Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo - Buenos Aires - Argentina

Departamento de Ingeniería e Investigaciones Tecnológicas

Cátedra de Sistemas de Computación II

Jefe de Cátedra: Carlos Neetzel
Jefe de trabajos prácticos: Fabio Rivalta

Año: 2011
Versión: 1.04

Guía de práctica de laboratorio

Índice

Introducción

Objetivo

Facilitarle al alumno el seguimiento de las clases y la investigación necesaria para la realización de los trabajos prácticos de la materia.

Organización

La idea original es que cada capítulo de la guía se corresponda con una clase. Por supuesto el docente podría decidir dar más de un capítulo en una clase, utilizar varias clases para explicar un capitulo o cambiar totalmente el contenido de éstas.

Cada capitulo es independiente a los otros. Ningún capitulo es requisito de otro capitulo a menos que explícitamente se especifique lo contrario. Con esto queda claro que no es necesario leer esta guía de corrido, sino que podría ser utilizada como un manual de referencia para temas específicos tratados en cada capítulo.

Algunas convenciones

Cuando se intente resaltar una palabra dentro del texto, esta se escribirá en *letra cursiva*.

Los enlaces a páginas de internet, direcciones de correo o distintas secciones del documento se encontrarán en el texto en color azul y subrayadas.

Todos los ejemplos prácticos de código fuente, lineas de comando, salidas por pantalla, nombres de archivos, nombres de variables y afines estarán escritos en letra de paso fijo dos puntos más chica que el tipo de letra del texto. Por ejemplo:

```
jmfera   tty1     Apr 14 14:58
jmfera   tty2     Apr 14 15:16
root     tty3     Apr 14 18:30
```

En el caso de que figuren éstas palabras mezcladas en el texto también se encontrarán escritas con el mismo `tipo de letra pero de tamaño normal`.

Números de versiones

Las versiones cuyo número de mayor magnitud sea 0, por ejemplo '0.2' o '0.14', corresponden a guías que no tienen todos sus tópicos completados mínimamente.

A partir de la versión 1.0, el segundo número cambia con cada corrección o incremento de contenido menor y el primero cada vez que se elimine o agregue una clase completa.

Scripts

Objetivo

Que el alumno pueda comenzar a programar sus propios scripts de shell para automatizar tareas repetitivas de administración de sistemas Linux. La mayoría de lo aprendido aquí debería ser aplicable a otros sistemas de la familia Unix.

También se intenta que esta guía sirva de referencia rápida para algunas funciones avanzadas de uso frecuente en la administación de sistemas operativos.

Requerimientos previos

Es recomendable haber realizado el trabajo práctico 1 (Uso básico de GNU/Linux) para seguir el desarrollo de esta clase, aunque no necesario.

Desarrollo

Introducción

Los scripts que nos ocupan en esta clase son secuencias de instrucciones que puedan ser leídas por el intérprete de comandos `bash`. Estas instrucciones pueden ser simplemente archivos ejecutables que residen en el disco rígido (como `ls`, `named` o `java`) o comandos internos del bash (como `cd`, `echo` o `alias`).

Además de la potencia que le pueden dar a un script los distintos programas que vienen con el sistema operativo, éste puede contar con distintas estructuras de control como `for`, `while` o `if`, que se encuentran disponibles como comandos internos del bash.

Cuestiones básicas de la sintaxis

Un programa para bash es un archivo de texto que contiene una serie de comandos que pueden ser interpretados por éste.

Los comandos pueden ser separados por caracteres de nueva línea, es decir un ENTER, o bien por el caracter ; (punto y coma).

Todo lo que se escriba luego de un # y hasta que se encuentre un caracter de nueva linea, será considerado por el `bash` como un comentario y por lo tanto no será ejecutado. La única excepción a esta regla se explica en el siguiente capítulo y es cuando en los primeros dos caracteres del archivo escribo #!.

Las variables no se declaran. Se crean a medida que se les asigna un valor en el programa. El bash diferencia entre letras mayúsculas y minúsculas en los nombres de variables. La asignación de un valor a una variable se realiza mediante el operador = y no deben existir espacios a derecha o a izquierda de éste para que la operación tenga éxito. Veamos un ejemplo sencillo:

```
NOMBRE_ARCHIVO=salida.log
```

Para acceder al contenido de una variable, hay que anteponer un signo $ al nombre de ésta. Cuando el bash se encuentre con un nombre de una variable con el signo $ antepuesto, reemplazará el nombre de la variable por el contenido de esta. Ahora vamos a ver un ejemplo de como podemos verificar ésto directamente en el símbolo de sistema o prompt del bash:

```
$ echo $NOMBRE_ARCHIVO

$ NOMBRE_ARCHIVO=salida.log
$ echo NOMBRE_ARCHIVO
NOMBRE_ARCHIVO
$ echo $NOMBRE_ARCHIVO
salida.log
$ _
```

Como habrán podido ver, el comando echo sirve para mostrar un mensaje en pantalla... y se utiliza muchísimo en los scripts.

Especificando el interprete de comandos de nuestro script

Durante esta clase vamos a ver cómo realizar programas que pueden ser interpretados por el bash. El problema es que en el mundo UNIX, el bash no es el único interprete de comandos que existe, sino que es uno más así como el ksh o el csh.

Para que nuestro programa funcione como nosotros esperamos que lo haga, independientemente del interprete desde el que se lo ejecute, la primera linea de éste debe ser `#!/bin/bash`. Es decir que los caracteres #! al principio del archivo indican que a continuación viene el interprete de comandos necesario para ejecutar.

Otros ejemplos válidos podrían ser `#!/bin/csh` o `#!/bin/awk -f`.

Código de salida de un script y el comando exit

El código de salida de cualquier programa nos da información acerca de cómo resultó su ejecución. Normalmente, en un entorno UNIX, cuando un programa termina bien, es decir, sin que haya sucedido ningúna situación de error, éste devuelve un 0 (cero) cómo código de salida. Un ejemplo de una condición de error es una lectura inesperada de fin de archivo no manejada o un intento de apertura de un archivo sin los permisos necesarios para hacerlo. Hay programas que siempre que encuentran un error, devuelven como código de salida un 1 (uno) y hay otros programas que devuelven códigos específicos dependiendo del tipo de error. En resumen siempre es bueno salir con un código que indique cómo fue la ejecución para nuestro programa y la forma de hacerlo es a través del comando exit, pasandole como parámetro el código que queremos devolver.

La utilidad de este código de salida se ve sobre todo cuando nuestro programa está siendo llamado desde otro programa y necesitamos saber si las cosas funcionaron cómo esperábamos. Más adelante veremos que esto se evalúa fácilmente llamando al programa

desde una construcción `if` o `for` desde otro script.

En este momento ya estamos en condiciones de realizar nuestro primer script:

```
#!/bin/bash
# Este script imprime el mensaje “Hola mundo” en la pantalla y termina.

echo “Hola mundo”
exit 0
```

Este texto debe ser introducido en un archivo y lo único que faltará para poder ejecutarlo es darle los permisos necesarios.

Variables de shell y de entorno

Dentro de un programa `bash`, podemos acceder a muchas variables que nos dan distintos datos acerca del entorno de ejecución en el que nos encontramos.

Unas de las más usadas son `$0`, `$1`, `$2`, etc., que contienen al comando con el que se ejecutó el script y sus parámetros (en la forma y el orden en que fueron escritos).

Otro ejemplo es `$#`, que contiene la cantidad de parámetros con la que fue llamado el script. En `$@` se guarda una cadena con todos los parámetros enviados al script. La variable `$?` contiene el código de salida del último comando ejecutado. En `$PWD` se encuentra el directorio de trabajo actual del programa. En la variable `$BASH_VERSION` se guarda la versión de la instancia de bash que está ejecutando el script.

No es la idea de esta sección explicar todas estas variables. Como referencia podemos recurrir a la página del manual del `bash`, en la sección nombrada “Shell Variables”.

Comando if

Básicamente esta construcción sirve para tomar una acción u otra dependiendo de una condición, en donde las acciones pueden ser a su vez una serie de comandos de `bash`. Bueno, en realidad, para ser precisos, lo que evalua el `if` del `bash` es el código de salida del comando pasado como condición. En caso de que este código sea 0, el `if` evaluará la condición como verdadera; en caso contrario, para cualquier otro valor de salida, la condición se evaluará como falsa.

Un comando muy util para poder evaluar condiciones en conjunto con el `if` es el `test`. Este comando, luego de evaluar una condicion que le pasamos por parámetro, devuelve 0 en caso de verdadero y un valor distinto de 0 en caso contrario. Un par de ejemplos útiles:

```
test $VALOR -eq 12    # Devuelve 0 en caso de que la variable contenga el entero
                      # 12.

test $VALOR -ge 12    # Devuelve 0 en caso de que la variable contenga un entero
                      # mayor o igual a 12.

test $CADENA = “Pepe” # Devuelve 0 en caso de que la variable contenga la cadena
                      # de caracteres Pepe.

test -f salida.log    # Devuelve 0 en caso de que salida.log exista y sea un
                      # archivo regular (no por ejemplo un directorio)
```

No es la idea de esta guía dar todos los parametros del `test`. Para este proposito les recomiendo consultar la página del manual.

Ahora que tenemos todos los elementos necesarios, les dejo un ejemplo completo de un script simple que verifica si un archivo determinado existe o no:

```
#!/bin/bash

if test $# -ne 1
then
  echo Error
  echo Uso: $0 nombre_de_archivo
  exit 1
fi

ARCHIVO=$1

if test -e $ARCHIVO
then
  echo La salida del test fue $? y por lo tanto el archivo existe
else
  echo La salida del test fue $? y por lo tanto el archivo NO existe
fi

exit 0
```

Van a ver en muchos scripts que el `if` se utiliza en conjunto con el operador `[]`. Bueno, este operador funciona exactamente igual que el `test`. Por ejemplo la tercer linea del script anterior podría escribirse de la forma `if [ $# -ne 1 ]` y funcionaría exactamente igual. Es importante dejar un espacio *siempre* después del caracter `[` ya que si no el script nos mostrará un mensaje de error y no funcionará como se espera.

También existe un `test` extendido, que se utiliza poniendo una condición entre dobles corchetes. Por ejemplo: `[[ $VARIABLE -le 20 ]]`. El `test` extendido previene muchos errores de sintaxis debido a que funciona con condiciones más parecidas a las de otros lenguajes de programación. Por ejemplo se pueden utilizar los operadores `&&`, `||`, `>` y `<`. También en caso de que `$VARIABLE` tenga un valor nulo, el `test` extendido devuelve un valor falso, mientras que el `test` devuelve error debido a que utilizamos un operador binario con un solo operando.

Otro comando muy útil es el doble paréntesis. Dentro de dobles paréntesis se pueden poner condiciones, operaciones o bien asignaciones utilizando operadores y operandos aritméticos. En caso de las condiciones, tiene la misma sintaxis que el C y también la misma precedencia de operadores, poniendo a nuestra disposición los operandos `||`, `&&`, `<` y `>` por ejemplo. En el caso de las operaciones, podemos utilizar operadores como `+`, `-` o `/`; teniendo en cuenta que el comando `(())` devolverá verdadero (cero) en caso de que de la operación resulte un valor distinto de 0 y falso en caso contrario. En el caso de las asignaciones, siempre devolverá 0 (verdadero) a menos que la operación no se pueda hacer, por ejemplo, por estar intentando una división por 0. Vamos por algunos ejemplos:

```
(( 0 ))          # Devuelve 1 (falso).
(( 1 ))          # Devuelve 0 (verdadero).
(( 5 > 4 ))      # Devuelve 0 (verdadero).
(( 5-5 ))        # Devuelve 1 (falso).  Si hay duda, miren el primer ejemplo y
                 # comparen.
(( 5/2 ))        # Devuelve 0 (verdadero).  Esto se debe simplemente a que se pudo
                 # realizar la operación sin errores.
(( 1/0 ))        # Devuelve 1 (falso).  Se debe a que no se pudo realizar la
                 # operación porque se está intentando una división por 0.
(( VAR=VAR++ ))  # Devuelve 0 (verdadero).  Se debe a que no hubo problemas para
                 # realizar la operación.  Noten que no es necesario anteponer el $
                 # a la variable con este comando.
```

Comando for

Hay dos formas de utilizar un ciclo for en bash. Una de ellas es utilizando la estructura vieja de shell de unix:

```
for VARIABLE in lista; do; N*(comando;) done
```

Abajo les voy a dar un ejemplo entero para clarificar:

```
#!/bin/bash

for NOMBRE in juan pedro jose pablo
do
  echo -n $NOMBRE
  if [ $NOMBRE == pablo ]
  then
    echo
  else
    echo -n ", "
  fi
done

exit 0
```

Con esta sintáxis, la variable que figura en el encabezado, se va cargando con uno de los valores de la lista cada vez que entra a un ciclo del `for`. Esta sintaxis es especialmente útil combinada con un método de inserción de texto que nos proporciona bash al utilizar las comillas ``. Si dentro de un script escribo un comando dentro de estas comillas, el bash primero reemplazará dentro de la línea este comando por la salida estándar de la ejecución de éste. Una vez que esto esté terminado recién ejecutará la linea. Es algo así como un preprocesamiento. Me gustaría mostrarles a continuación un ejemplo.

```
#!/bin/bash

for ARCHIVO in `ls .`
do
  echo -n $ARCHIVO
  if [ -f $ARCHIVO ]
  then
    echo " es un archivo regular."
  else
    echo " NO es un archivo regular."
  fi
done

exit 0
```

Tal cómo podrán verificar en sus equipos, este escript realiza un ciclo por cada archivo que se encuentre en el directorio de trabajo del script.

Por defecto el caracter separador de la lista es cualquier espacio en blanco, caracter tabulador o caracter nueva linea. Esto se puede cambiar configurando la variable `IFS` con el caracter que queramos que sea el separador de campos de la lista. Un ejemplo de cuando esto sería útil es cuando queremos procesar los campos de una entrada del archivo de usuarios de unix. Cómo estos campos están separados por caracteres `:`, puede ser útil en este caso cambiar el valor de la variable `IFS` de la siguiente forma:

```
IFS=:
```

La otra forma de utilizar un ciclo `for` en bash es muy parecida a como se utiliza el for en el lenguaje C o C++:

```
for (( instrucción inicial; condicion; instrucción de fin de ciclo )); do;
N*(comando;) done
```

Dentro de los dobles paréntesis, las expresiones son evaluadas de la misma forma que en el operador `(( ))` que ya explicamos más atrás en esta sección. También se utiliza la misma sintaxis. En el próximo ejemplo, mostraremos un problema clásico que es el de necesitar hacer un ciclo una cantidad de veces determinada de antemano, para repetir alguna

operación. En el caso específico de éste, vamos a hacer que el usuario ingrese tres números por pantalla y vamos a calcular su promedio.

```
#!/bin/bash

CANTIDAD=3
SUMA=0

for (( CONTADOR=0; CONTADOR < CANTIDAD; CONTADOR++))
do
  echo -n "Por favor ingrese un número: "
  read VALOR

  (( SUMA+=VALOR ))
done
(( PROMEDIO=SUMA/CANTIDAD ))
echo "El promedio de los "$CANTIDAD" números ingresados es "$PROMEDIO"."

exit 0
```

Comandos ssh y scp

El comando `ssh` se utiliza para ejecutar un comando en una máquina remota. Cumple la misma función que su antecesor, el `rsh`, con la diferencia de que la comunicación se realiza en forma encriptada a través de un método de claves públicas y privadas. Lo podemos utilizar para ejecutar un único comando en una máquina remota y que nos arroje la salida de éste en nuestra consola, aunque también podríamos utilizarlo para abrir una sesión (ejecutar una instancia de bash) en la máquina remota que le especifiquemos.

El siguiente ejemplo, escrito de esta forma, intentará listar el archivo de usuarios de la máquina cuyo nombre en la red es "`maquina18`". Digo intentará, porque se conectará al equipo con el usuario `pepe` y para poder realizar el trabajo que le estoy encomendando, éste debe tener los permisos necesarios.

```
ssh pepe@maquina18 /bin/cat /etc/passwd
```

Cuando presionamos la tecla `enter`, el `ssh` nos pedirá la clave de `pepe` en la máquina remota y si todo ocurrió cómo debía, nos dará la lista de usuarios de ésta.

La forma en que mas veces vamos a utilizar este comando seguramente es simplemente para conseguir una consola en la máquina remota y trabajar sobre ella. Para lograr este objetivo simplemente debemos escribir la siguiente línea:

```
ssh pepe@maquina18
```

El sistema nuevamente nos pedirá la clave de `pepe` en `maquina18` y nos ejecutará el intérprete de comandos habitual de este usuario.

El comando `scp` se utiliza para copiar archivos de una máquina a otra. La comunicación también se realiza en forma encriptada y también tuvo un antecesor, el `rcp`. Por ejemplo, si nos queremos traer el archivo `passwd` de `maquina18` para poder trabajarlo localmente deberiamos ejecutar el comando de la siguiente forma:

```
scp pepe@maquina18:/etc/passwd ./passwd_maquina18
```

En el caso de querer copiar un archivo de la máquina local a la máquina remota, deberiamos ejecutar algo parecido a esto:

```
scp archivo_a_copiar.txt pepe@maquina18:/home/pepe
```

De esta forma, el archivo local se copiará al directorio home del usuario `pepe` en la máquina llamada `maquina18`.

Comando tar, realización de backups

El comando `tar` se utiliza generalmente para empaquetar muchos archivos en uno solo. Esto puede ser muy útil en muchas ocasiones, como puede ser la necesidad de realizar un backup de uno o varios directorios completos en un solo archivo. También es útil cuando tenemos muchos archivos para enviar por mail o copiar de una máquina a otra o cualquier otra operación en la que el tratamiento de los archivos por separado impliquen un aumento considerable de overhead.

Las operaciones más utilizadas del `tar` son las que nos permiten crear un archivo de backup conteniendo los archivos originales, consultar su contenido, anexar nuevos archivos, actualizar archivos que han sido modificados y por último extraer los archivos nuevamente depositandolos en el sistema. Ahora vamos a ver una serie de ejemplos que ilustrarán el uso básico del tar, utilizando siempre el modificador v de verbose (que significa algo así como exceso de detalles) para que nos muestre los archivos que está procesando.

Existen más operaciones para realizar con el comando `tar` y muchos modificadores además de los que vamos a ver en esta guía que se pueden consultar en la página del manual del comando.

```
$ tar -cvf guia.tar guia
guia/
guia/scripts/
guia/scripts/script4.sh
guia/scripts/script5.sh
guia/scripts/ej1.sh
guia/scripts/ej2.sh
guia/scripts/script3.sh
$ ls -la guia.tar
-rw-rw-r--  1 jmfera jmfera 10240 abr  2 22:24 guia.tar
```

Lo que hice en la consola, en el ejemplo transcripto anteriormente, es crear (para eso el parametro c) un archivo `guia.tar` el cual debia contener todo el contenido del directorio `guia`.

