:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
        pid_t pid;
        int tuberia[2];

        pipe(tuberia);
        pid = fork();

        if (pid)
        {
                char cadena[4];

                close(tuberia[1]);

                read(tuberia[0], cadena, 5);
                printf("%s", cadena);
        }
        else
        {
                close(tuberia[0]);
                close(1);
                dup(tuberia[1]);

                printf("Hola");
        }

        return 0;
}
```

Nuevamente cerramos la salida estándar del hijo, y cuando realizamos dup(tuberia[1]) hacemos que tanto tuberia[1] como la salida estándar estén ligadas al pipe de escritura. Por lo que el printf("Hola") producirá la escritura de este mensaje en el pipe.

Llamada al sistema SIGNAL()

Prototipo:

#include <signal.h>

void (*signal(int signum, void (*manejador)(int)))(int);

Páinas del manual:

man 2 signal

Descripción:

La llamada al sistema signal instala un nuevo manejador de señal para la señal cuyo número es signum. El manejador de señal se establece como una función especificada por el usuario, o una de las siguientes macros:

SIG_IGN: No tener en cuenta la señal.

SIG_DFL: Dejar la señal con su comportamiento predefinido.

El argumento entero que se pasa a la rutina de manejo de señal es el número de la señal. Esto hace posible emplear un mismo manejador de señal para varias de ellas. Los manejadores de señales son rutinas que se llaman en cualquier momento en el que el proceso recibe la señal correspondiente. Usando la función alarm(2), que envía una señal SIGALRM al proceso, es posible manejar fácilmente trabajos regulares. A un proceso también se le puede decir que relea sus ficheros de configuración usando un manejador de señal (normalmente, la señal es SIGHUP).

signal devuelve el valor anterior del manejador de señal, o SIG_ERR si ocurre un error.

Ejemplo del uso:

"Cree un programa que al recibir la señal SIGUSR1 (User Defined Signal 1) muestre en pantalla "Llego la señal SIGUSR1!". El programa solo debe iterar sin realizar operaciones mientras espera la llegada de señales"

```
#include <signal.h>
#include <stdio.h>

void handler(int signum);

int main(void)
{
        signal(SIGUSR1, &handler);

        while(1) ;

        return 0;
}

void handler(int signum)
{
        printf("Llego la señal SIGUSR1!\n");
}
```

(Nota: para probar este ejemplo primero dejamos corriendo este proceso en una terminal, luego en la otra terminal obtenemos el pid de ese proceso (comando ps) y le enviamos la señal SIGUSR1 (10) al mismo para ver los resultados, (kill -10 'pid'). Para detener el proceso solo usamos el comando kill (kill 'pid'))

Llamada al sistema KILL()

Prototipo:

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Páginas del manual:

man 2 kill

Descripción:

La llamada al sistema kill envía a la señal 'sig' al proceso cuyo pid es 'pid'.

Ejemplo del uso:

"Cree dos procesos. Uno llamado Cajón y el otro llamado Manzanero. Los procesos no tienen ninguna relación entre ellos (padre/hijo), y Manzanero recibe como parámetro el PID de Cajón.

La tarea que debe realizar cada uno es la siguiente:

- Manzanero debe enviar manzanas al cajón cada dos segundos (para enviar una manzana envía una señal SIGUSR1). El manzanero envía un total de 5 manzanas y por cada envío muestra en pantalla el siguiente mensaje: "Recorcholis! Se me cayó una manzana!".
- Cajón debe estar atento a las manzanas que envía el Manzanero, y cada vez que recibe una manzana debe mostrar en pantalla el siguiente mensaje: "Que suerte tengo, al Manzanero se le cayó una Manzana! Ahora tengo 'n' Manzanas!"."

Cajón:

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

int manzanas = 0;

void recoger(int sig);
```

```
int main()
{
        signal(SIGUSR1, &recoger);

        while (manzanas != 5) ;

        return 0;
}

void recoger(int sig)
{
        manzanas++;
        printf("Que suerte tengo, al Manzanero se le cayo una Manzana!\n");
        printf("Ahora tengo %d Manzanas!\n", manzanas);
}
```

Manzanero:

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
        int pid_c, i;

        pid_c = atoi(argv[1]);

        for (i = 0; i < 6; i++)
        {
                printf("Recorcholis! Se me cayó una manzana!\n");
                kill(pid_c, SIGUSR1);
                sleep(2);
        }

        return 0;
}
```

(Nota: Para probarlo primero corremos en una terminal Cajón, luego en la otra terminal obtenemos su pid con el comando ps y a continuación ejecutamos manzanero 'pid').

Llamada al sistema PAUSE()

Prototipo:

```
#include <sys/types.h>
#include <unistd.h>

int pause(void);
```

Páginas del manual:

man 2 pause

Descripción:

La llamada al sistema pause() bloquea al proceso hasta la recepción de una señal. Se utiliza para no gastar ciclos de CPU mientras se espera la llegada de una señal. La llamada al sistema pause() solo retorna luego de haberse ejecutado un manejador de señal.

Ejemplo del uso:

(Nota: Tomaremos el ejemplo de la llamada al sistema signal, en ese ejemplo se utilizaba una estructura while (TRUE) que no realizaba ninguna acción para esperar una señal. Ahora que disponemos de la llamada al sistema Pause podemos dejar de malgastar ciclos)

```
#include <signal.h>
 #include <stdio.h>
void handler(int signum);

 int main(void)
{
        signal(SIGUSR1, &handler);

        while(1) pause();

       return 0;
}

void handler(int signum)
{
        printf("Llego la señal SIGUSR1!\n");
}
```

Llamada al sistema ALARM()

Prototipo:

```
#include <sys/types.h>
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Páginas del manual:

man 2 alarm

Descripción:

La llamada al sistema alarm() causa el envío de una señal SIGALRM al proceso que la invoca luego de una cierta cantidad de segundos que son pasados como argumentos.

Las alarmas no se colocan en una pila, esto significa que sucesivos llamados a alarm() resetean el tiempo de envío (al final se recibe una sola alarma con el último tiempo seteado). Si se pasa 0 como argumento se cancela automáticamente cualquier alarma.

Ejemplo del uso:

"Realice un programa cuya función sea setear una alarma para dentro de 5 segundos, y cuando llegue la misma muestre en pantalla: "Sonó la alarma!" y luego termine."

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void reloj(int i);

int main()
{
        signal(SIGALRM, &reloj);

        alarm(5);

        pause();

        return 0;
}

void reloj(int i)
{
        printf("Sonó la alarma!\n");
}
```