Ahora les voy a mostrar dos formas distintas de consultar el contenido del archivo `guia.tar`. Cómo podrán ver en lo único que se diferencian es en el modificador v (verbose) que en la primer orden no se encuentra y en la segunda sí.

```
$ tar -tf guia.tar
guia/
guia/scripts/
guia/scripts/script4.sh
guia/scripts/script5.sh
guia/scripts/ej1.sh
guia/scripts/ej2.sh
guia/scripts/script3.sh
$ tar -tvf guia.tar
drwxrwxr-x jmfera/jmfera     0 2009-03-09 22:08:34 guia/
drwxrwxr-x jmfera/jmfera     0 2009-04-02 22:08:56 guia/scripts/
-rwx------ jmfera/jmfera   180 2009-03-31 00:23:04 guia/scripts/script4.sh
-rwx------ jmfera/jmfera   392 2009-04-02 22:08:56 guia/scripts/script5.sh
-rwx------ jmfera/jmfera   277 2009-03-09 22:16:47 guia/scripts/ej1.sh
-rwx------ jmfera/jmfera   277 2009-03-11 21:54:27 guia/scripts/ej2.sh
-rwx------ jmfera/jmfera   151 2009-03-31 00:11:38 guia/scripts/script3.sh
```

Lo que vamos a hacer ahora es agregar un archivo nuevo al backup.

```
$ tar -rvf guia.tar puertos.txt
puertos.txt
$ tar -tvf guia.tar
drwxrwxr-x jmfera/jmfera     0 2009-03-09 22:08:34 guia/
drwxrwxr-x jmfera/jmfera     0 2009-04-02 22:08:56 guia/scripts/
-rwx------ jmfera/jmfera   180 2009-03-31 00:23:04 guia/scripts/script4.sh
```

```
-rwx------ jmfera/jmfera   392 2009-04-02 22:08:56 guia/scripts/script5.sh
-rwx------ jmfera/jmfera   277 2009-03-09 22:16:47 guia/scripts/ej1.sh
-rwx------ jmfera/jmfera   277 2009-03-11 21:54:27 guia/scripts/ej2.sh
-rwx------ jmfera/jmfera   151 2009-03-31 00:11:38 guia/scripts/script3.sh
-rw-rw-r-- jmfera/jmfera   208 2009-03-11 21:46:24 puertos.txt
```

Es muy útil la función de actualización que nos brinda el `tar`. Por medio de ésta, el comando solo copiará al backup un archivo en el caso de que este tenga una fecha de modificación posterior al del archivo que se ya se encuentra en el backup. De esta forma podemos crear backups incrementales. Para intentar clarificar, en el ejemplo que sigue intentaré actualizar el directorio `guia` sin haber hecho modificacion alguna en ningún archivo. Luego modificaré un archivo y volveré a realizar la operación. Ustedes podrán ver la diferencia en el comportamiento del sistema.

```
$ tar -uvf guia.tar guia
guia/
guia/scripts/
$ vi guia/scripts/script5.sh
$ tar -uvf guia.tar guia
guia/
guia/scripts/
guia/scripts/script5.sh
$ tar -tvf guia.tar
drwxrwxr-x jmfera/jmfera     0 2009-03-09 22:08:34 guia/
drwxrwxr-x jmfera/jmfera     0 2009-04-02 22:08:56 guia/scripts/
-rwx------ jmfera/jmfera   180 2009-03-31 00:23:04 guia/scripts/script4.sh
-rwx------ jmfera/jmfera   392 2009-04-02 22:08:56 guia/scripts/script5.sh
-rwx------ jmfera/jmfera   277 2009-03-09 22:16:47 guia/scripts/ej1.sh
-rwx------ jmfera/jmfera   277 2009-03-11 21:54:27 guia/scripts/ej2.sh
-rwx------ jmfera/jmfera   151 2009-03-31 00:11:38 guia/scripts/script3.sh
-rw-rw-r-- jmfera/jmfera   208 2009-03-11 21:46:24 puertos.txt
drwxrwxr-x jmfera/jmfera     0 2009-03-09 22:08:34 guia/
drwxrwxr-x jmfera/jmfera     0 2009-04-02 22:08:56 guia/scripts/
drwxrwxr-x jmfera/jmfera     0 2009-03-09 22:08:34 guia/
drwxrwxr-x jmfera/jmfera     0 2009-04-02 22:40:15 guia/scripts/
-rwx------ jmfera/jmfera   396 2009-04-02 22:40:15 guia/scripts/script5.sh
```

Y finalmente solo nos queda averiguar cómo se recuperan los archivos que residen en nuestra copia de seguridad. Primero voy a mostrar cómo se recupera un archivo puntual del backup y luego cómo recuperar todo el árbol de archivos a la vez.

```
$ mkdir restore
$ cd restore
$ ll
total 0
$ tar -xvf ../guia.tar puertos.txt
puertos.txt
$ ll
total 4
-rw-rw-r--  1 jmfera jmfera 208 mar 11 21:46 puertos.txt
$ tar -xvf ../guia.tar
guia/
guia/scripts/
guia/scripts/script4.sh
guia/scripts/script5.sh
guia/scripts/ej1.sh
guia/scripts/ej2.sh
guia/scripts/script3.sh
puertos.txt
guia/
guia/scripts/
guia/
guia/scripts/
guia/scripts/script5.sh
$ ll
total 8
drwxrwxr-x  3 jmfera jmfera 4096 mar  9 22:08 guia
-rw-rw-r--  1 jmfera jmfera  208 mar 11 21:46 puertos.txt
$ ll guia/scripts/
total 20
-rwx------  1 jmfera jmfera 277 mar  9 22:16 ej1.sh
-rwx------  1 jmfera jmfera 277 mar 11 21:54 ej2.sh
-rwx------  1 jmfera jmfera 151 mar 31 00:11 script3.sh
```

```
-rwx------  1 jmfera jmfera 180 mar 31 00:23 script4.sh
-rwx------  1 jmfera jmfera 396 abr  2 22:40 script5.sh
```

Redirecciones de entrada y salida estándar

El bash relaciona la entrada estándar, la salida estándar y la salida por error estándar con los descriptores de archivo 0, 1 y 2 respectivamente. Estos flujos pueden ser redireccionados a través de una notación específica dentro de bash.

Quizá la redirección más utilizada es la de enviar la salida estándar a un archivo. Hay comandos que pueden estar mucho tiempo escribiendo en la consola o bien lo que escriben en la consola puede ocupar muchísimas pantallas. En estos casos es útil que en vez de escribir en la pantalla, el comando escriba en un archivo. Nuestro intérprete de comandos nos permite realizar esto con el operador `[n]>` atrás del comando y seguido del nombre de archivo en el cual se guardará la salida. El número antes del operador de redirección especifica el descriptor de archivo a direccionar, que al no ser especificado, el sistema toma por defecto el 1 que es la salida estandar.

```
$ find $HOME > lista_mis_archivos.txt
$ ll lista_mis_archivos.txt
-rw-rw-r--  1 jmfera jmfera 1836 abr  2 23:20 lista_mis_archivos.txt
```

En el ejemplo anterior ejecutamos un comando que lista todos los archivos (incluyendo directorios y archivos especiales) que se encuentran en nuestro directorio home. Es muy probable que para cualquier usuario, la salida de este comando sea mucha información, por lo tanto la guardamos en un archivo para después poder revisarla o bien reprocesarla.

También podríamos direccionar la salida por error a otro archivo, en vez de que salga por pantalla. En el caso anterior, si se diera el caso de que no tenemos permiso de lectura en algún directorio, el comando mostraría el error por pantalla, a menos que lo escribamos en un archivo de la siguiente forma:

```
find $HOME > lista_mis_archivos.txt 2> errores.txt
```

También es probable que queramos direccionar tanto la salida estándar como la salida por error al mismo archivo para que se grabe de la misma forma que se vería por pantalla. Esto se puede lograr escribiendo:

```
find $HOME > lista_mis_archivos.txt 2>&1
```

O bien un atajo para conseguir el mismo comportamiento:

```
find $HOME &> lista_mis_archivos.txt
```

Si queremos, en vez de crear un nuevo archivo con la salida de nuestro comando, agregar la informacción (append) al final del archivo sin borrarlo, debemos simplemente reemplazar el operador > por el >>.

También podemos reemplazar la entrada estándar de un comando con el contenido de un archivo con el operador <. Por ejemplo podríamos hacer algo así:

```
$ cat < errores.txt
find: /home/jmfera/guia/scripts/dir_no_permiso: Permiso denegado
```

Tengan en cuenta que el archivo errores.txt es el que generé en el ejemplo anterior del find.

Con respecto a la entrada estándar también la podemos direccionar para que el comando la tome desde un documento o una cadena en linea con los operadores << y <<< respectivamente. Les paso un par de ejemplos para que prueben en sus máquinas.

```
$ ll
total 0
$ cat > archivo.txt <<FIN
> Este es un archivo nuevo.
> Está siendo generado con la redirección de un documento.
> FIN
$ ll
total 4
-rw-rw-r--  1 jmfera jmfera 83 abr  5 17:06 archivo.txt
$ cat archivo.txt
Este es un archivo nuevo.
Está siendo generado con la redirección de un documento.


$ cat > archivo.txt <<<"Este es un archivo nuevo.  Está siendo generado con la
redirección de una cadena de caracteres."
$ cat archivo.txt
Este es un archivo nuevo.  Está siendo generado con la redirección de una cadena
de caracteres.
```

Dispositivo /dev/null

El dispositivo nulo opera como un operador nulo para todas las operaciones con las cuales es compatible. Un ejemplo de uso bastante corriente de éste es cuando no nos interesa ver ni guardar la salida por error de un comando determinado.

```
$ find $HOME > salida.txt 2>/dev/null
```

El resultado es simplemente que la salida por error se pierde, con lo cual no nos enteraremos de los directorios en los cuales no tenemos permiso de lectura.

```
$ cp /dev/null archivo_vacio.txt
```

Con el comando anterior, creamos un nuevo archivo vacío. El efecto se logra con dos reglas muy sencillas: Todos los bytes que se escriban en /dev/null serán desechados y cualquier lectura del dispositivo devolverá EOF.

Utilización de funciones

Podemos utilizar funciones en un script de bash al igual que en muchos lenguajes de programación. La sintaxis es un poco parecida a la de las funciones del lenguaje C.

```
funcion1 () {
  comando1
  comando2
  …
  comandon
}
```

Las funciones se invocan luego como si fueran un comando cuyo nombre es el nombre de la función. La única restricción a esto es que la funcion completa (no su encabezado) debe figurar en el script antes de que sea llamada para que funcione. Abajo un ejemplo sencillo pero completo de un script que utiliza una función:

```
#!/bin/bash

saludar () {
  echo !Hola a todos!
}

saludar
```

A la función le podemos pasar parámetros para que utilice en su procesamiento, los cuales siempre se deben poner a continuación de la llamada a ésta separados por espacios en blanco. Luego, dentro de la función se puede hacer referencia a éstos tal como hacemos referencia a los parámetros del script desde dentro de éste, es decir con $1, $2, ${22}, etc.

```
#!/bin/bash

saludar () {
  echo !Hola $1!
  echo !Hola $2!
}

saludar Pepe José
```

Los parámetros pasados al script no son visibles dentro de la función, con lo cual si necesitamos esa funcionalidad, la vamos a conseguir de la siguiente forma:

```
#!/bin/bash

saludar () {
  echo !Hola $1!
}
saludar $1
```

Una función también puede devolver un código de salida de la misma forma que lo hace un comando a través de la orden return:

```
#!/bin/bash

contabilizar () {
  du --max-depth=0 $1
  return $?
}

contabilizar $1
```

Fuentes

Página del manual de bash, test, ssh, scp, tar y null.

Advanced Bash Scripting Guide 5.6
Mendel Cooper
http://www.tldp.org/guides.html

AWK

Objetivo

Que el alumno sepa confeccionar programas sencillos en el lenguaje de programación AWK y hacerlos trabajar en conjunto con sus scripts.
También en esta clase se hace una introducción a expresiones regulares.

Requerimientos previos

Ninguno, aunque es conveniente estar capacitado para manejar en forma básica el intérprete de comandos bash.

Desarrollo

Llamada al intérprete

Existen muchas formas de ejecutar un programa AWK. Todas ellas incluyen la llamada al intérprete, que se encuentra en nuestro sistema en un ejecutable llamado `awk` o `gawk`. Cuando los programas son muy sencillos o cortos, es muy útil poder escribirlos en la misma linea de comandos en que se llama al intérprete. La sintaxis sería algo así: `awk [parámetros] 'programa' /camino/al/archivo.ext /camino/al/archivo2.ext2`. A continuación les paso un ejemplo que pueden probar en sus consolas.

```
$ cat /etc/modprobe.conf
alias scsi_hostadapter megaide
alias eth0 e1000
alias eth1 e1000
alias scsi_hostadapter1 ata_piix
alias usb-controller ehci-hcd
alias usb-controller1 uhci-hcd
alias dev17132 e1000
alias eth2 e1000
$ awk '{print $2}' /etc/modprobe.conf
scsi_hostadapter
eth0
eth1
scsi_hostadapter1
usb-controller
usb-controller1
dev17132
eth2
$ awk '{print $2, $3}' /etc/modprobe.conf
scsi_hostadapter megaide
eth0 e1000
eth1 e1000
scsi_hostadapter1 ata_piix
usb-controller ehci-hcd
usb-controller1 uhci-hcd
dev17132 e1000
eth2 e1000
$ awk '{print $2 "=" $3}' /etc/modprobe.conf
scsi_hostadapter=megaide
```

```
eth0=e1000
eth1=e1000
scsi_hostadapter1=ata_piix
usb-controller=ehci-hcd
usb-controller1=uhci-hcd
dev17132=e1000
eth2=e1000
```

En el caso de que especificáramos más de un archivo al final de la linea de comandos, el programa procesaría el primero y al finalizar seguiría procesando el otro. En caso de que no especificáramos archivo a procesar, AWK intentará procesar la entrada estándar, lo cual nos permite hacer uso de las redirecciones de flujos que nos proporciona el bash. Un ejemplo de ejecución sería:

```
$ awk '{print $2}' < /etc/modprobe.conf
scsi_hostadapter
eth0
eth1
scsi_hostadapter1
usb-controller
usb-controller1
dev17132
eth2
```

Por último, les voy a comentar un par de parámetros del comando AWK que se utilizan bastante. Para ver una lista completa de estos, revisen la página del manual de éste.

Uno de éstos sirve cuando nuestro programa ya es un poco más complicado o largo y queremos guardarlo en un archivo para no tener que reescribirlo cada vez que lo necesito. Para esto escribo `awk -f nombre_archivo.awk` en la linea de comandos. También podemos hacer un programa ejecutable awk si en la primera linea de un archivo de texto escribimos `#!/bin/awk -f` y luego modificamos sus permisos de lectura y ejecución convenientemente.

Los separadores de campos que toma por defecto el AWK (que algunos ya se habrán dado cuenta que se referencian de alguna manera con $1, $2, etc.) son los espacios y tabuladores. En caso de querer cambiar esto por cualquier otra cadena, lo debo hacer con la opción `-F`, por ejemplo de la siguiente forma: `awk -F ":"`.

Por último, otro parámetro interesante es uno que nos permite crear una variable inicializada a un determinado valor antes de empezar a ejecutar el programa. Esto es muy útil cuando se intenta concatenar la potencia de un script de bash con un programa de AWK, ya que muchas veces, el valor de una variable es conocida en el entorno de ejecución bash y necesitamos que también lo sea en nuestro entorno de ejecución AWK. La forma de lograrlo es con el parámetro `-v variable=valor`. A continuación les paso un ejemplo de cómo utilizar estos parámetros.

```
$ getent passwd | head
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
$ cat homes.awk
BEGIN {
        print "Usuario          Directorio HOME"
}

{
        if ($1 == usuario)
          print $1"             "$6
}
$ getent passwd | head | awk -F ":" -v usuario=root -f homes.awk
```

```
Usuario         Directorio HOME
root            /root
$ getent passwd | head | awk -F ":" -v usuario=lp -f homes.awk
Usuario         Directorio HOME
lp              /var/spool/lpd
```

Bloques principales

Los programas de AWK pueden estar compuestos por hasta 3 tipos de bloques principales. Un primer bloque encerrado entre llaves y precedido por la cadena BEGIN, un segundo bloque (vamos a ver más adelante que este se puede repetir varias veces en un programa), también encerrado entre llaves pero que puede estar (o no) precedido por una expresión regular encerrada entre dos barras (/expresión/) y un último bloque igual al primero pero precedido por la cadena END.

```
BEGIN {
  ...
}
{
  ...
}
.
.
.
{
  ...
}
END {
  ...
}
```

Todas las sentencias que se encuentren en la primer construcción van a ejecutarse una única vez al inicio del programa, antes de comenzar a procesar el archivo de entrada. Las sentencias que se encuentren en el segundo bloque, serán ejecutadas una vez por cada línea del archivo de entrada. Si hay más de un bloque de este tipo se ejecutará a continuación del anterior en cada línea. Por último, las instrucciones que se encuentren en el último bloque precedido por END, serán ejecutadas por única vez cuando se hayan terminado de procesar todos los registros del archivo de entrada.

Para ejemplificar les muestro a continuación un programa muy sencillo.

```
$ cat bloques.awk
#!/bin/awk -f

BEGIN {
  print ""
  print "Usuario     \tEntrada al sistema"
  print "=======     \t=================="
  contador=0
}
{
  print $1"   \t"$3" "$4" "$5
  contador++
}
END {
  print ""
  print "Hay " contador " usuarios en el sistema."
  print ""
}
$ who | ./bloques.awk

Usuario         Entrada al sistema
=======         ==================
federico        May 30 11:05
juanma          May 31 21:11

Hay 2 usuarios en el sistema.
```

```
$
```

Estructuras de control

En awk, una sentencia puede estar terminada en un carácter nueva linea o un punto y coma. Los grupos de sentencias están agrupados por llaves { … } al igual que en el lenguaje C.

Una expresión, al igual que una variable en este lenguaje, puede ser de tipo numérica o cadena de caracteres.

En un contexto de selección booleana, como en un if o un while, una expresión de cadena de caracteres es evaluada como verdadera siempre y cuando su resultado no sea la cadena vacía “”. A su vez una expresión numérica es verdadera si y solo si su resultado no es cero. A continuación les muestro un ejemplo para ver mejor este aspecto puntual.

```
$ cat expresiones_booleanas.awk
#!/usr/bin/awk -f
BEGIN {
       if ( 0 ) print "Por acá no pasa";
       if ( 4 - 4 ) print "Por acá tampoco";
       if ( 1 ) print "Por acá si";
       if ( 1234.99 ) print "Por acá también";
       if ( "Pepe" ) print "También se muestra porque no es una cadena vacía";
       if ( "" ) print "Esta no";
}
$ ./expresiones_booleanas.awk
Por acá si
Por acá también
También se muestra porque no es una cadena vacía
$
```

Cualquier comparación del tipo (expresión operador expresión) tiene como resultado 1 o 0, según la comparación sea verdadera o falsa respectivamente.

Dados ya estos conceptos, les voy a transcribir una tabla con las distintas sentencias que controlan el flujo de un programa en este lenguaje.

Uso	Formato
En caso de que la expresión sea verdadera, ejecuta el grupo de sentencias	if (expresión) sentencia
Ídem anterior, y en caso de que la expresión sea falsa, ejecuta el segundo grupo de sentencias	if (expresión) sentencia else sentencia
Ejecuta el grupo de sentencias siempre y cuando la expresión sea evaluada como verdadera	while (expresión) sentencia
Ídem anterior, pero la primera vez siempre ejecuta el grupo de sentencias	do sentencia while (expresión)
Ejecuta las sentencias siempre que la segunda expresión del for sea verdadera. Funciona igual al for del lenguaje C	for (expresión optativa; expresión optativa; expresión optativa) sentencia
Ejecuta el grupo de sentencias una vez por cada valor del vector, instanciando la variable al valor del actual del vector	for (variable en vector) sentencia
Corta la ejecución del bloque en un determinado lugar y vuelve a evaluar la condición	continue
Corta la ejecución del bloque y sale del ciclo	break

Variables incorporadas

El awk nos provee dentro del entorno de ejecución de algunas variables muy útiles para saber cosas de éste o controlar cómo queremos que se comporte nuestro programa. A continuación les voy a mostrar algunas que considero importantes, y los invito a consultar la página del manual del intérprete para consultar otras.

Nombre	Descripción
ARGC	Número de parámetros pasados en la línea de comandos.
ARGV	Vector que contiene los parámetros.
FILENAME	Nombre de archivo de entrada actual.
RS	Separador de registros de entrada. Por defecto es el carácter nueva línea.
ORS	Separador de registros de salida. Por defecto es el carácter nueva línea.
FS	Separador de campos de entrada.
OFS	Separador de campos de salida.
OFMT	Formato para números de salida.
NR	Número de registro que está siendo procesado actualmente.
NF	Número de campos que contiene el registro actual.

Ahora un ejemplo para que prueben en sus consolas cómo funciona.

```
$ cat variables.awk
#!/usr/bin/awk -f
BEGIN {
        print ""
        print "Número        Parámetro"
        print "======        ============================"
        for (i=0; i<ARGC; i++)
                printf "%6d        %s\n", i, ARGV[i];
        print""
        ARGC=2
        ARGV[1]="/etc/passwd"
}
{
        if(NR==1)
                print "Las dos primeras lineas del archivo " FILENAME " son..."

        if (NR<=2)
                print
}
$ ./variables.awk cualquier cosa

Número        Parámetro
======        ============================
     0        awk
     1        cualquier
     2        cosa

Las dos primeras lineas del archivo /etc/passwd son...
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

Registros y campos

Los registros son leídos del flujo de entrada y asignados a la variable $0. Los campos que componen al registro se guardan en las variables $1, $2, $3, ..., $NF. Recordemos que la variable NF está siempre instanciada con la cantidad de campos que contiene el registro.

La variable NR contiene el número de registro que se está procesando en este momento, es decir, el que está contenido en $0. Abajo les copio un ejemplo para que vean estos conceptos.

```
$ cat archivo_entrada.txt
pepe jose pedro juan
mateo javier andres pablo
$ cat registros_campos.awk
#!/usr/bin/awk -f
BEGIN {
       print "Registro       Campo1  Campo2  Campo3  Campo4  OtrosCampos"
       print "--------       ------  ------  ------  ------  -----------"
}
{
       print NR"             "$1"    "$2"    "$3"    "$4"    "$5
}
$ cat archivo_entrada.txt |./registros_campos.awk
Registro       Campo1  Campo2  Campo3  Campo4  OtrosCampos
--------       ------  ------  ------  ------  -----------
1              pepe    jose    pedro   juan
2              mateo   javier  andres  pablo
$
```

Operadores

Los operadores son prácticamente los mismos que en el lenguaje C. Abajo les transcribo una tabla con todos los operadores, en orden de precedencia descendente. La precedencia puede ser modificada utilizando paréntesis.

Asignación	= += -= *= /= %= ^=
Condicional	? :
O lógico	||
Y lógico	&&
Pertenencia a un vector	in
Coincidencia	~ !~
Relación	< > <= >= == !=
Adición	+ -
Multiplicación	* / %
Signo	+ -
No lógico	!
Exponente	^
Incremento	++ -- (post y pre)
Campo	$

Vectores

En awk podemos utilizar vectores. Éstos son vectores asociativos, debido a que se indizan a través de cadenas de caracteres. Esto quiere decir que puedo ponerle nombres en vez de números a los componentes del vector. En el ejemplo vamos a mostrarlo más claro.

Se puede utilizar la construcción for de una forma parecida a la que está disponible en bash, de forma tal que entre al ciclo una vez por cada valor que contenga el vector. En cada ciclo la variable del encabezado va tomando uno de los indices del vector, permitiendo luego acceder a los valores que éste contenga.

```
for (variable in vector) sentencia
```

Se puede utilizar la sentencia `delete` para borrar una o todas las entradas de un vector. Escribiendo `delete vector[indice]` borrará un solo elemento del vector. Escribiendo `delete vector`, en cambio, borrará todo el contenido de este.

Abajo les transcribo un ejemplo para mostrar el funcionamiento de estos conceptos.

```
$ cat vectores.awk
#!/usr/bin/awk -f
{
        nombres["1"]=$1
        nombres["1b"]=$2
        nombres[2]=$3
        nombres["anteultimo"]=$4
        nombres["ultimo"]=$5

        for (indice in nombres)       print nombres[indice]
        print ""
        for (indice in nombres)       print indice
        print ""
        print nombres[2]
        print nombres["ultimo"]
        print ""
        if ( "ultimo" in nombres ) print "El último elemento está en el vector"
        else print "El último elemento no está en el vector"
        print ""
        delete nombres["ultimo"]
        if ( "ultimo" in nombres ) print "El último elemento está en el vector"
        else print "El último elemento no está en el vector"
        print ""
        for (indice in nombres)       print nombres[indice]
}
$ echo juan manuel pedro pablo Jose|./vectores.awk
manuel
pablo
juan
pedro
Jose

1b
anteultimo
1
2
ultimo

pedro
Jose

El último elemento está en el vector

El último elemento no está en el vector

manuel
pablo
juan
pedro
$
```

Expresiones regulares

Una expresión regular es una forma de describir un grupo de cadenas de caracteres sin necesariamente enumerar todos sus elementos. Puede definirse como un patrón de cadenas de caracteres.

Por ejemplo, la expresión regular “p[io]pa” coincide con las cadenas “pipa” y “popa”. La expresión regular “^Pepe” hace referencia a cualquier cadena de caracteres que comience con “Pepe”.

A continuación les muestro una lista de operadores con los que se pueden construir expresiones regulares.

Expresión regular	Uso
c	Coincide con cualquier carácter, siempre y cuando no sea un carácter reservado (por ejemplo un operador.
\c	Coincide con el carácter literal c. Es útil esta forma cuando queremos incluir en nuestra expresión un carácter como & o |, que son operadores.
.	Cualquier carácter. Incluido el carácter nueva línea.
^expresión	Cualquier cadena que comience con “expresión”.
expresión$	Cualquier cadena que termine con “expresión”.
[abc...]	Cualquiera de los caracteres abc...
[^abc...]	Coincide con cualquier carácter excepto abc...
expresión1|expresión2	Coincide tanto con las cadenas representadas por la expresión1 como con las representadas por la expresión2.
expresión+	Coincide si se encuentra con una o más ocurrencias de la expresión.
expresión*	Coincide si se encuentra con ninguna o mas ocurrencias de la expresión.
expresión?	Coincide si se encuentra con una o ninguna ocurrencia de la expresión.
()	Los paréntesis se utilizan para cambiar la precedencia de los operadores.
expresión{n}	La expresión debe estar repetida en la cadena de caracteres n veces.
expresión{n,}	La expresión debe estar repetida en la cadena por lo menos n veces.
expresión{n, m}	La expresión debe estar repetida por lo menos n veces y como máximo m veces.
\w	Coincide con cualquier carácter que pueda constituir una palabra. Es decir, una letra, un dígito o un guión de subrayado (bajo).
\W	Coincide con cualquier carácter que no pueda constituir una palabra.
[:alnum:]	Carácter alfanumérico.
[:alpha:]	Carácter alfabético.
[:blank:]	Espacio o tabulador.
[:digit:]	Carácter numérico.
[:lower:]	Carácter alfabético minúscula.
[:print:]	Carácter imprimible, es decir que no sea un carácter de control.
[:punct:]	Carácter de puntuación, como un punto y coma, una coma o un punto.
[:space:]	Espacios, tabuladores, caracteres nueva linea o cualquier otro carácter que represente un espacio en blanco.
[:upper:]	Carácter alfabético mayúscula.

La forma mas común en las que podemos utilizar una expresión regular es en una construcción if para saber si una cadena coincide con una expresión regular. Para hacer esto el awk nos proporciona el operador ~. Las expresiones regulares en awk deben estar contenidas entre dos barras (/). Por ejemplo, si quiero saber si una determinada cadena de caracteres empieza con una A o una a, debo escribir:

```
if (cadena ~ /^[Aa]/) sentencia;
```

Si deseo saber si el registro actual contiene la cadena “pepe”, “Pepe” o “PEPE”; en cualquier lugar de éste, debo escribir:

```
if ($0 ~ /[pP]epe|PEPE/) sentencia;
```

Otra forma en la que se utilizan las expresiones regulares es para filtrar qué registros

procesar en nuestro programa y cuales no. Para esto simplemente debo poner la expresión regular antes de las llaves de la construcción que se ejecutará por cada registro. Abajo les muestro un ejemplo de esto.

```
$ cat archivo_entrada_2.txt
El pez por la boca muere.
Más vale pájaro en mano que cien volando.
Si el pez no tuviera boca tampoco podría vivir
La curiosidad mató al gato.
$ cat expresiones_regulares.awk
#!/usr/bin/awk -f
/^La/ { print "La linea " NR " comienza con \"La\"." }
/^El/ { print "La linea " NR " comienza con \"El\"." }
/\.$/ { print "La linea " NR " termina con un punto." }
$ cat archivo_entrada_2.txt | ./expresiones_regulares.awk
La linea 1 comienza con "El".
La linea 1 termina con un punto.
La linea 2 termina con un punto.
La linea 4 comienza con "La".
La linea 4 termina con un punto.
```

Funciones incorporadas

El lenguaje awk tiene una biblioteca de funciones concebidas para el manejo de cadenas de caracteres y operaciones aritméticas. Abajo les muestro una tabla con algunas de las funciones más utilizadas. En la página del manual de awk pueden encontrar más de estas funciones. También se pueden dirigir al manual oficial de gawk para una lista más completa de estas.

Función	Explicación
gsub(r, s, t) gsub(r, s)	Una función para sustitución “global”. Cada ocurrencia de la expresión regular r en la variable t será reemplazada por la cadena s. Si no escribo el parámetro t, se usará $0.
index(s, t)	Si t es una cadena de caracteres dentro de la cadena s, ésta función devuelve la posición donde comienza t. Hay que tener en cuenta que el primer carácter de s es el 1 y no el 0. Si t no se encuentra en la cadena s, la función devuelve 0.
length(s)	Devuelve el tamaño en caracteres de s.
match(s, r)	Funciona igual que index(), solo que r puede ser una expresión regular.
sprintf(formato, lista-de-valores)	Funciona de la misma forma que el printf del lenguaje C.
sub(r, s, t)	Igual que gsub(), pero solo sustituye la primer ocurrencia.
substr(s, i, n) substr(s, i)	Devuelve la porción de la cadena s que comienza en la posición i y termina en la posición n. Si no se especifica n, devuelve desde i hasta el final de la cadena.
tolower(s)	Devuelve una copia de s, con todas las mayúsculas convertidas a minúsculas.
toupper(s)	Devuelve una copia de s, con todas las minúsculas convertidas a mayúsculas.
int(x)	Devuelve el número x truncado a entero.
rand()	Devuelve un número aleatorio entre 0 y 1.
sin(x)	Devuelve el seno de x.
sqrt(x)	Devuelve la raíz cuadrada de x.
srand(expr) srand()	Establece una semilla para el generador de números aleatorios que usa rand(). Si no se especifica expr, utiliza el reloj del sistema.

Funciones definidas por el usuario

El lenguaje nos permite crear funciones propias para construir un código más prolijo. La sintaxis para crear una función es la siguiente:

```
function nombre(argumentos) { sentencias }
```

El cuerpo de la función puede tener una sentencia return o no. Las llamadas a una función pueden ser anidadas o recursivas. Los argumentos a la función se pasan por valor (copias) si son variables y por referencia si son vectores. Las funciones pueden ser llamadas antes de ser definidas en el código.

Fuentes

Página del manual de gawk.

Página del manual de mawk.

http://www.gnu.org/software/gawk/manual/ - Documentación oficial de la versión de AWK que

viene con Linux. Está en ingles, pero es altamente recomendable.

Mini-HowTo AWK
Guillermo Greco
http://www.yafuiste.com.ar/files/apuntes/scripts/awk_comp2.zip

http://www.lawebdelprogramador.com/cursos/mostrar.php?id=174&texto=AWK - En esta página encontrarán 3 cursos o manuales sobre AWK en castellano. No son tan confiables como el anterior, pero seguro los pueden utilizar como referencia.

http://www.gnu.org/software/bash/manual/bash.html - Documentación oficial de Bash. No explica tanto como programar en bash, lo que nos da es un conocimiento más profundo del shell en sí. También está en inglés.

http://www.tldp.org/guides.html - En esta página tienen que buscar el título "Advanced Bash-Scripting Guide". Esta guía se puede usar como referencia y es muy buena. La contra, nuevamente es que está en inglés.

http://www.lawebdelprogramador.com/cursos/mostrar.php?id=186&texto=Linux/Unix+Shell+Scripting - En esta página hay manuales sobre como realizar scripts en shell. La mayoría están en castellano.

Wikipedia

Comando make y makefiles

Objetivo

Que el alumno pueda utilizar el comando make eficientemente. Para esto debe saber crear los makefiles necesarios. Este comando será de utilidad en el desarrollo de los trabajos prácticos en los que sea necesario programar en lenguajes que requieran compilación del código, como es el C o el C++.

Requerimientos previos

Es muy recomendable, aunque no estrictamente necesario, que el alumno maneje los conceptos básicos de uso de un entorno bash, el intérprete de comandos de un sistema GNU/Linux.

Desarrollo

Introducción

El comando make está pensado para manejar proyectos de desarrollo grandes o pequeños. La potencia de éste se ve mas en los proyectos grandes, ya que el make sabe manejar las dependencias entre los códigos fuente y los ejecutables o archivos objeto, de forma tal que solo procesa las porciones de código que se han modificado desde la última vez que se compiló el proyecto.

En general se ha utilizado este comando para manejar proyectos en C o C++. Aunque en realidad se podría manejar un proyecto de cualquier lenguaje de programación, siempre que los comandos para compilar el código se puedan ejecutar en el interprete del sistema operativo. Es más, se podría utilizar el make para cualquier tarea que consista en crear archivos a partir de otros, cuando estos últimos fueron modificados. Un caso medio rebuscado podría ser el de realizar una copia de respaldo de determinados archivos cuando éstos se modifiquen.

Para que el make pueda hacer su trabajo, debemos proporcionarle a éste un makefile para que sepa cómo hacerlo. Un makefile es un archivo de texto donde se deben expresar las dependencias entre las distintas porciones del proyecto y que hacer cuando estas son modificadas.

Escribiendo Makefiles - Reglas

La estructura de un Makefile consiste básicamente en un conjunto de reglas que tienen la siguiente estructura:

```
objetivo...: requisitos...
        comando
        …
        …
```

El objetivo normalmente es un archivo. Los requisitos también son uno o varios archivos. Lo que le estamos diciendo al make con una regla de este tipo es que siempre que los archivos requisito se modifiquen (es decir que su fecha de última modificación sea mayor que la del archivo objetivo) se debe ejecutar el comando de la regla, que en teoría debe actualizar el archivo objetivo en base a los nuevos archivo requisito. Dado lo complicado de este último concepto, vamos a ver todo nuevamente con un ejemplo concreto:

```
meminfo: meminfo.c
        gcc -o meminfo meminfo.c
```

La idea de esta regla es muy simple: Siempre que se modifique meminfo.c, hay que volver a compilarlo para que se genere el nuevo ejecutable meminfo. Un detalle importante a tener en cuenta es que siempre un comando de una regla debe ser precedido por un carácter de tabulación.

Ahora vamos a desarrollar un ejemplo un poquito más complejo pero también más útil. Imaginemos un programa que dependiendo de una orden ingresada desde un teclado se conecte a un servidor y nos traiga el resultado. Ahora, supongamos que somos unos programadores prolijos y decidimos dividir el código en un archivo con todas las funciones que implementan la comunicación por sockets, un archivo que implementa el protocolo del cliente y finalmente un archivo principal que maneje la consola y utilice las funciones cliente. También es lógico pensar en un archivo de definiciones (header) independiente para cada módulo del programa.

Dado el ejemplo anterior, se supone que tengo seis archivos fuente: principal.c, principal.h, cliente.c, cliente.h, socket.c, socket.h. También en este ejemplo, el módulo cliente depende del módulo socket y el módulo principal a su vez depende del módulo cliente. Así como también cada archivo módulo depende tanto de su archivo de código como de su archivo de definiciones. Por último, vamos a suponer que el nombre final del ejecutable debería ser ConsultaServidor. Dadas estas definiciones, el archivo makefile debería ser el siguiente:

```
ConsultaServidor: principal.o cliente.o socket.o
        gcc -o ConsultaServidor principal.o cliente.o socket.o

principal.o: principal.c principal.h cliente.h
        gcc -c principal.c

cliente.o: cliente.c cliente.h socket.h
        gcc -c cliente.c

socket.o: socket.c socket.h
        gcc -c socket.c
```

De esta forma, si el contenido de una de las funciones del módulo socket cambia, solo debo volver a generar socket.o y luego el ejecutable principal, sin volver a compilar el módulo cliente y el módulo principal. En cambio, si lo que cambia son las definiciones de las funciones del socket, es decir socket.h, debo volver a compilar socket.o y también cliente.o, ya que dentro del módulo cliente utilizo las funciones del módulo socket. La idea de exponer este ejemplo un poquito más complicado es que se imaginen la cantidad de tiempo que se puede ganar si estoy haciendo un proyecto con, digamos, 200 objetos, cada uno con su archivo separado y quiero recompilar luego de haber modificado uno de esos objetos. Les dejo a ustedes las matemáticas.

Un aspecto importante de cómo procesa make los makefiles, es que siempre toma la primer regla que encuentre en el archivo como la regla por defecto. Por lo tanto en nuestro último ejemplo, siempre intentará construir el archivo ConsultaServidor cuando llamemos a make sin ningún parámetro:

```
$ make
```

Si, por ejemplo, acabo de modificar socket.c y lo único que quiero por el momento es testear si compila ese módulo independientemente de todos los demás, tendría que especificarle el objetivo que quiero lograr al make como un parámetro por linea de comandos de la siguiente forma:

```
$ make socket.o
```

Objetivos que no son archivos – Phony targets

Las reglas no necesariamente tienen que tener un requisito, y tampoco su objetivo tiene que ser obligatoriamente un archivo. Hay reglas que se escriben para automatizar operaciones que se realizan seguido durante el desarrollo de un sistema. El ejemplo más utilizado en la práctica es cuando queremos recompilar todo el código sin importar que módulos han sido modificados y cuales no. Para esto creamos la regla que les muestro a continuación.

```
clean:
	rm -f *.o
```

De esta forma, cuando escriba en el intérprete de comandos la orden que les muestro a continuación, se borrarán todos los archivos objeto del proyecto y por lo tanto la próxima vez que ejecute make, se compilará obligatoriamente todo el código.

```
$ make clean
```

El problema que se puede llegar a presentar es que a alguien se le ocurra crear un archivo clean en el directorio. Si esto pasara, como la regla que les transcribí antes no tiene requisito, nunca se daría la condición de que el requisito tenga fecha de modificación posterior al objetivo y por lo tanto la regla dejaría simplemente de funcionar. Para evitar esto lo que tenemos que hacer es declarar explícitamente en el makefile que el objetivo es falso (en ingles es phony target) de la siguiente forma:

```
.PHONY: clean

clean:
	rm -f *.o
```

Uso de Variables

En un makefile se pueden declarar variables para simplificar la lectura de éste y también escribir menos. Consideremos el siguiente makefile, que es igual al último que trabajamos, con el agregado de que queremos compilar con símbolos de debuging, pero no siempre sino solo cuando queremos debuguear.

```
.PHONY: clean

ConsultaServidor: principal.o cliente.o socket.o
	gcc -g -o ConsultaServidor principal.o cliente.o socket.o

principal.o: principal.c principal.h cliente.h
	gcc -g -c principal.c

cliente.o: cliente.c cliente.h socket.h
	gcc -g -c cliente.c

socket.o: socket.c socket.h
	gcc -g -c socket.c

clean:
	rm -f *.o
```

Los problemas que se ven rápidamente en este makefile son dos. Uno es que se repiten muchas veces en distintas lineas la misma información. El otro problema es que cuando quiero compilar sin los símbolos de dubug (quizá porque quiero ya mandarlo a test o a producción y sin símbolos el ejecutable ocupa mucho menos espacio) debo modificar todas las líneas de comandos para sacar la opción '-g' de éstas. La forma que nos da el make de solucionar estos problemas son las variables, y abajo les transcribo cómo se haría con el makefile que estamos estudiando.

```
.PHONY: clean

objetos = principal.o cliente.o socket.o
cflags = -g

ConsultaServidor: $(objetos)
	gcc $(cflags) -o ConsultaServidor $(objetos)

principal.o: principal.c principal.h cliente.h
	gcc $(cflags) -c principal.c

cliente.o: cliente.c cliente.h socket.h
	gcc $(cflags) -c cliente.c

socket.o: socket.c socket.h
	gcc $(cflags) -c socket.c

clean:
	rm -f $(objetos)
```

Además de solucionar los problemas antedichos, podemos ver como la regla clean ahora es más segura y solamente borrará los archivos objeto de nuestro proyecto sin margen de error.

El programa make es muchísimo más potente de lo que se vimos en esta clase, pero con estos simples conceptos, podrán encarar proyectos bastante complejos. El que quiera profundizar más tendrá que dirigirse al manual de usuario, que lamentablemente se encuentra solo disponible en inglés.

Fuentes

GNU Make Manual
Free Software Fundation
http://www.gnu.org/software/make/manual

Fork, wait y exec

Objetivo

Lograr que el alumno comprenda cómo se implementa en un sistema operativo real GNU/Linux (Válido también para todos los sistemas operativos desarrollados bajo la norma POSIX) la paralelización de tareas utilizando varios procesos, respetando la precedencia de las tareas.

Requerimientos previos

Saber programar en C. Saber compilar un programa en GNU/Linux (Véase el capítulo "Comando make y makefiles" en esta misma guía). Tener un conocimiento básico del entorno bash.

Desarrollo

Primer problema, procesar dos tareas en paralelo

El problema más sencillo que nos podemos plantear es cómo ejecutar concurrentemente (o casi) dos procesos independientes. Es decir, cómo realizar un programa que responda al siguiente grafo de precedencia:

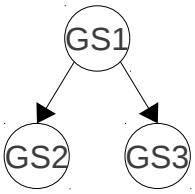

Fig. 1

Donde GS1 significa "Grupo de sentencias 1".

Para lograr nuestro objetivo vamos a utilizar la función fork(), que al ser llamada, crea un nuevo proceso que es una copia exacta, salvo por un par de excepciones, del proceso que la lanza y lo posiciona en el sistema operativo cómo listo para ser ejecutado y le devuelve el control al proceso inicial para que continue con su ejecución. Las diferencias entre el proceso nuevo y el viejo son tres:

1. Tienen distinto PID (Identificador de proceso).
2. El proceso viejo recibe como salida del fork() el PID del nuevo y el nuevo recibe 0.
3. El espacio de direccionamiento de memoria del proceso nuevo no se copia en un principio, sino que apunta al del proceso viejo. Cuando el proceso nuevo cambia algún dato de su memoria, por ejemplo por cambiar el valor de una variable, el sistema operativo primero le crea una copia nueva de la página de memoria

involucrada y luego modifica esta última. Este mecanismo está pensado para que la llamada al fork() sea más rápida, ya que solo tiene que crear el nuevo PCB y copiar las tablas de páginas de memoria del padre (llamamos al proceso viejo padre y al nuevo hijo).

Entonces, el segundo punto citado nos indica cómo verificar, en tiempo de ejecución si estoy ejecutando en el proceso padre o en el hijo, luego de llamar a fork(). No nos olvidemos que son copias identicas. Ahora voy a transcribir un ejemplo y explicarlo para fijar los conceptos. Hay que tener en cuenta que no se realizan las comprobaciones de seguridad por motivos didácticos. Más adelante entraremos en ese tema, ya que hacen que el código sea más difícil de leer.

```
#include<stdio.h>
#include<sys/types.h> // Se define el tipo de dato pid_t
#include<unistd.h>    // Contiene las definiciones de fork() y sleep().

int main() {
  pid_t hijo=0;                                                  // GS1
                                                                 // GS1
  printf("Soy el padre y tardaré 5s en empezar.\n");             // GS1
  sleep(5);                                                      // GS1

  if((hijo=fork())!=0) {
    printf("Soy el padre, cree a (%d) dormiré 4s.\n", hijo); // GS2
    sleep(4);                                                    // GS2
    printf("Soy el padre, dormiré 1s y terminaré.\n");           // GS2
    sleep(1);                                                    // GS2
  } else {
      printf("Soy el hijo y dormiré 4s.\n");                     // GS3
      sleep(4);                                                  // GS3
      printf("Soy el hijo, esperaré 1s y terminaré.\n");         // GS3
      sleep(1);                                                  // GS3
  }

  return 0; // Este código se ejecuta 2 veces, para GS2 y GS3
}
```

Cómo se ve en el ejemplo, el if es el que diferencia si estoy ejecutando el proceso padre o el proceso hijo. Si estoy ejecutando dentro del primero, el fork devolverá el pid del hijo y entrará a ejecutar la sección por verdadero (GS2). Si estoy ejecutando dentro del hijo, el fork devolverá 0, con lo cual ejecutará la sección por falso (GS3).

Ahora analicemos la salida que generó este script en mi máquina. Aclaro que es la salida en mi máquina ya que dependiendo donde y cuando se ejecute la salida podría tener ciertas variaciones. Ahí va:

```
Soy el padre y tardaré 5s en empezar.
Soy el hijo y dormiré 4s.
Soy el padre, cree a (20138) dormiré 4s.
Soy el hijo, esperaré 1s y terminaré.
Soy el padre, dormiré 1s y terminaré.
```

Tal como era de esperarse, las lineas pertenecientes a GS2 y GS3 se mezclan tal como lo indicaba el gráfico del que partimos. Ahora les voy a mostrar la salida del comando ps luego de que se ejecutara el fork y antes de que terminaran los dos procesos. Se puede ver claramente cómo el bash es el padre del proceso viejo y éste a su vez es el padre del proceso nuevo.

```
jmfera   19638 19637  0 20:58 pts/7    00:00:00 -bash
jmfera   20135 19638  0 23:18 pts/7    00:00:00 ./forkywait
jmfera   20138 20135  0 23:18 pts/7    00:00:00 ./forkywait
```

Por último, en caso de que fork() no sea capaz de crear el nuevo proceso, devolverá un -1 y escribirá errno (ejecute “man errno” para saber de qué estoy hablando y “man fork” para conocer los códigos de error que genera fork()) con información acerca del motivo del fallo.

Haciendo respetar la precedencia

Muchas veces nos encontramos con el problema de que una tarea no puede ser realizada hasta que otras no esten terminadas. Así como en la construcción no se puede hacer el techo antes que las columnas, en un programa podría no ser posible ejecutar una sentencia hasta que otras no hayan sido completadas, y como lo que buscamos es paralelizar la ejecución, esto empieza a ser un problema. Un ejemplo sencillo de lo que hablo se puede graficar de la siguiente forma:

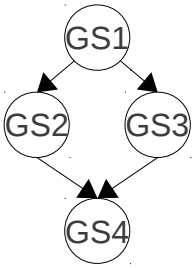

Fig. 2

Para poder lograr este objetivo debemos utilizar la función wait(), que precisamente sirve para esperar que un hijo termine.

```
#include<stdio.h>
#include<sys/types.h> // Se define el tipo de dato pid_t
#include<unistd.h>    // Contiene las definiciones de fork() y sleep().
#include<sys/wait.h>  // Contiene la definición de wait().

int main() {
  pid_t hijo=0, hijoWaited=0;                                  // GS1
                                                               // GS1
  printf("Soy el padre y tardaré 5s en empezar.\n");           // GS1
  sleep(5);                                                    // GS1

  if((hijo=fork())!=0) {
    printf("Soy el padre, cree a (%d) dormiré 4s.\n", hijo); // GS2
    sleep(4);                                                  // GS2
    printf("Soy el padre, esperaré a (%d).\n", hijo);          // GS2
    hijoWaited=wait(NULL);
    printf("Ya terminó (%d).\n", hijoWaited);                  // GS4
  } else {
      printf("Soy el hijo y dormiré 6s.\n");                   // GS3
      sleep(6);                                                // GS3
      printf("Soy el hijo, esperaré 1s y terminaré.\n");       // GS3
      sleep(1);                                                // GS3
  }

  return 0; // Este código se ejecuta 2 veces, para GS3 y GS4
}
```

Si leemos con atención el código anterior, nos daremos cuenta de que el proceso padre llegará al wait 3 segundos antes de que termine el hijo, sin embargo la salida de este programa es la siguiente:

```
Soy el padre y tardaré 5s en empezar.
Soy el hijo y dormiré 6s.
Soy el padre, cree a (23884) dormiré 4s.
Soy el padre, esperaré a (23884).
Soy el hijo, esperaré 1s y terminaré.
Ya terminó (23884).
```

Tal como se puede ver, la precedencia entre los grupos de sentencias se respetó. El siguente grafo de precedencia complica un poco las cosas, con el objetivo de necesitar una nueva función llamada waitpid().

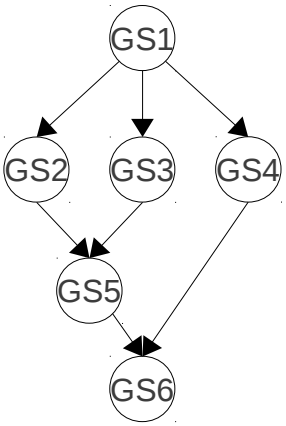

Fig. 3

Para lograr nuestro objetivo, vamos a hacer que el hilo de ejecución del padre sea elque ejecute GS1, GS2, GS5 y GS6. Por lo tanto, el proceso inicial deberá lanzar dos hijos, uno para ejecutar GS3 y uno para ejecutar GS4. Bien, si antes de ejecutar GS5 el proceso padre hiciera una llamada a wait(), correríamos el riesgo de que el proceso encargado de ejecutar GS4 terminara antes que el que tiene que ejecutar GS3. Por lo tanto siguiendo esa lógica se transformaría en un grafo distinto, que presento aquí para intentar (cosa difícil) dar claridad a este párrafo.

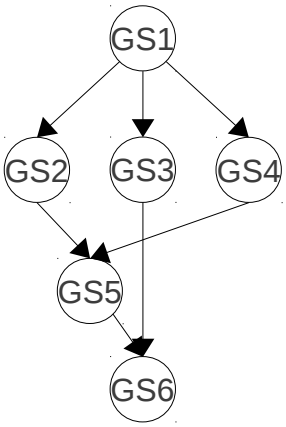

Fig. 4

Por eso utilizaremos waitpid(), que nos permite especificarle al sistema operativo a qué proceso puntual estamos esperando. Les transcribo abajo el código de ejemplo.

```
#include<stdio.h>
#include<sys/types.h> // Se define el tipo de dato pid_t
#include<unistd.h>    // Contiene las definiciones de fork() y sleep().
#include<sys/wait.h>  // Contiene la definición de wait().

int main() {
  pid_t hijoGS3=0, hijoGS4=0, hijoWaited=0;                              // GS1
                                                                         // GS1
  printf("Soy el padre y tardaré 5s en empezar.\n");                     // GS1
  sleep(5);                                                              // GS1

  if((hijoGS3=fork())!=0) {
```

```
    if ((hijoGS4=fork())!=0) {
      printf("P: Cree a (%d) y a (%d), dormiré 4s.\n", hijoGS3, hijoGS4); // GS2
      sleep(4);                                                          // GS2
      printf("P: Esperaré a (%d).\n", hijoGS3);                          // GS2
      hijoWaited=waitpid(hijoGS3, NULL, 0);
      printf("P: Ya terminó (%d).\n", hijoWaited);                       // GS5
      printf("P: Esperaré a (%d).\n", hijoGS4);                          // GS5
      hijoWaited=waitpid(hijoGS4, NULL, 0);
      printf("P: Ya terminó (%d).\n", hijoWaited);                       // GS6
    } else {
        printf("%d: Dormiré 4s.\n", getpid());                           // GS4
        sleep(4);                                                        // GS4
    }
  } else {
      printf("%d: Dormiré 6s.\n", getpid());                             // GS3
      sleep(6);                                                          // GS3
  }

  return 0; // Este código se ejecuta 3 veces, para GS3, GS4 y GS6
}
```

A pesar de que el proceso hijoGS4 termina antes que el proceso hijoGS3, la precedencia se respeta, ya que la salida del programa es:

```
Soy el padre y tardaré 5s en empezar.
24253: Dormiré 6s.
24254: Dormiré 4s.
P: Cree a (24253) y a (24254), dormiré 4s.
P: Esperaré a (24253).
P: Ya terminó (24253).
P: Esperaré a (24254).
P: Ya terminó (24254).
```

La función vfork()

Ésta llamada al sistema operativo crea un nuevo proceso al igual que `fork()` (los posibles valores a devolver también son los mismos) pero con algunas diferencias importantes.

Al llamar a `vfork()` la ejecución del padre es suspendida hasta que el proceso hijo llame a una función `exec()` o bien finalice su ejecución, ya sea normalmente (a través de una llamada a `_exit()`) o anormalmente.

El proceso hijo puede hacer muy pocas cosas para que el resultado de su ejecución y la de su padre no sea indefinido. Éste solo podría modificar la variable en la que se guardó su pid, ningúna otra variable. No podría llamar a otra función que no sea `_exit()` o `execve()`. No podría tampoco salir de la función desde donde se llamó a `vfork()`.

Estas diferencias tienen que ver con que el proceso hijo comparte toda la memoria del proceso padre, incluida la pila.

La utilización de la llamada `vfork()` no es recomendable a menos que la aplicación que estamos programando necesite que la creación del nuevo proceso sea realmente rápida y que inmediatamente sea necesario cambiar el código de nuestro proceso hijo con el de otro programa a través de `execve()`.

Vale la aclaración de que las últimas versiones de `fork()` son apenas mas lentas que `vfork()`, ya que `fork()` solo copia la pila del proceso padre para el proceso hijo en el momento de la creación y su espacio de memoria apunta al del padre. Si luego el proceso hijo modifica alguna porción de memoria (por ejemplo una variable cualquiera), el sistema operativo copia la página correspondiente del padre al espacio de direccionamiento del hijo y recién ahí la modifica.

Solo para ejemplificar, les copio más abajo un ejemplo de uso de `vfork()` sacado de la

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

int main (int argc, char *argv[]) {
  pid_t childpid;

  printf("Voy a ejecutar el comando %s en un nuevo proceso.\n", argv[1]);
  if ((childpid = vfork()) == -1) {
    perror("Error al crear el hijo");
    exit(1);
  }
  else if (childpid ==0) {
    if (execvp(argv[1], &argv[1]) <0) // Note que execvp() y toda la familia de
    _exit(1);                         // funciones exec() son solo macros que
  }                                   // llaman a execve().
  else {
    printf("Soy el padre y termino\n");
    exit(0);
  }
}
```

La llamada execve() y la familia de funciones exec()

La llamada al sistema `execve()` se utiliza para ejecutar un archivo binario o un script cuya primera línea comience con `#!intérprete [argumentos]`. El prototipo de la llamada es el siguiente:

```
int execve(const char *nombre_archivo, char *const args[], char *const entorno[]);
```

Esta llamada, en caso de ejecutarse exitosamente, no retorna y el nuevo proceso es cargado en los segmentos de código, datos, bss y pila del proceso que llamó a la función. En caso de error, devuelve -1 y la variable de entorno `errno` es cargada apropiadamente.

Abajo les copio un ejemplo sencillo directamente copiado y pegado de la página del manual de execve() para entender bien cómo se utilizan los parámetros.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>

int main(int argc, char *argv[]) {
  char *newargv[] = { NULL, “hello”, “world”, NULL };
  char *newenviron[] = { “HOME=/root”, NULL };  /* La definición de la variable
                                                   (entorno) no estaba en el código
                                                   original, pero la agregue como ejemplo */

  assert(argc == 2);  /* argv[1] identifies
                         program to exec */
  newargv[0] = argv[1];

  execve(argv[1], newargv, newenviron);
  perror(“execve”);   /* execve() only returns on error */
  exit(EXIT_FAILURE);
}
```

El primer parámetro que recibe la función debe ser la ruta completa al archivo a ejecutar, que en nuestro ejemplo debe ser pasado como parámetro al programa.

Puede verse claramente en el ejemplo que la función recibe como segundo parámetro la lista de argumentos (siendo el primero la ruta completa al archivo a ejecutar, que aunque no es obligatorio, es altamente recomendable) como un vector de cadenas de caracteres terminado en un elemento nulo. El tercer parámetro es el entorno de ejecución, que son las

definiciones de las variables de entorno que tendrá disponibles nuestro nuevo proceso. Cómo se ve en el ejemplo serían cadenas del tipo "NOMBRE=valor", una por cada variable de entorno que quiero que sea visible en el nuevo proceso.

Como resultado, si este programa se llamara ejemplo_exec y el programa a ejecutar por execve() se llamara nuevo_programa, cuando yo ejecute:

```
$ ejemplo_exec ./nuevo_programa
```

Nuestro código ejecutaría nuevo_programa como si lo hubiera llamado desde el intérprete de comandos de la siguiente forma:

```
$ ./nuevo_programa hello world
```

Con la diferencia de no tener accesibles dentro de nuevo_programa todas las variables de entorno de la sesión sino solamente a la variable HOME.

En la práctica el uso de esta llamada puede volverse un poco incómodo, debido a ésto, la biblioteca nos proporciona una serie de funciones que nos permite llamar a execve() sin necesidad de hacer varias tediosas transformaciones y copiado de datos. Los prototipos de estas funciones son:

```
int execl(const char *ruta, const char *arg0, const char *arg1, ...);
int execlp(const char *archivo, const char *arg0, const char *arg1, ...);
int execle(const char *ruta, const char *arg0, ..., char * const envp[]);
int execv(const char *ruta, char *const argv[]);
int execvp(const char *archivo, char *const argv[]);
```

En el caso de execl(), el primer argumento deberá ser la ruta completa al ejecutable que queremos pasar a execve(), y los siguientes serán los argumentos que se pasarán al nuevo proceso. El último argumento *debe* ser nulo y debe estar casteado: (const char *)NULL. El entorno de ejecución del nuevo programa será el mismo (sin necesidad de pasarlo como parámetro) que el del proceso que llama a la función.

La función execlp() se diferencia de execl() en la forma de tratar al primer argumento. Si éste no contiene una barra (/) entonces buscará el ejecutable como lo haría el interprete de comandos, recorriendo los directorios enumerados en la variable de entorno PATH.

La función execle() funciona igual a execl(), pero nos permite pasarle un nuevo entorno luego del argumento nulo en forma de vector de cadenas de caracteres, con el mismo formato que en execve().

Las funciones execv() y execvp() reciben el primer argumento (el ejecutable a cargar) como execl() y execlp() respectivamente. A diferencia de éstas, reciben sus argumentos en forma de vector, al igual que execve(). La ventaja de utilizar execv() en lugar de execve() es que no necesito pasar como parámetro el entorno cuando necesito que sea el mismo que el del programa que llama a la función. Pueden ver un ejemplo de uso de la función execvp() en la sección anterior, utilizada en conjunto con vfork().

Fuentes

Página del manual de fork(), wait(), vfork(), execve() y exec().
KernelAnalysis-HOWTO: Linux Multitasking.

Señales

Objetivo

Brindar los conocimientos necesarios para comenzar a comunicar procesos utilizando el sencillo mecanismo de las señales POSIX.

Requerimientos previos

El alumno deberá tener conocimientos del lenguaje C. Es recomendable manejar la funcionalidad de multiprogramación que nos brindan las funciones `fork()`, `wait()` y `waitpid()` del estándar POSIX que fueron desarrolladas en un capítulo anterior de esta guía.

Desarrollo

Generalidades

Las señales son un mecanismo de comunicación entre procesos. Su potencial es bastante limitado, pero no por eso menos útil.

La idea es la siguiente:

- Que un proceso pueda avisar a otro de la ocurrencia de un evento.
- Que el proceso que recibe la señal tenga asociada a ésta una acción.
- Que el proceso, en ese instante, pare la ejecución en la instrucción en que se encuentra y comience a ejecutar la rutina que tiene asignada para manejar la señal. Cuando dicha rutina termina, el proceso continua la ejecución desde donde fue interrumpido.

Las señales se identifican por un número y nombre, y todos los procesos tienen una rutina de atención asignada por defecto a cada señal.

Las rutinas asignadas por defecto son solo cuatro:

- Term: Finaliza la ejecución.
- Core: Escribe un archivo Core, con la información de entorno de ejecución del proceso en ese momento y finalmente termina la ejecución.
- Ign: La señal se ignora y se sigue ejecutando normalmente.
- Stop: El proceso se detiene en ese punto de ejecución hasta recibir otra señal.

Algunas señales importantes, con sus respectivas acciones por defecto, según la página del manual (man 7 signal) son las siguientes:

Nombre	Número	Acción
SIGHUP	1	Term
SIGINT	2	Term
SIGQUIT	3	Core
SIGSEGV	11	Core
SIGALRM	14	Term
SIGTERM	15	Term
SIGUSR1	30,10,16	Term
SIGUSR2	31,12,17	Term
SIGCHLD	20,17,18	Ign
SIGCONT	19,18,25	
SIGSTOP	17,19,23	Stop

Dándole uso particular a las señales

Esta acción por defecto puede ser cambiada por una rutina programada por nosotros. Esto puede ser útil en muchos casos.

La forma más fácil de realizar esto es a través de la función `signal()`. Con esta función, podemos hacer que el proceso ejecute una porción de código cuando reciba una señal en particular.

```
#include<sys/types.h>
#include<unistd.h>
#include<signal.h>
#include<stdio.h>
/* Esta es la función que se ejecutará cuando se reciba la señal. Recibirá
 * por parámetro el número de señal.
 */
void capturar_senal(int iNumSen) {
  printf("Se recibió la señal %d\n", iNumSen);
}
int main() {
  signal(SIGINT, capturar_senal); // Cuando se reciba SIGINT, ejecutar
                                  // capturar_senal.
  signal(SIGALRM, capturar_senal); // Ídem para SIGALRM.
  signal(SIGTERM, capturar_senal); // Ídem para SIGTERM.
  for(;;) // Un loop infinito.
    pause(); // Se ejecuta pause dentro del loop para que
             // no sea espera activa.
  return 0;
}
```

Nuestro programa, cuando esté ejecutando, será capaz de capturar las señales SIGINT, SIGALRM y SIGTERM y mostrarnos por pantalla un mensaje advirtiéndonos de ese suceso.

Ahora bien, para probar nuestro programa, tenemos que enviarle una señal al proceso. Una señal se puede enviar desde otro proceso o bien desde la línea de comandos del sistema operativo.

Para enviarle una señal a un proceso desde el sistema operativo, debo utilizar el comando `kill` con la siguiente sintaxis:

```
$ kill -INT 3809
```

En este caso particular, le estamos enviando la señal SIGINT al proceso cuyo identificador es

3809.

Otra opción es enviar la señal desde un programa, con la función homónima al comando.

```
#include<sys/types.h>
#include<unistd.h>
#include<signal.h>
#include<stdlib.h>
int main(int argc, char *argv[]) {
  kill(atoi(argv[1], atoi[argv[2]);
  return 0;
}
```

Al llamar a este programa, le debemos pasar como primer parámetro el identificador del proceso y como segundo parámetro el número de la señal a enviar.

Un dato muy importante a tener en cuanta cuando trabajamos con señales es que SIGSTOP y SIGKILL no se pueden capturar.

La señal SIGCHLD

Es muy típico el uso de señales en conjunción con `fork()`. Como sabíamos, un proceso que utiliza `fork()` para crear un proceso hijo, debe esperar a que termine con la sentencia `wait()`. En caso de que un proceso hijo termine y el padre no ejecute `wait()`, el proceso hijo quedará en estado *zombie*.

El problema que se nos plantea en estos casos, es que si queremos generar hijos y seguir ejecutando otro código que no sea `wait()`, como podría ser el caso de un servidor, no lo podríamos hacer. Les paso un ejemplo para que lo vean:

```
#include<sys/types.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
int main(int argc, char *argv[]) {
  pid_t iPidHijo = 0;
  iPidHijo = fork();
  if (iPidHijo == 0) { // Pregunto si soy el hijo
    printf("PID del hijo: %ld\n", (long) getpid());
    exit(0);
  }
  else {
    for(;;)     // En caso de ser el padre ejecuto un código
      pause();  // cualquiera.  Éste es solo un ejemplo.
  }
  return(EXIT_SUCCESS);
}
```

Mientras el proceso padre ejecuta `pause()` vayan a otra pantalla y consulten los procesos con el comando `ps`. Podrán verificar que el proceso hijo queda *zombie*.

Para este caso el uso de señales es de mucha utilidad, ya que cuando un proceso termina, le envía a su padre la señal SIGCHLD. De esta forma, el padre puede crear hijos y seguir ejecutando cualquier código, y en caso de que el hijo termine, ejecutar el `wait()` dentro de una función asociada a la señal a través de `signal()`.

Acá va un ejemplo de como capturar SIGCHLD para que no queden procesos *zombies* a medida que mueren los hijos:

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

```
#include<unistd.h>
#include<stdio.h>
void esperar_hijo(int iNumSen) {
  wait(NULL);
}
int main(int argc, char *argv[]) {
  pid_t iPidHijo = 0;
  signal(SIGCHLD, esperar_hijo);
  iPidHijo = fork();
  if (iPidHijo == 0) { // Pregunto si soy el hijo
    printf("PID del hijo: %ld\n", (long) getpid());
    exit(0);
  }
  else {
    for(;;)
      pause();
  }
  return(EXIT_SUCCESS);
}
```

Ahora vuelvan a verificar los procesos con el comando `ps` y observen la diferencia.

Ahora nuestro programa es invulnerable a la generación de procesos *zombies*. Esto es verdad solamente debido a que solo generamos un proceso hijo. En el caso de un servidor, que puede generar cientos de hijos, se podría dar el caso de que un hijo termine mientras nosotros estamos ejecutando la rutina "esperar_hijo". En ese caso, este último proceso hijo quedará *zombie*. Para evitar eso, debemos cambiar la función "esperara_hijo" por la siguiente:

```
void esperar_hijo(int iNumSen) {
  while (waitpid (-1, NULL, WNOHANG) > 0);
}
```

Un paso más: Uso de sigaction()

Hasta ahora sabemos como se recibe una señal. Si queremos tener mejor control de lo que pasa con nuestro sistema y con nuestros procesos en una aplicación multiprogramada, tenemos que saber algo más acerca del entorno que generó la señal. Un ejemplo de esto puede ser querer saber cual fue el proceso que generó la señal. Otro ejemplo: Cuando un hijo manda una señal SIGCHLD, puede ser porque terminó, porque alguien lo mató, porque terminó anormalmente, porque se ha parado (STOP) o porque estaba parado y continuará.

Nosotros vamos a desarrollar ejemplos con estos dos casos, pero leyendo la página de la sección 2 del manual de `sigaction()` se puede explotar todo el potencial de esta función.

En primer término vamos a ver el caso de un programa que sabe cual es el identificador del proceso que le envió la señal.

En el caso de la función `signal()`, ésta recibía como parámetros la señal y el nombre de la rutina de tratamiento. A la función `sigaction()`, podemos darle más información para que pueda hacer su trabajo. Por eso recibe como parámetro la señal a capturar y una estructura de datos:

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer)(void);
}
```

Tanto `sa_handler` como `sa_sigaction` se utilizan para especificar la rutina que va a manejar la señal. No se pueden utilizar ambas a la vez. Si utilizo `sa_handler`, el resultado

será igual que si utilizara la función signal(), es decir que la rutina recibirá como único parámetro el número de señal que fue capturada. Si utilizo `sa_sigaction` la rutina recibirá el número de señal, una estructura `siginfo_t` con los famosos datos del entorno que generó la señal y un puntero a `void` que no se utiliza.

El campo `sa_mask` se utiliza para especificar qué señales se deben bloquear mientras se ejecuta la rutina. En nuestros ejemplos utilizaremos la función `sigfillset()` para cargar este campo y así bloquear todas las señales.

El campo `sa_flags` se utiliza para modificar el comportamiento de la rutina. Por ejemplo, para que a la rutina le llegue la estructura `siginfo_t`, `sa_flags` debe tener configurada la bandera `SA_SIGINFO`.

La estructura `siginfo_t` contiene los siguientes campos:

```
siginfo_t {
  int si_signo; /* Número de señal */
  int si_errno; /* Un valor errno */
  int si_code; /* Código de señal */
  pid_t si_pid; /* ID del proceso emisor */
  uid_t si_uid; /* ID del usuario real del proceso emisor */
  int si_status; /* Valor de salida o señal */
  clock_t si_utime; /* Tiempo de usuario consumido */
  clock_t si_stime; /* Tiempo de sistema consumido */
  sigval_t si_value; /* Valor de señal */
  int si_int; /* señal POSIX.1b */
  void * si_ptr; /* señal POSIX.1b */
  void * si_addr; /* Dirección de memoria que ha producido el fallo */
  int si_band; /* Evento de conjunto */
  int si_fd; /* Descriptor de fichero */
}
```

Abajo les transcribo un programa que sabe identificar cual es el identificador del proceso que le mandó la señal. En este programa solamente utilizamos los campos `si_pid` y `si_signo`.

```
#include<sys/types.h>
#include<unistd.h>
#include<signal.h>
#include<stdio.h>
void capturar_senal(int iNumSen, siginfo_t *info, void *ni) {
  printf("Se recibió la señal: %d\n", iNumSen);
  printf("También se puede leer por acá: %d\n", info->si_signo);
  printf("El proceso que envió la señal fue: %d\n", info->si_pid);
}
int main () {
  struct sigaction act; // La estructura que definirá como manejar la
                        //señal.
  act.sa_sigaction=capturar_senal; // definimos la rutina.
  sigfillset(&act.sa_mask); // Bloqueamos todas las señales mientras
                            // se ejecuta la rutina.
  act.sa_flags=SA_SIGINFO; // Es para que la estructura llegue
                           // instanciada a la rutina.
  sigaction(SIGINT, &act, NULL); // Establecemos la captura de la
                                 //señal.
  for(;;)
    pause();
  return 0;
}
```

Para hablar del otro ejemplo, es decir el de saber en que circunstancias estoy recibiendo un SIGCHLD, debo introducir primero el concepto con el que fue programado el campo `si_code` en la estructura `siginfo_t`. Este campo puede tomar distintos juegos de valores, dependiendo del tipo de señal que está recibiendo. Abajo solamente voy a transcribir los valores posibles para todas las señales y los específicos que puede tomar cuando se recibe `SIGCHLD`. Estas tablas se encuentran completas en la página del manual de `sigaction()`.

si_code	
Valor	Origen de la señal
SI_USER	Kill, sigsend o raise
SI_KERNEL	El núcleo
SI_QUEUE	sigqueue
SI_TIMER	El cronómetro ha vencido
SI_MESGQ	Ha cambiado el estado de mesq
SI_ASYNCIO	Ha terminado una E/S asíncrona
SI_SIGIO	SIGIO encolada

SIGCHLD	
CLD_EXITED	Ha terminado un hijo
CLD_KILLED	Se ha matado un hijo
CLD_DUMPED	Un hijo ha terminado anormalmente
CLD_TRAPPED	Un hijo con seguimiento paso a paso ha sido detenido
CLD_STOPPED	Ha parado un hijo
CLD_CONTINUED	Un hijo parado ha continuado

Vamos al ejemplo:

```
#include<sys/types.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
void capturar_senal(int iNumSen, siginfo_t *info, void *ni) {
  printf("Se recibi la señal: %d\n", iNumSen);
  printf("También se puede leer por acá: %d\n", info->si_signo);
  printf("El proceso que envió la señal fue: %d\n", info->si_pid);
  switch (info->si_code) {
    case CLD_EXITED:
      printf("El proceso hijo terminó normalmente\n");
      break;
    case CLD_KILLED:
      printf("El proceso hijo fue terminado\n");
      break;
    case CLD_DUMPED:
      printf("El proceso hijo terminó anormalmente\n");
      break;
    case CLD_STOPPED:
      printf("El proceso hijo fue parado\n");
      break;
    case CLD_CONTINUED:
      printf("El proceso hijo estaba parado y ahora continua\n");
      break;
  }
}
int main(int argc, char *argv[]) {
  struct sigaction act; // La estructura que definir como manejar la
                        // señal.
  pid_t iPidHijo = 0;
  act.sa_sigaction=capturar_senal; // definimos la rutina.
  sigfillset(&act.sa_mask); // Bloqueamos todas las señales mientras
                            // se ejecuta la rutina.
  act.sa_flags=SA_SIGINFO; // Es para que la estructura llegue
                           // instanciada a la rutina.
  iPidHijo = fork();
  if (iPidHijo == 0) { // Pregunto si soy el hijo
    printf("PID del hijo: %ld\n", (long) getpid());
    for(;;)
```

```
      pause();
    exit(0);
  }
  else {
    sigaction(SIGCHLD, &act, NULL); // Establecemos la captura de
                                    // la señal.
    for(;;)
      pause();
  }
  return(EXIT_SUCCESS);
}
```

Ahora compilen este programa, ejecútenlo y prueben de mandarle las señales SIGSTOP, SIGCONT y SIGTERM con el comando `kill`.

Aplicación de señales a sistemas de tiempo real

Hasta ahora hemos trabajado con un modelo simple de la realidad, en el cual los procesos, cuando reciben una señal, detienen su ejecución, dan paso a la rutina de atención de esa señal, y cuando terminan de atender la señal continúan por el paso en el que se habían quedado.

Uno de los casos que puede venir a tirarnos abajo este mundo simple que teníamos hasta ahora, podría ser el de una señal que caiga mientras estoy atendiendo una señal anterior y no mientras estoy en mi proceso principal. Por eso, en el último ejemplo agregamos, sin dar mayores detalles, la linea:

```
sigfillset(&act.sa_mask); // Bloqueamos todas las señales mientras
                          // se ejecuta la rutina.
```

En este ejemplo, el campo `sa_mask` es de tipo `sigset_t`, que es un vector de bits de 64 posiciones, en el cual están representadas todas las señales. O sea que la posición 1 del vector es SIGHUP, que es la señal número 1, la 10 es SIGUSR1 y así. La función `sigfillset()` inicializa este vector con todos los bits en 1. De esta forma, `sigaction()` establecerá el manejador de señal con la estructura `act` configurado para que mientras se está atendiendo a la señal, se bloqueen las señales.
Cuando terminemos de atender la señal, esta señal que quedó bloqueada deberá podrá ser atendida.

Lamentablemente aún pueden surgir nuevos inconvenientes. En el caso de recibir dos señales del mismo tipo, mientras la atención de éstas está bloqueada, el sistema puede reaccionar de dos formas distintas, dependiendo del rango de señal del que estemos hablando.

En el caso de las primeras 32 señales, las que aparecen en la página del manual (`man 7 signal`) el sistema operativo solamente guarda la ocurrencia o no de una señal. Por lo tanto, en caso de que un tipo de señal esté bloqueada y durante ese lapso recibamos más de una ocurrencia de ésta, el sistema solamente sabrá que se recibió por lo menos una señal de este tipo. En resumidas cuentas nos perdimos la segunda señal. Si este comportamiento no es aceptable, como pasa en los sistemas de tiempo real, debemos utilizar el segundo conjunto de 32 señales para nuestros propósitos. En este caso, al recibir una señal que está bloqueada en este momento, el sistema operativo lo guarda en una cola que está asociada al proceso y ya no se pierden las señales.

En algunos Unix el limite entre señales comunes y de tiempo real puede variar, por eso los estándares promueven el uso de una constante `SIGRTMIN` para establecer el piso de las señales de tiempo real. Entonces, por compatibilidad, nuestros programas no deberían hacer referencia a la señal 44, sino a la señal (`SIGRTMIN+12`) por poner un ejemplo. También hay un `SIGRTMAX` para no pasarnos.

Bloqueando señales en forma explícita

También nos puede pasar, en el caso de un sistema de tiempo real, que estemos ejecutando una rutina que no puede ser interrumpida por ninguna razón, y que sea parte del proceso principal, o sea que no sea una rutina de atención de señales.

En este caso, deberemos bloquear explícitamente las señales. Para realizar esta tarea hay varios caminos, cada uno con distintos pros y contras, y no es el objetivo de este texto explicarlos todos, por lo tanto se va a tomar un camino y se explicará. De cualquier forma les advierto que estudiando las páginas del manual de las funciones involucradas hay más información al respecto.

Lo primero que se debe hacer es tener un vector de 64 bits, de tipo `sigset_t`que nos permita especificar que señales vamos a bloquear. Para inicializarlo vacío, o sea con todos los bits en 0, podemos utilizar la función `sigemptyset(sigset_t *conjunto)`. Luego, le podemos agregar al conjunto las señales que luego vamos a bloquear con la función `sigaddset(sigset_t *conjunto, int señal)`. Cuando hayamos agregado todas las señales a bloquear, las bloqueamos efectivamente con la función `sigprocmask(int cod_op, sigset_t *conjunto, int señal)`. Si lo que quiero hacer es bloquear todas las señales, el conjunto debe ser inicializado con la función `sigfillset(sigset_t *conjunto)`.

Abajo transcribo un ejemplo de un proceso que bloquea explícitamente 4 tipos de señal durante 20 segundos. Dos de ellas son de tiempo real y dos no. Luego atiende las señales que haya recibido durante 5 segundos y muere, teniendo la precaución de bloquear todas las señales mientras está atendiendo a otra señal.

```
#include<stdio.h>
#include<sys/types.h>
#include<signal.h>
void manejar_senal(int no_senal, siginfo_t *info, void *noseusa) {
  printf("Recibí la señal: %d\n", info->si_signo);
}
int main () {
  sigset_t conjsen; // El conjunto de señales.
  struct sigaction sa;
  printf("PID del proceso: %d\n", getpid());
  printf("Vamos a capturar las señales %d, %d, %d y %d.\n", SIGUSR1,
    SIGUSR2, SIGRTMIN+10, SIGRTMIN+11);
  sigemptyset(&conjsen); // Se vacía el conjunto.
  sigaddset(&conjsen, SIGUSR1);     // Se agregan las señales.
  sigaddset(&conjsen, SIGUSR2);     // Las primeras dos no se
  sigaddset(&conjsen, SIGRTMIN+10); // van a encolar, las otras
  sigaddset(&conjsen, SIGRTMIN+11); // si.
  sigprocmask(SIG_SETMASK, &conjsen, NULL); // Se bloquean las 4 señales.
  sigfillset(&conjsen); // Se llena el conjunto.
  sa.sa_mask=conjsen;             // Con esta linea le vamos a
  sa.sa_flags=SA_SIGINFO;         // indicar que se bloqueen todas
  sa.sa_sigaction=&manejar_senal; // las señales mientras se
                                  // ejecuta la rutina de atención.
  sigaction(SIGUSR1, &sa, NULL); // Se establecen las rutinas de
  sigaction(SIGUSR2, &sa, NULL); // atención.
  sigaction(SIGRTMIN+10, &sa, NULL);
  sigaction(SIGRTMIN+11, &sa, NULL);
  sleep(40); // Durante este tiempo podemos
             // probar mandarle al proceso
             // varias señales de cada tipo
             // y ver que pasa una vez que
             // estas son desbloqueadas.
  sigemptyset(&conjsen);
  sigprocmask(SIG_SETMASK, &conjsen, NULL); // Se desbloquean las señales.
  sleep(5);
  return 0;
}
```

Fuentes

Páginas del manual de signal(2), kill(1), kill(2), pause(2), sigaction(2) y signal(7).

Fifos

Objetivo

Que el alumno sea capaz de comunicar dos o más procesos no emparentados a través del archivos especiales fifo.

Requerimientos previos

Para seguir esta clase es necesario que el alumno tenga conocimientos del lenguaje C.

Desarrollo

Introducción

En esta clase vamos a ver uno de los métodos más sencillos de comunicar dos procesos no emparentados, es decir, que no son padres ni hijos ni hermanos entre sí.

Los fifos generalmente se utilizan para comunicar procesos que residen en una misma máquina, debido a que requieren que ambos procesos tengan acceso al fifo, que se implementa como un archivo en disco. Debido a esto es que tienen la restricción de que ambos procesos deben por lo menos compartir un mismo sistema de archivos.

Aquel que ya haya trabajado con archivos en C, utilizando las funciones de fnctl.h (open o creat) y unistd.h (read, write, etc.) ya tiene la mitad del camino recorrido, ya que comunicar dos procesos a través de un fifo consiste en crear un archivo de tipo fifo y luego escribir o leer de él según sea el caso. A continuación, a través de una tabla intentaré graficar como sería la interacción entre un proceso productor y otro consumidor.

PRODUCTOR	CONSUMIDOR
Crear el archivo fifo: mkfifo()	El archivo también lo puede crear el consumidor, o podría estar ya creado de antemano.
Abrir el archivo para escritura: open()	Abrir el archivo para lectura: open()
Escribir los datos en el fifo: write()	
	Leer los datos del fifo: read()
Cerrar el archivo: close()	Cerrar el archivo: close()

Creando un fifo

Para tener un nuevo fifo simplemente hay que crear un archivo en el file system de tipo fifo. Esta operación se puede realizar directamente desde la linea de comandos con mkfifo simplemente como se muestra a continuación.

```
mkfifo -m 0660 archivofifo
```

Este comando creará un fifo de nombre “archivofifo” con permisos de lectura y escritura para el usuario que lo creó y para todos los miembros de su grupo. El 0 que antecede a los otros números sirve para que el comando se de cuenta de que lo que viene después son los permisos en forma de número en base ocho. Luego los tres números que siguen definen los permisos de la misma forma que en chmod().

La otra forma de crear un fifo y que seguramente será la más utilizada en sus proyectos de software, es la de realizar la operación desde adentro del programa con la función mkfifo() de la biblioteca stat.h.

```
mkfifo("archivofifo", 0660);
```

La única diferencia a tener en cuenta entre el comando y la función, es la forma en que se interpretan los permisos que quiero que tenga el archivo fifo. En el caso del comando la interpretación es directa y por lo tanto funciona de la misma forma que en chmod. En cambio, en la función, el modo que le pasemos a ésta será modificado según umask, que es la máscara de modo de creación de archivos del proceso. En mi sistema operativo, por ejemplo umask está configurada en 0022.

```
$ umask
0022
$
```

Los permisos que finalmente tenga el archivo en nuestro ejemplo serán configurados según la siguiente formula: 0660 & ~0022 = 0640. O sea que finalmente el archivo tendrá permisos de lectura y escritura para el usuario que lo cree y solo de lectura para el resto de su grupo, mientras que los demás usuarios no tendrán permiso de hacer nada sobre el archivo. Abajo intentaré mostrarles paso a paso cómo llega a este resultado la función mkfifo().

Operación	Cálculo
~0022	0777 - 0022 === 0755 o bien... ~000010010 ======== 111101101
0660 & ~0022	110110000 & 111101101 ======= 110100000 Este resultado es 640

Abriendo el fifo

El fifo debe ser abierto para poder acceder a él. Esto se realiza a través de la función open(). La forma más sencilla de utilizarlo es la siguiente:

```
int fd=open("fifo", O_RDONLY);
int fd=open("fifo", O_WRONLY);
```

Dependiendo si lo que quiero hacer es escribir o leer en el archivo. También puedo abrirlo de lectura y escritura especificando O_RDWR en vez de O_RDONLY por ejemplo. La función devolverá un entero que identificará al archivo fifo para siguientes accesos a él. También es posible pasarle otros flags a la función, anexados a los modos de acceso, para que los fifos se comporten de distintas formas. Un flag muy interesante es O_NONBLOCK, que hace que cuando hago una lectura, si no hay nada disponible, no se quede esperando bloqueado, sino que devuelva el control al programa informando que no había nada. Un consumidor que trabaje de esta forma debería abrir el archivo de la siguiente forma:

```
int fd=open("fifo", O_RDONLY | O_NONBLOCK);
```

Comunicando dos procesos

Finalmente, una vez creado y abierto el archivo fifo, lo único que queda es utilizarlo para escribir datos en él desde una punta y leerlos desde la otra, y esto se realiza a través de las funciones read() y write(), que son las mismas que se pueden utilizar para escribir y leer de archivos regulares.

```
write(fd, "mensajeA", sizeof("mensajeA"));
read(fd, buffer, sizeof(buffer));
```

El primer parámetro es el identificador del archivo (nuestro fifo) a utilizar. El segundo parámetro, es el mensaje a enviar y el último parámetro es la cantidad de bytes que ocupa el mensaje. La aclaración que es importante hacer acá es que el mensaje está definido como un puntero a void (const void *) con lo cual puedo, a través de casting, enviar cualquier tipo de dato. En el ejemplo utilizo una cadena de caracteres, pero podría ser un puntero a una estructura, a un vector o a cualquier otro tipo de datos. Lo único que obviamente tengo que asegurarme es que el lector utilice el mismo tipo de datos para que el mensaje sea legible.

Finalmente les voy a transcribir un ejemplo muy simple pero completo de un productor y un consumidor. Lo único que no está programado en este ejemplo es todo el tema de control de errores, por motivos didácticos. Aún así les recuerdo que esto es muy importante tanto para los objetivos de la materia como para encontrar los errores en la prueba unitaria como para darle una idea al usuario de por qué falla o no nuestro programa.

```
$ cat productor.c
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>

int main() {
        int fd=0;

        mkfifo("fifo", 0660);
        fd=open("fifo", O_WRONLY);
        write(fd, "mensajeA", sizeof("mensajeA"));
        close(fd);

        return 0;
}
$
```

```
$ cat consumidor.c
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>

int main() {
        int fd=0;
        char buffer[256];

        fd=open("fifo", O_RDONLY);
        read(fd, buffer, sizeof(buffer));
        close(fd);

        fprintf(stdout, "Mensaje: %s\n", buffer);

        return 0;
}
$
```

Fuentes

Página del manual de mkfifo(1), mkfifo(3), open(2), write(2), read(2), close(2) y umask(2).

Semáforos y memoria compartida

Objetivo

Que el alumno logre comunicar dos o más procesos a través de un segmento de memoria compartida, coordinando de forma eficaz las lecturas y escrituras en éste.

Requerimientos previos

El alumno deberá tener conocimientos del lenguaje C y manejar la funcionalidad de multiprogramación que nos brindan las funciones `fork()`, `wait()` y `waitpid()` del estándar POSIX que fueron desarrolladas en un capítulo anterior de esta guía.

Desarrollo

¿Qué es memoria compartida?

Un segmento de memoria compartida es una porción de memoria que puede ser compartida por distintos procesos. Es decir que más de un proceso puede tener mapeado un segmento de memoria compartida como propio. Ésta es lejos la forma más rápida de comunicar dos procesos, debido a que la intervención del sistema operativo es mínima. Un segmento de memoria es mapeado directamente por cada uno de los procesos que la comparten.

Funcionamiento general

Abajo se muestra un esquema temporal de cuales serían las operaciones necesarias para que se lleve a cabo una comunicación entre dos procesos utilizando memoria compartida.

Proceso 1	Proceso 2
Crea un segmento de memoria compartida	
Mapea el segmento dentro de su espacio de direccionamiento	Encuentra el segmento creado por el proceso 1
	Mapea el segmento dentro de su espacio de direccionamiento
Lee y/o escribe sobre el segmento	Lee y/o escribe sobre el segmento
Desasociar el segmento del espacio de direccionamiento	Desasociar el segmento del espacio de direccionamiento

Eliminar el segmento	

Hay que tener en cuenta que las cosas podrían cambiar un poco, por ejemplo en cuanto a que no necesariamente uno de los dos es el encargado de crear la memoria compartida. En una determinada aplicación, podría crearla el primero que llega a necesitarla. De cualquier forma nos centraremos en este esquema para mostrar el funcionamiento de este recurso.

Crear el segmento de memoria compartida

Para realizar esta operación es necesario utilizar la llamada al sistema `shmget()` cuyo prototipo es:

```
int shmget (key_t clave, int tamanio, int mcflag);
```

El primer argumento es una clave con la que se identificará al recurso compartido dentro del sistema. Tengamos en cuenta que este identificador debe ser único dentro del sistema y que además deben saberlo todos los procesos que van a compartir el recurso para poder acceder a el.

Normalmente el tipo de dato `key_t` es igual al tipo de dato `int` (creado con un typedef) y por lo tanto podríamos simplemente utilizar un número igual en todos nuestros procesos (cualquiera, podría ser el 1234567) para identificar a nuestro recurso.

Un método más portable y prolijo sería utilizar la función `ftok()` de la biblioteca `<sys/ipc.h>` para generar la clave. El prototipo de la función es el siguiente:

```
key_t ftok (char *camino, char proyecto);
```

En donde el camino es el path a un archivo del sistema y proyecto es un caracter que identifica a nuestra pieza de software. La función, con determinado algoritmo, combina el número de inodo del archivo con la letra del proyecto y genera una clave. Un ejemplo de uso sería:

```
key_t clave;
clave = ftok(“.”, 'M');  // La M es de memoria compartida
```

En este ejemplo vamos a utilizar como archivo el directorio actual y como letra de proyecto la M. Siempre que utilicemos el mismo archivo y la misma letra en todos los procesos que van a acceder a la memoria, no habrá problemas para encontrarla y accederla.

El próximo argumento de `shmget()`, es el tamaño en bytes que queremos que tenga nuestro segmento de memoria compartida.

Por último, deberemos pasarle un flag para indicarle como debe funcionar. Para esto podemos utilizar las constantes IPC_CREAT, IPC_EXCL y una máscara octal concatenadas. IPC_CREAT indica que si el segmento de memoria no existe en el sistema, lo cree. IPC_EXCL solo puede ser utilizada en conjunto con IPC_CREAT, para indicarle a la función que si ya existe un recurso en el sistema identificado con esa clave, devuelva un error. Por último la máscara octal se utiliza para decirle al sistema los permisos que queremos que tenga nuestro recurso compartido, de una forma muy parecida a como lo hacemos con el comando `chmod` para los archivos. Un ejemplo de uso de la función sería el siguiente:

```
key_t clave;
int shmid;

clave = ftok(“.”, 'M');
shmid = shmget(clave, 4096, IPC_CREAT | IPC_EXCL | 0660);
```

De esta forma le estamos pidiendo al sistema que cree un nuevo segmento de memoria compartida, que si la clave ya existe, que termine la función con error y que a ese espacio de memoria solo puedan acceder el usuario que lo creó tanto para leer como para escribir y los miembros de su grupo principal solo para leer. Esto último si umask está definida en 0022, como en mi sistema. Para una explicación más detallada de cómo interpreta los permisos la función, vean la clase de fifos, la sección “Creando un fifo”.

Encontrar el segmento ya creado

Es lo que debe hacer el proceso 2 una vez que el segmento haya sido creado en el sistema operativo. Afortunadamente esto es muy simple y se realiza de la misma forma en que se crea el segmento de memoria compartida, con la única variante de que no es necesario especificar la constante IPC_EXCL en los flags pasados a shmget(). O sea que el proceso 2 podría encontrar el segmento de memoria ya creado con la instrucción que transcribo a continuación.

```
key_t clave;
int shmid;

clave = ftok(“.”, 'M');
shmid = shmget(clave, 4096, 0660);
```

Cómo verán es muy parecido a lo que hace el proceso 1. Y aún lo podemos hacer más parecido. Podemos hacer que ambos procesos ejecuten exactamente las mismas lineas si ejecutamos shmget de la siguiente forma.

```
shmid = shmget(clave, 4096, IPC_CREAT | 0660);
```

De esta forma, el segmento es creado por el primer proceso que llame a shmget(). El segundo que llegue tomará el segmento que ya esté creado.

Mapea el segmento dentro de su espacio de direccionamiento

Hasta ahora, si bien ha sido creado el segmento de memoria compartida, nosotros solo tenemos acceso a un entero que la identifica, que en el ejemplo de la sección anterior se encuentra alojado en la variable shmid. Si queremos poder acceder a esa memoria compartida ya sea para leerla o escribirla, tenemos que tener una dirección de memoria (o sea un puntero) que nos diga donde está para poder accederla. Para que esto pueda ser posible, es necesario mapear el segmento de memoria compartida al espacio de direccionamiento del proceso. Una vez que ese mapeo se haya realizado, podemos acceder al segmento a través de su puntero como si fuera una memoria dinámica creada por un malloc() o bien un vector. La llamada al sistema que realiza esta operación tiene el siguiente formato.

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

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Cómo se puede apreciar, la función devuelve un puntero void. Esto nos permite, a través de un simple casteo, conseguir un puntero a cualquier tipo de dato y de esa forma manejar la memoria de la forma que nos resulte más cómoda. Vamos a un ejemplo un poco más concreto, siguiendo los ejemplos de secciones anteriores.

```
key_t clave;
int shmid;
char *buffer;
```

```
clave = ftok(“.”, 'M');
shmid = shmget(clave, 4096, IPC_CREAT | 0660);
buffer = (char *)shmat(shmid, NULL, 0);
```

De esta forma estaríamos creando un buffer de 4K que podría ser accedido como una cadena de caracteres dentro de nuestro programa. De la misma forma podría tener una memoria compartida para un simple entero o para contener una compleja estructura de datos, como se muestra a continuación.

```
struct mensaje_cadena {
       int codigo_comando;
       char nombre_comando;
       char lista_parametros[1017];
}

struct mensaje_cadena *mensaje;

clave = ftok(“.”, 'M');
shmid = shmget(clave, sizeof(struct mensaje_cadena), IPC_CREAT | 0660);
mensaje = (struct mensaje_cadena *)shmat(shmid, NULL, 0);
```

Noten que siempre el segundo parámetro de shmat es NULL y el tercero es 0. Esto se debe a que estos dos parámetros se utilizan para especificarle al programa cual es la dirección de memoria del proceso en donde queremos mapear el segmento en vez de que el sistema operativo nos asigne una automática y convenientemente. Afortunadamente eso está fuera del alcance de los trabajos prácticos de la materia y por lo tanto de esta guía.

Lee y/o escribe sobre el segmento

El valor devuelto por shmat es un puntero al espacio de memoria creado. Cómo la función devuelve un puntero a void, podemos castearla para acceder a la estructura de datos que creamos conveniente o que sea funcional a la lógica del programa.

```
buffer = (char *)shmat(shmid, NULL, 0);
strcpy(buffer, “este es el mensaje.”);
buffer[0]='E';
```

También podemos ver que es igual de directa la utilización del segmento con el segundo ejemplo.

```
mensaje = (struct mensaje_cadena *)shmat(shmid, NULL, 0);
mensaje->codigo_comando=34;
strcpy(mensaje->nombre_comando, “UN_COMANDO”);
```

Desasociar el segmento del espacio de direccionamiento

Cuando ya no necesito utilizar un segmento de memoria compartida, es un buena práctica quitarlo del espacio de direccionamiento de mi proceso. Eso se realiza con la función shmdt(), cuyo prototipo es el siguiente..

```
int shmdt ( char *shmaddr );
```

O sea que recibe como parámetro el puntero al segmento de memoria ya mapeado y no el identificador del segmento. Siguiendo el último de los ejemplos de la sección anterior, tendríamos que ejecutar la siguiente línea.

```
shmdt(mensaje);
```

El uso de shmdt es obligatorio en el caso de que tenga que utilizar varios segmentos de memoria compartida que podrían, sumados, exceder el espacio de direccionamiento máximo (en un kernel de 32 bits esto sería 4Gb) de un proceso. De esta forma estaría obligado a

desasociar un segmento para poder asociar el próximo segmento a utilizar.

Eliminar el segmento

Finalmente, cuando todos los procesos hayan dejado de utilizar el segmento y éste no sea necesario, debemos eliminarlo para que no siga utilizando memoria en el sistema. Esto se realiza con la función shmctl(), cuyo prototipo describimos a continuación.

```
int shmctl ( int shmid, int cmd, struct shmid_ds *buf );
```

Y específicamente para nuestro proposito, debemos utilizarla así:

```
shmctl(shmid, IPC_RMID, 0);
```

Como los más observadores habrán podido apreciar, la función sirve también para realizar otras operaciones sobre la memoria compartida. Estas operaciones podrían ser conseguir datos sobre el consumo de recursos del segmento o limitaciones del sistema, como ser el tamaño máximo de segmento que se puede crear. También se podría trabar el segmento para que no pueda ser enviado al área de intercambio (disco generalmente). El tercer parámetro de la función es una estructura que se llenaría con la información pedida, en caso de utilizar el comando apropiado.

¿Qué son los semáforos?

Los semáforos son una herramienta que nos dan la mayoría de los sistemas operativos para sincronizar la ejecución de los procesos o bien controlar el acceso a un determinado recurso.

Si el problema que tiene que resolver entra dentro de los siguientes ejemplos, lo más probable es que sea conveniente valerse de los semáforos para resolverlo:

- Asegurar la mutua exclusión entre varios procesos (utilizo este termino para referirme tanto a un proceso pesado como a un hilo de ejecución) que tienen que utilizar un recurso crítico.
- Asegurar una determinada secuencia de ejecución entre varios procesos concurrentes.
- Asegurar que varios procesos no utilicen más de la N cantidad de instancias que tiene un recurso limitado.

La idea de esta sección no es repetir lo mismo que está escrito en todos los libros de teoría de los sistemas operativos, sino simplemente decir que la idea de los semáforos es implementar las funciones P() y V() que pueden encontrar en estos textos en la sección de semáforos. La lista dada más arriba no es extensiva, sino quizá los problemas más comunes que se suelen resolver con semáforos.

Funcionamiento General

La utilización de los semáforos consiste en cinco básicas operaciones: crearlo, inicializarlo, decrementarlo (P()) o incrementarlo (V()), y finalmente eliminarlo.
Si bien la biblioteca POSIX nos da alguna funcionalidad extra, nos vamos a centrar en estas simples operaciones, que serán las que necesitaremos para realizar nuestros trabajos prácticos.

Crear un semáforo

Para crear un semáforo debemos utilizar la llamada semget(). Esto se debe a que hay que crear una estructura que maneje el kernel del SO con los datos de éste. Pensemos que la utilización de semáforos implica la realización de operaciones atómicas y el único que puede asegurarnos esa funcionalidad es el sistema operativo.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

La llamada a semget() requiere que le pasemos una clave. Esta clave, su forma de creación y su función está explicada más arriba en la sección llamada “crear un segmento de memoria compartida”. La idea es exactamente la misma, ya que se trata de un método para identificar también un recurso compartido, que en este caso es el semáforo.

El segundo parámetro que se le pasa a la función es el número de semáforos que vamos a necesitar. Esto se debe a que en realidad lo que genera la llamada semget() no es un semáforo individual, sino un conjunto de semáforos, que después podremos operar individualmente con otras llamadas al sistema que nos da la biblioteca. Hay que tener en cuenta que si el proceso está abriendo un conjunto que ya existe, este parámetro simplemente se ignora.

Con respecto al tercer argumento, son los flags que modificarán la forma en que trabaja la función. Los flags que se deben usar son IPC_CREAT e IPC_EXCL. El primero indica que si el semáforo no existe debe ser creado y el segundo, combinado con el primero, devuelve un error en caso de que el semáforo ya exista en el sistema. A estos flags también se le deben concatenar (con un “|” que funciona como un “o lógico”) los permisos que quiero que tenga mi semáforo, que funcionan de la misma manera que en la creación de un archivo. Por ejemplo, para darle permiso de lectura y escritura al usuario dueño del proceso que lo cree y a los miembros de su grupo, debo concatenar los permisos 0660 a los flags. Abajo muestro cómo sería un ejemplo de ésto.

La función devolverá el identificador del grupo de semáforos si todo sale bien, o -1 en caso de que algo salga mal y no lo pueda crear.

```
int    sid;
sid = semget( mykey, numsems, IPC_CREAT | 0660 )
```

Inicializar el semáforo

Cuando los semáforos son creados, su valor por defecto es indeterminado, según el estándar POSIX. Puntualmente la implementación de Linux inicializa los semáforos a 0 cuando son creados, pero una aplicación portable no debe confiarse de esa funcionalidad. Siempre deberíamos inicializar los semáforos al valor inicial que queremos que tengan.

Para realizar esta operación vamos a utilizar la llamada semctl(), que tiene el siguiente prototipo:

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

El primer parámetro pasado a la función es el identificador del grupo de semáforos, que es el que devuelve semget(). El segundo parámetro es el número de semáforo sobre el que queremos operar. Hay que tener en cuenta que el primer semáforo del grupo es el 0. El tercer parámetro es la operación que queremos hacer. En nuestro caso lo que queremos hacer es setearle un valor arbitrario al semáforo, y eso se hace indicando el valor SETVAL

como comando. Y finalmente el último argumento (nótese que este último parámetro es optativo) es una unión que debe ser explícitamente declarada por el programa y debe tener la siguiente estructura:

```
union semun {
               int val;
               struct semid_ds *buf;
               ushort *array;
          } arg;
```

Nosotros utilizaremos el miembro val, para setear el valor inicial del semáforo. Los otros componentes de la unión son para utilizar con otros comandos implementados en semctl(). Finalmente entonces, para configurar un mutex, indicandole un valor inicial de 1, debemos ejecutar las siguientes instrucciones:

```
arg.val = 1;
semctl(IdSemaforo, 0, SETVAL, arg);
```

Ante un error, semctl() devuelve -1. Si la llamada es exitosa, la función devuelve 0.

Operar un semáforo – P() y V()

Un semáforo debe ser operado a través de la llamada al sistema semop(), cuya estructura es la siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsops);
```

El primer parámetro sirve para especificar el grupo de semáforos a modificar, o sea el valor devuelto por semget(). El segundo parámetro, es un vector de operaciones. Con el tercer parámetro le decimos a la función la cantidad de operaciones que van en el vector. Detengámonos un minuto en la estructura sembuf. Esta consta de tres elementos, sem_num, sem_op y sem_flg. Con el primero estamos diciendo que semáforo del grupo queremos operar, con el segundo estamos especificando qué operación hacer y con el tercero modificadores para el funcionamiento de la primitiva que veremos en un momento. Si sem_op es menor que cero, estaremos pidiendo (P()) el semáforo y decrementando su valor, de acuerdo al número pasado. Si sem_op es mayor que cero, estaremos devolviendo (V()) el semáforo. Un caso particular es si en sem_op ponemos 0, ya que en este caso la llamada bloqueará al proceso hasta que el semáforo tenga un valor de 0. Por último, en sem_flg podemos especificar IPC_NOWAIT, si queremos que la operación sea no bloqueante y SEM_UNDO si queremos que la operación sea deshecha automáticamente cuando el proceso termine.

Para no marearnos tanto con las palabras, abajo les transcribo una muy sencilla implementación de las funciones P() y V() implementadas con los semáforos POSIX(). Les dejo a ustedes el control de errores y el estudio de las demás funcionalidades que nos ofrece semop().

```
void pedirSemaforo(int IdSemaforo) {
        struct sembuf OpSem;

        OpSem.sem_num = 0;
        OpSem.sem_op = -1;
        OpSem.sem_flg = 0;
        semop(IdSemaforo, &OpSem, 1);
}

void devolverSemaforo(int IdSemaforo) {
        struct sembuf OpSem;

        OpSem.sem_num = 0;
        OpSem.sem_op = 1;
```

```
        OpSem.sem_flg = 0;
        semop(IdSemaforo, &OpSem, 1);
}
```

Eliminar un semáforo

Para ser más precisos, lo que se puede eliminar no es un semáforo en particular, sino el grupo de semáforos completo. Para realizar esta operación, nos vamos a valer de una primitiva a la cual ya nos referimos antes, en la sección de inicialización de los semáforos: semctl(). En este caso solo es necesario pasarle a la función el identificador del grupo de semáforos y especificarle el comando IPC_RMID. El parámetro semnum es obviado por la función. Y
siguiendo la metodología de la sección anterior:

```
void eliminarMutex(int IdSemaforo) {
        semctl(IdSemaforo, 0, IPC_RMID);
}
```

Vale la pena notar que todos los procesos que estén bloqueados en semop() por este grupo de semáforos, serán desbloqueados y la función semop() terminará con error, devolviendo -1.

Poniendo todo junto

Vamos a resolver un problema típico de productor / consumidor en un buffer utilizando memoria compartida y semáforos, para ver como funciona todo esto.

Supongamos un productor que genera números aleatorios en un buffer y un consumidor que los lee y los consume, o sea que también tiene que escribir en el buffer para que el productor se entere de que ese elemento se ha consumido.

Para resolver el problema vamos a necesitar el buffer, que implementaremos como una memoria compartida con lugar para tres números. El productor y el consumidor no podrán operar al mismo tiempo el buffer, ya que ambos quieren escribir sobre éste. De esta forma nuestro buffer se convierte en nuestro recurso crítico. Necesitaremos un semáforo mutex, por lo tanto, para regular el ingreso a la región crítica.

También necesitaremos dos semáforos para sincronizar al productor y al consumidor. La idea es que estos semáforos no permitan que el productor pase el límite de 3 elementos en el buffer y por otro lado que el consumidor no intente sacar un elemento del buffer cuando no hay nada.

Dicho esto, vamos a poner en una tabla cómo deberían sincronizarse los procesos y como debería protegerse a la región crítica, con un mutex (M), un semáforo para proteger el límite superior (LS) de 3 elementos (inicializado en 3) y un semáforo para proteger el límite inferior (LI) de 0 elementos (inicializado en 0).

Productor	Consumidor
generar_dato()	P(LI)
P(LS)	P(M)
P(M)	sacar_dato_de_buffer()
poner_dato_en_buffer()	V(M)

V(M)	V(LI)
V(LI)	consumir_dato()

Y por último el código del ejemplo:

```
$ ls
aleatorios.c  aleatorios.h  main.c  makefile  semaforos.c  semaforos.h
$ cat makefile
.PHONY: clean
FLAGS = -ggdb

memoriaysemaforos: main.o semaforos.o aleatorios.o
	gcc $(FLAGS) -o memoriaysemaforos main.o semaforos.o aleatorios.o

main.o: main.c semaforos.h aleatorios.h
	gcc $(FLAGS) -o main.o -c main.c

semaforos.o: semaforos.c semaforos.h
	gcc $(FLAGS) -o semaforos.o -c semaforos.c

aleatorios.o: aleatorios.c aleatorios.h
	gcc $(FLAGS) -o aleatorios.o -c aleatorios.c

clean:
	rm -f *.o memoriaysemaforos
$ cat main.c
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<signal.h>
#include<stdlib.h>

void limpiarNumeros(long int *Numeros);
void ponerNumero(long int *Numeros, long int Numero);
long int sacarNumero(long int *Numeros);

int main () {
	key_t claveMutex, claveInferior, claveSuperior, claveMemoria;
	int IdMemoria, IdProceso, IdMutex, IdInferior, IdSuperior, i;
	long int *Numeros;
	long int Numero;

	/* Creo una clave, con la cual a su vez crearé una memoria compartida
	   del tamaño de un entero largo.  Luego guardaré en el puntero
	   llamado Numero la dirección de la memoria compartida. */
	claveMemoria = ftok("/", 5000);
	IdMemoria = shmget(claveMemoria, sizeof(long int[3]), IPC_CREAT | 0600);
	Numeros = shmat(IdMemoria, NULL, 0);

	/* El programa genera números aleatorios (Productor) para que otro
	   programa los vaya tomando (Consumidor) y utilizarán semáforos
	   para sincronizar esta colaboración.  En el siguiente bloque
	   se genera una semilla aleatoria, para que los números que
	   generemos luego sean realmente aleatorios. */
	inicializarAleatorios();

	/* Ahora creamos los semáforos necesarios. Uno para proteger la
	   región crítica, al que llamaremos mutex.  Otro para el límite
	   inferior.  Finalmente un último semáforo para proteger el límite
	   superior.*/
	claveMutex = ftok("/", 6000);
	claveMutex = ftok("/", 6001);
	claveMutex = ftok("/", 6002);

	IdMutex = obtenerMutex(claveMutex);
	IdInferior = obtenerSemaforo(claveInferior, 0);
	IdSuperior = obtenerSemaforo(claveSuperior, 3);

	limpiarNumeros(Numeros);

	if((IdProceso = fork()) == 0) // Hijo.  Productor.
```

```
                for(i = 0; i < 10; i++) {
                        Numero = obtenerAleatorio();
                        pedirSemaforo(IdSuperior);
                        pedirSemaforo(IdMutex);
                        ponerNumero(Numeros, Numero);
                        devolverSemaforo(IdMutex);
                        devolverSemaforo(IdInferior);
                }
        else  // Padre.  Consumidor.
                for(i = 0; i < 10; i++) {
                        pedirSemaforo(IdInferior);
                        pedirSemaforo(IdMutex);
                        Numero = sacarNumero(Numeros);
                        devolverSemaforo(IdMutex);
                        devolverSemaforo(IdSuperior);
                        printf("%ld\n", Numero);
                }

        shmdt(Numeros);
        shmctl(IdMemoria, IPC_RMID, 0);
        eliminarSemaforo(IdInferior);
        eliminarSemaforo(IdSuperior);
        eliminarSemaforo(IdMutex);

        return 0;
}

void limpiarNumeros(long int *Numeros) {
        int i;

        for(i = 0; i < 3; i++)
                Numeros[i] = -1;
}

void ponerNumero(long int *Numeros, long int Numero) {
        int i;

        for(i = 0; i < 3; i++)
                if (Numeros[i] == -1) {
                        Numeros[i] = Numero;
                        break;
                }
}

long int sacarNumero(long int *Numeros) {
        int i;
        long int Numero;

        for(i = 0; i < 3; i++)
                if (Numeros[i] != -1) {
                        Numero = Numeros[i];
                        Numeros[i] = -1;
                        break;
                }

        return Numero;
}

$ cat semaforos.h
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/stat.h>
#include<sys/sem.h>
#include<fcntl.h>
#include<stdio.h>
#include<signal.h>

union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
        struct seminfo *__buf;
};

int obtenerMutex(key_t clave);
int obtenerSemaforo(key_t clave, int valor);
```

```
void pedirSemaforo(int IdSemaforo);
void devolverSemaforo(int IdSemaforo);
void eliminarSemaforo(int IdSemaforo);
void eliminarMutex(int IdSemaforo);
$ cat semaforos.c
#include"semaforos.h"

int obtenerMutex(key_t clave) {
        int IdSemaforo;
        union semun CtlSem;

        IdSemaforo = semget(clave, 1, IPC_CREAT | 0600);
        CtlSem.val = 1;
        semctl(IdSemaforo, 0, SETVAL, CtlSem);

        return IdSemaforo;
}

int obtenerSemaforo(key_t clave, int valor) {
        int IdSemaforo;
        union semun CtlSem;

        IdSemaforo = semget(clave, 1, IPC_CREAT | 0600);
        CtlSem.val = valor;
        semctl(IdSemaforo, 0, SETVAL, CtlSem);

        return IdSemaforo;
}

void pedirSemaforo(int IdSemaforo) {
        struct sembuf OpSem;

        OpSem.sem_num = 0;
        OpSem.sem_op = -1;
        OpSem.sem_flg = 0;
        semop(IdSemaforo, &OpSem, 1);
}

void devolverSemaforo(int IdSemaforo) {
        struct sembuf OpSem;

        OpSem.sem_num = 0;
        OpSem.sem_op = 1;
        OpSem.sem_flg = 0;
        semop(IdSemaforo, &OpSem, 1);
}

void eliminarSemaforo(int IdSemaforo) {
        semctl(IdSemaforo, 0, IPC_RMID);
}

void eliminarMutex(int IdSemaforo) {
        semctl(IdSemaforo, 0, IPC_RMID);
}
$ cat aleatorios.h
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdlib.h>

unsigned int inicializarAleatorios();

long int obtenerAleatorio();
$ cat aleatorios.c
#include"aleatorios.h"

unsigned int inicializarAleatorios() {
        int IdArchivo;
        unsigned int Semilla;

         IdArchivo = open("/dev/random", O_RDONLY | 0600);
         read(IdArchivo, (void *)&Semilla, sizeof(unsigned int));
         close(IdArchivo);
         srandom(Semilla);

        return Semilla;
```

```
}

long int obtenerAleatorio() {
       return random();
}
$
```

Fuentes

The Linux Programmers Guide
Versión 0.4
1995 Sven Goldt, Sven van der Meer, Scott Burkett, Matt Welsh
http://tldp.org/LDP/lpg/lpg.html

Programming in C. UNIX System Calls and Subroutines using C.
1994-2005 Dave Marshall
http://www.cs.cf.ac.uk/Dave/C/

Unix multiprocess programming
2001 Dr. C.-K. Shene
shene@mtu.edu
http://www.csl.mtu.edu/cs4411/www/NOTES/process/process.html

The Linux Kernel
Versión 0.8-3
1996-1999 David A. Rusling
david.rusling@arm.com
http://tldp.org/LDP/tlk/tlk.html

Página del manual de semget() y semctl().

Sockets

Objetivo

Que el alumno pueda programar sistemas cliente-servidor sencillos y no tan sencillos al combinar esta clase con lo aprendido en clases anteriores de multiprogramación e IPC.

Requerimientos previos

El alumno deberá tener conocimientos del lenguaje C.

Desarrollo

Introducción

Los sockets son una forma de comunicar dos procesos a través de la red leyendo y escribiendo sobre descriptores de archivos Unix. La forma en que trabaja esta API es muy apropiada para sistemas del tipo cliente/servidor, ya que se trata de un proceso a la espera de una o varias conexiones y otro u otros procesos intentando conectarse.

Un esquema sencillo a grandes rasgos de la comunicación entre procesos a través de sockets sería el siguiente:

Servidor	Cliente
Creación de un socket de escucha	Creación del socket de comunicación
Asociación de una dirección al socket	
Habilitar el socket para recibir conexiones	
Recibir una conexión, creando un nuevo socket (de comunicación) para manejarla	Conectar al servidor
Enviar y recibir datos a través del socket de comunicación	Enviar y recibir datos a través del socket de comunicación
Cerrar el socket de comunicación	Cerrar el socket de comunicación
Recibir otra conexión, atenderla y cerrar socket de comunicación tantas veces como corresponda	
Cerrar socket de escucha	

En adelante explicaremos cómo realizar cada una de estas operaciones. Vale la pena recordar que la idea de esta guía sigue siendo dar herramientas prácticas y rápidas para la realización de los trabajos prácticos y no que el alumno comprenda plenamente todas las funcionalidades de programación para internet en UNIX. Para este fin se pueden consultar las fuentes citadas al final de la clase.

Creación de los sockets

Crear un socket significa generar un descriptor de archivo que nos sirva de interfase para la comunicación entre procesos. Para eso vamos a utilizar la llamada al sistema socket() que nos devuelve precisamente un descriptor de archivo, que recordemos que en UNIX es simplemente un entero, tal como el que devuelve la función open() que utilizamos para archivos, fifos, etc.

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

int socket(int domain, int type, int protocol);
```

El parámetro domain especifica un grupo de protocolos de comunicación. El que a nosotros nos interesa por ahora es el que identifica al grupo de protocolos de internet ipv4, por lo tanto debemos especificar AF_INET.

El parámetro type especifica el tipo de protocolo a utilizar. El que vamos a explotar inmediatamente es SOCK_STREAM, que corresponde a protocolos confiables (no se pierden paquetes, la interfase se ocupa del reenvío y control de errores), secuenciados (los paquetes se reciben en el orden que se envían), de ida y vuelta y basados en conexiones (los extremos se conocen y aceptan antes de comenzar la comunicación). El único protocolo de este tipo para el dominio AF_INET implementado en UNIX es TCP/IP. El otro tipo de protocolos que nos puede llegar a interesar más adelante es SOCK_DGRAM, cuyo único representante en UNIX es UDP/IP.

El parámetro protocolo tiene sentido cuando existe en un sistema más de un protocolo del grupo y tipo especificados antes. Para AF_INET y SOCK_STREAM, solo existe un protocolo, por lo tanto en este parámetro podemos especificar un 0.

```
int listen_socket = 0;
listen_socket = socket(AF_INET, SOCK_STREAM, 0);
```

Esta operación se realiza tanto en el cliente como en el servidor.

Asociar una dirección de internet al socket

Un equipo puede tener varias direcciones de internet, y normalmente las tiene. Puede tener varias placas de red, o un módem u otra interfaz de red que tenga una dirección accesible desde internet. Cuando programo un servidor puedo elegir una determinada interfaz para escuchar conexiones o bien permitir las conexiones desde cualquier interfaz.

Para asociar un socket a una dirección de red, necesito utilizar la función bind() cuyo protocolo les transcribo a continuación

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

El primer parámetro es el identificador del socket, el cual nos fue devuelto anteriormente por la función socket(). El segundo parámetro es el más complejo y es en el que nos vamos a detener. El tipo de dato sockaddr es un poco complejo de manejar, por eso existe la struct sockaddr_in, que es un poco menos compleja, que podemos completar y luego “castear” a sockaddr. A continuación vamos a ver como se conforma la sockaddr_in.

```
struct sockaddr_in {
  short int sin_family;      // AF_INET
  unsigned short sin_port;   // Numero de puerto.
  struct in_addr sin_addr;   // Dirección IP.
  unsigned char sin_zero[8]; // Relleno.
```

```
};

struct in_addr {
  unsigned long s_addr;      // 4 bytes.
};
```

En nuestro programa tenemos que declarar un variable de tipo struct sockaddr_in y completarla con la dirección de red y puerto en el que nuestro servidor va a estar escuchando o bien a la que nuestro cliente se va a conectar. Ahora vamos a analizar un ejemplo de como llenaríamos el dato para un servidor.

```
struct sockaddr_in listen_address;

listen_address.sin_family = AF_INET;
listen_address.sin_port = htons(5432); // Un ejemplo, podría ser otro puerto.
listen_address.sin_addr.s_addr = htonl(INADDR_ANY); // Cualquier dirección propia.
```

Cómo verán tenemos por lo menos tres novedades por aclarar. Primero vamos a explicar por qué necesitamos htons() y htonl(). Desafortunadamente cuando se desarrollo el protocolo ip y los protocolos tcp, se decidió que los enteros se representarían de forma tal que el bit *más* significativo se pondría primero, y cuando se desarrolló el i386, los diseñadores decidieron que el bit *menos* significativo sería el que se guarde primero. Las funciones htons() y htonl() transforman nuestros números (que por defecto están en formato host) al formato que reconocen los protocolos de red. Entonces htons() transforma un entero corto (short) del formato host al formato network y htonl() realiza la misma operación con los enteros largos (large).

Otra novedad es el uso de INADDR_ANY para definir la dirección de internet. INADDR_ANY se utiliza para servidores y especifica que nuestro servidor estará escuchando en todas las interfaces de red que tenga nuestro sistema, que como dijimos antes, puede y normalmente tiene varias. En el caso de que estemos escribiendo el código de un cliente, la línea debería especificar la dirección IP del servidor al que me quiero conectar y debería ser algo parecido a lo que especificamos a continuación.

```
listen_address.sin_addr.s_addr = inet_addr(“127.0.0.1”);
```

Noten que no necesitamos transformar la dirección con htonl() ya que inet_addr ya devuelve la dirección en “network byte order” por defecto.

Por último, tenemos que saber cómo hacer para acceder a un equipo cuando no tenemos su *dirección* de IP, pero si tenemos su *nombre* de red. Para esto nos vamos a valer de la función gethostbyname(), que tiene la siguiente declaración:

```
#include<netdb.h>

struct hostent *gethostbyname(const char *nombre);
```

Si le especificamos un nombre que se pueda resolver dentro de nuestra red, se completará la estructura hostent con los datos del equipo al que estamos haciendo referencia. Esta estructura tiene la siguiente declaración:

```
struct hostent {
  char  *h_name;            // Nombre oficial del equipo
  char **h_aliases;         // Lista de alias
  int    h_addrtype;        // Tipo de direcciónes (AF_INET o AF_INET6)
  int    h_length;          // El tamaño de las direcciones en bytes
  char **h_addr_list;       /* Lista de direcciones, terminadas por un puntero
                               nulo, en “network byte order”.
}
```

Pasado a código, gethostbyname() se utilizaría de la siguiente forma:

```
struct hostent *he = gethostbyname(“www.google.com”); // Por poner un ejemplo.
memcpy(&listen_address.sin_addr.s_addr, he->h_addr_list[0], he->h_length);
// La parte de memcpy() se puede hacer de varias formas.  Esta es solo una.
```

Finalmente, y volviendo a nuestra acción en el servidor de asociar una dirección de internet al socket, ahora debemos utilizar la función bind().

```
bind(listen_socket, (struct sockaddr *)&listen_address, sizeof(struct sockaddr));
```

Habilitar el socket para recibir conexiones

Un paso más de configuración de los sockets necesario antes de ponerlos a esperar conexiones es el que se realiza con la función listen cuya declaración transcribo a continuación.

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

int listen(int sockfd, int backlog);
```

Lo que hace listen() es marcar al socket como un socket pasivo, o sea un socket que se utilizará para recibir conexiones con la llamada accept(). El primer parámetro que recibe es el propio socket. El segundo parámetro especifica cuantas conexiones podrá tener el socket encoladas antes de que se atiendan con accept(). Si esta cola se hace más grande, el cliente que intente un connect() al servidor, recibirá un error de tipo “conexión rehusada”. No debemos confundir este parámetro con la cantidad de clientes que puede manejar un servidor al mismo tiempo, ya que esto depende del paralelismo del servidor. Normalmente un servidor crea un hilo de ejecución nuevo por cada cliente que atiende y vuelve a ejecutar accept() para recibir otra conexión, por lo tanto la cola de la que hablamos en “backlog” es la que se va formando si atendemos conexiones más lento de lo que se van creando.

Recibir una conexión, creando un nuevo socket (de comunicación) para manejarla

Ahora, en el servidor, tenemos que usar la función accept(), que se quedará bloqueada esperando a que lleguen las conexiones desde internet. La función tiene la siguiente estructura:

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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Esta llamada solo se utiliza en sockets basados en la conexión (SOCK_STREAM y SOCK_SEQPACKET). Lo que hace es tomar la primer conexión de la cola del socket (sockfd) y crea un nuevo socket, esta vez de comunicación, o sea que no queda marcado como pasivo. En caso de que no haya ninguna conexión en la cola, se queda bloqueado esperando que llegue una. Es posible marcar al socket como no bloqueante, pero no vamos a ver este tema en esta clase.

El segundo y tercer parámetro, sirven si quiero tener información sobre el cliente que se está conectando. Es decir que se llenarán con la información del cliente del que se está recibiendo la información. Para ver cual es la estructura de sockaddr, véase dos secciones para atrás. Si esta última funcionalidad no nos interesa, podemos especificar un 0 en ambos parámetros.

```
int comm_socket = 0;
comm_socket = accept(listen_socket, 0, 0);
```

Una vez aceptada una conexión, podemos utilizar el socket comm_socket para enviar y recibir mensajes.

Conectar al servidor

Ahora que hemos dejado al servidor en espera de conexiones, vamos a saltar al código del cliente. En éste, inmediatamente después de crear un socket (que esta vez será de comunicación y no de escuha), operación que se realiza de la misma forma que en el servidor, tenemos que directamente conectarnos al servidor. La función que utilizaremos será connect(), luego de configurar apropiadamente el socket. Veamos la declaración de la llamada.

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

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Como se puede ver, el primer trabajo que tenemos que hacer en el cliente es crear el socket con el que vamos a realizar las comunicaciones. Luego debemos llenar una estructura sockaddr con la información del servidor al que nos queremos conectar y finalmente conectarnos. Estos son todos temas desarrollados oportunamente en la explicación del servidor así que pasamos al código de ejemplo.

```
int caller_socket = 0;
struct sockaddr_in listen_address;

caller_socket = socket(AF_INET, SOCK_STREAM, 0);         // Creamos el socket.
listen_address.sin_family = AF_INET;                     // Llenamos la estructura
listen_address.sin_port = htons(5432);                   // con la info del
listen_address.sin_addr.s_addr = inet_addr("127.0.0.1"); // servidor.

connect(caller_socket, (struct sockaddr *)&listen_address, sizeof(struct
sockaddr));  // Conectamos el socket
```

Una vez que el socket está conectado, ya podemos empezar a enviar y recibir mensajes a través de éste.

Enviar y recibir datos a través del socket de comunicación

Una vez que el servidor y el cliente tienen su socket de comunicación conectado, el servidor luego del accept y el cliente luego del connect, ya pueden comunicarse a través de las primitivas recv() y send(). Las llamadas están declaradas de la siguiente forma:

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

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Por defecto ambas funciones son bloqueantes, a menos que especifiquemos MSG_DONTWAIT dentro de *flags.*

El hecho de que el parámetro buffer esté declarado en ambos casos como un puntero a void, nos permite utilizar un puntero a cualquier tipo de estructura para realizar nuestras comunicaciones. Podríamos simplemente utilizar cadenas de caracteres, como simples variables o estructuras completas, siempre y cuando le pasemos a la llamada la dirección de donde comienza el dato y el tamaño del mensaje.

Existen otros modificadores para el comportamiento de las funciones, que se pueden especificar en *flags*, pero no las vamos a cubrir en esta guía.

Un cliente podría enviar un mensaje al servidor de la siguiente forma:

```
char buffer[256];
```

```
strncpy(buffer, “Hola Sr. Servidor”, TAMBUF);
send(caller_socket, buffer, TAMBUF, 0);
```

Un servidor podría recibir el mensaje de la siguiente forma:

```
char buffer[256];

bzero(buffer, TAMBUF);
recv(comm_socket, buffer, TAMBUF, 0);
```

Cabe aclarar que los servidores también pueden ejecutar send() y viceversa. Estos son solo simples ejemplos para entender las llamadas.

Cerrar los sockets

Como todo recurso del sistema operativo, tenemos que ocuparnos de “cerrarlo” para avisarle a éste que ya puede destruir las estructuras internas que ha creado para manejarlos. Es importante entender que si no cerramos los recursos que le pedimos al sistema operativo, estos quedarán ocupando espacio innecesariamente en el sistema.

```
#include <unistd.h>

int close(int fd);
```

La llamada es la misma que se utiliza para cerrar un fifo, operación que ya vimos en otra sección de esta guía. Tambié para archivos comunes abiertos con open(). Siguiendo con nuestros ejemplos, para cerrar uno de nuestros sockets tendríamos que ejecutar:

```
close(listen_socket);
```

Un ejemplo completo

Finalmente, vamos a ver un servidor y un cliente muy simple. La idea es que el servidor se quede esperando conexiones. Solo atenderá de a una conexión por vez. Cuando el cliente se conecte, le enviará un mensaje que será leído por el servidor y mostrado por pantalla. Acá va el código del servidor:

```
#include<sys/types.h>  // socket(), bind()
#include<sys/socket.h> // socket(), bind(), inet_addr()
#include<netinet/in.h> // inet_addr()
#include<arpa/inet.h>  // inet_addr()
#include<string.h>     // bzero()
#include<stdio.h>

#define TAMBUF 1024
#define MAXQ 10
#define PORT 53210

int main() {
  int listen_socket = 0, comm_socket = 0;
  unsigned short int listen_port = 0;
  unsigned long int listen_ip_address = 0;
  struct sockaddr_in listen_address, con_address;
  socklen_t con_addr_len;
  char buffer[TAMBUF];

  // Creación del socket
  listen_socket = socket(AF_INET, SOCK_STREAM, 0);

  // Asignandole una dirección a éste
  bzero(&listen_address, sizeof(listen_address));
  listen_address.sin_family = AF_INET;
   listen_port = htons(PORT);  // Solo un ejemplo, se puede elegir cualquier otro
puerto
  listen_address.sin_port = listen_port;
  listen_ip_address = htonl(INADDR_ANY);
  listen_address.sin_addr.s_addr = listen_ip_address;
      bind(listen_socket,   (struct   sockaddr   *)&listen_address,   sizeof(struct
sockaddr));
```

```
  listen(listen_socket, MAXQ);

  printf("Comenzamos a escuchar conexiones\n");

  while (strcasecmp(buffer, "fin")!=0) {
    bzero(&con_address, sizeof(con_address));
        comm_socket = accept(listen_socket, (struct sockaddr *)(&con_address),
&con_addr_len);
    printf("Conexión recibida\n");
    bzero(buffer, TAMBUF);
    recv(comm_socket, buffer, TAMBUF, 0);
    printf("%s: %s\n", inet_ntoa(con_address.sin_addr), buffer);
    close(comm_socket);
  }

  close(listen_socket);
  printf("Se desconectó\n");
  return 0;
}
```

Como se ve en el código, este servidor no recibe parámetros y simplemente se queda esperando conexiones en todas las interfaces del equipo en el puerto 53210. El código del cliente lo vamos a ver a continuación. Para ejecutar este cliente le tenemos que pasar la dirección IP del servidor (no funciona con el nombre) y el mensaje que queremos enviar, que si es de más de una palabra tendrá que estar entrecomillado.

```
#include<sys/types.h>    // socket(), bind()
#include<sys/socket.h>   // socket(), bind(), inet_addr()
#include<netinet/in.h>   // inet_addr()
#include<arpa/inet.h>    // inet_addr()
#include<string.h>       // bzero(), strerror()
#include<stdio.h>
#include<errno.h>        // variable global errno

#define TAMBUF 1024
#define PORT 53210

int main(int argc, const char *argv[]) {
  int caller_socket = 0;
  unsigned short int listen_port = 0;
  unsigned long int listen_ip_address = 0;
  struct sockaddr_in listen_address;
  char buffer[TAMBUF];

  // Creación del socket
  caller_socket = socket(AF_INET, SOCK_STREAM, 0);

  // Asignandole una dirección a éste
  listen_address.sin_family = AF_INET;

  listen_port = htons(PORT);
  listen_address.sin_port = listen_port;

  listen_ip_address = inet_addr(argv[1]);
  listen_address.sin_addr.s_addr = listen_ip_address;

  bzero(&(listen_address.sin_zero), 8);

  // Se conecta con el servidor
     connect(caller_socket, (struct sockaddr *)&listen_address, sizeof(struct
sockaddr));

  // Enviamos el mensaje
  strncpy(buffer, argv[2], TAMBUF);
  send(caller_socket, buffer, TAMBUF, 0);

  // Cerramos el socket como corresponde.
  close(caller_socket);

  return 0;
}
```

Para que no tengan problemas al ejecutarlo, les paso una salida de mi consola. En el cliente...

```
$ ./client 192.168.106.190 "Hola Sr. Servidor"
$
```

En el servidor...

```
$ ./server
Comenzamos a escuchar conexiones
Conexión recibida
192.168.102.9: Hola Sr. Servidor
```

Fuentes

Programacion de sockets en lenguaje C
1999 - 2003 Ariel Pereira
http://www.eslinux.com/articulos/8591/programacion-sockets-lenguaje-c

Beej's Guide to network programming - Using internet sockets
2008 Brian Hall
beej@beej.us
http://www.beej.us/guide/bgnet/output/html/singlepage/bgnet.html

Páginas del manual biteorder(3), socket(2), protocols(5), getprotoent(3), bind(2), connect(2), send(2), recv(2), inet_ntoa(2), etc.

Networking programming under UNIX systems – Part II
1998 – 2002 Guy Keren
http://users.actcom.co.il/~choo/lupg/tutorials/internetworking/internet-programming.html

Threads

Objetivo

Esta guía tiene como finalidad servir de referencia rápida para el uso de Threads, según la definición de la API que realiza la extensión POSIX.1c de POSIX.

Requerimientos previos

Se presupone que el lector tiene conocimientos sobre programación en lenguaje C, teoría de procesos y threads, y uso de herramientas de programación en UNIX.

Historia

Los POSIX Threads (de aquí en adelante los llamaremos Pthreads, tal como se los conocen) nacen por la necesidad de estandarizar el uso de los threads, ya que cada fabricante desarrollaba su propia versión de threads, y no existía portabilidad de los programas entre distintas plataformas.

POSIX define una interfaz, en forma de una API, que busca lograr esta portabilidad de las aplicaciones, por lo que ha favorecido la unificación de las diferentes versiones de UNIX. De todas formas, la implementación de POSIX en cada plataforma puede tener (y tiene) variantes, sobre todo en aquellos casos donde el estándar no define muy claramente cómo implementar alguna funcionalidad, o bien el fabricante quiere sacar provecho especial de su hardware o de una funcionalidad que provee su plataforma y que el resto no tiene.

Contexto

Podemos hacer un paralelismo en el manejo y administración de los threads con los procesos, de forma que vemos claramente reflejado el concepto de “proceso liviano” (lightweight process) al que apuntan los threads. Así, pensando a los threads como procesos, pero livianos, se los puede crear, matar, esperar su finalización, consultar su estado, suspenderlos en espera de una señal, sincronizar, realizar intercambio de información (¿podríamos definirlo como IPC liviano?), etc.; con la diferencia que todas estas operaciones ocurren dentro del contexto de un único proceso. La diferencia con estos últimos es que al “vivir” dentro de un único proceso comparten los recursos del mismo (y esto incluye las áreas de memoria).

Los problemas de sincronización entre threads son, por lo tanto, un poco más sensibles que los problemas de sincronización entre procesos, debido que los threads no tienen protección sobre qué recursos están utilizando (en general, no hay un mecanismo por parte del SO que garantice la integridad de ciertas operaciones). Es decir, hay más probabilidades de generar algún error debido a la falta de sincronización de los threads, responsabilidad ésta que recae por completo en el programador.

El ejemplo más claro y sencillo es que, al compartir las áreas de memoria del proceso, dos threads independientes pueden estar leyendo y grabando una misma variable global en (casi) el mismo momento, produciendo así inconsistencias en los datos.

Otra diferencia con los procesos es que en éstos no hace falta especificarle al hijo que el proceso padre va a esperar su finalización usando `wait()`. Para los threads en cambio, debemos decidir de antemano si en algún momento posterior vamos a esperar la finalización de un thread específico, o bien el mismo se va a ejecutar en paralelo con el resto de los threads del proceso y no nos interesa en qué momento finalice. Para esto, existen dos tipos de threads: *attached* y *detached* (se interpreta como *juntos* y *separados*). Para los primeros, las rutinas de Pthread guardan el estado de finalización en sus estructuras internas, de forma que podemos esperar su finalización y consultar el resultado de su ejecución; en cambio para los *detached* no se almacena el estado de finalización, y no podemos "esperarlo".

Así, si un thread se lo crea como *attached*, podemos luego esperar su finalización desde otro thread usando `join`, pero si fue creado *detached*, no podemos saber en qué momento finalizó. En general, el default en las implementaciones es crear los threads *attached*, y luego se pueden *detached* en cualquier momento. Una vez el thread está *detached*, no se lo puede volver a attached.

Es recomendable para generar código 100% portable que siempre se inicialicen los atributos de cada objeto, independientemente cuál sea el default de POSIX. De esta forma garantizamos que en cualquier plataforma que implementa POSIX nuestro programa funcionará de la misma forma, independientemente de cómo está implementado POSIX.

Desarrollo

Para la utilización de los threads se necesita incluir la biblioteca `pthread.h`.

Los threads se identifican utilizando una estructura llamada `pthread_t` (el equivalente a `pid_t` para procesos).

Para pasarles atributos como parámetros a las funciones, se utilizan variables del tipo `pthread_attr`.

Creación

Al utilizar threads, la rutina `main()` de un programa pasa a ser automáticamente un thread, y se llama precisamente *main thread* o *thread principal*. Si el *main thread* finaliza, entonces el proceso finalizará su ejecución. Normalmente es el *main thread* encargado de lanzar los threads que realizaran el trabajo, y luego se queda esperando la finalización de los mismos para finalizar el proceso, o bien se finaliza a él mismo.

Prototipo de la función:

```
int pthread_create(pthread_t *th, pthread_attr_t *attr, *start_routine, void
*arg);
```

Argumentos:

`th` devuelve el TID (thread identification) del thread creado.

`attr` puede ser NULL en ese caso se crea un thread con los atributos defaults, sino se pasan los atributos de creación (por ejemplo para crearlo *detached*).

`start_routine` es la función de inicio del thread, su prototipo es

```
void *start_routine(void *arg)
```

`arg` se utilice para pasarle argumentos a la función de inicio.

Devolución:

0 si se puedo crear el thread, distinto de cero si hubo error.

Finalización

Un thread finaliza cuando:

- Se ejecuta el `return` de la función de inicio del thread.
- Se ejecuta la función `pthread_exit()`.
- Otro thread lo cancela llamando a la función `pthread_cancel()`.
- El proceso finaliza debido a la llamada de `exec()` o `exit`.
- El *main thread* finalizó su ejecución.

En caso que el *main thread* llame a `pthread_exit()`, si hay threads que están ejecutándose entonces el proceso sigue en ejecución. Si fuese el último thread en ejecución quien llama a `pthread_exit()`, entonces se finaliza el proceso.

Prototipo de las funciones:

```
void pthread_exit( void *retval );
```

```
int pthread_cancel( pthread_t th );
```

Argumentos:

`retval` valor de retorno del thread, es el equivalente al valor pasado a return, que luego puede ser consultado por otro thread usando `pthread_join()`.

`th` es el identificador del thread a cancelar.

Devolución:

0 si se puedo cancelar el thread, distinto de cero si hubo error.

Join

Es el equivalente de la función `waitpid()` en procesos, se utiliza la función `pthread_join()`, que (al igual que su par para procesos) es una función bloqueante, es decir que detiene la ejecución de quien la llama, en este caso, del thread llamador, y devuelve el control cuando el thread especificado finaliza.

Para poder hacer un join de un thread, este debe haber sido creado *attached*. Si el thread estuviera *detached*, entonces no podemos hacer `pthread_join()`, para esperar su finalización y conocer su estado de finalización.

Prototipo de la función:

```
int pthread_join(pthread_t *th, void **thread_return );
```

Argumentos:

`th` es el identificador del thread a esperar.

`thread_return` retorna el valor devuelto por el `th` al finalizar (es el valor que pasó al llamar `pthread_exit()`). Puede ser `NULL`, si no interesa el valor de retorno del thread.

Devolución:

0 si fue exitoso, distinto de cero si hubo error.

Mutex

Los mutexes es el mecanismo de sincronización que se utiliza en threads. Siguiendo con el paralelo con los mecanismos de los procesos, son los semáforos de los threads. La palabra mutex viene del acrónimo de *mutua exclusión*, que refleja de forma muy clara su funcionalidad. Al utilizar un mutex, garantizamos el acceso de un thread a una región

crítica, bloqueando a los otros threads que deseen acceder a la región crítica mientras está tomada por el primer thread.

Un mutex se define como una variable, y sólo un thread a la vez puede adquirir un *lock* de un mutex, y poder así acceder de forma segura a los recursos compartidos.

La secuencia típica en el uso de mutex es la siguiente:

1. Se crea e inicializa una variable mutex.
2. Los threads (uno o varios) intentan adquirir el mutex.
3. Sólo un thread es exitoso, y mutex pasa a pertenecerle. Los threads restantes quedan bloqueados en su ejecución (suspendidos), esperando la liberación del mutex
4. El thread propietario del mutex realiza las tareas sobre la región crítica
5. Desbloquea el mutex
6. Otro proceso logra adquirir el mutex y se repite el proceso (desde el paso 3)
7. Cuando no hace falta más, el mutex se destruye

Como se dijo previamente, es responsabilidad del programador asegurarse que en todos los casos en que se accede a recursos compartidos que son críticos se utilice un mutex para garantizar la consistencia de los datos.

El tipo de dato utilizado para la variable mutex es `pthread_mutex_t`.

Prototipo de la función:

```
int pthread_mutex_init( pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *mutexattr );
```

Descripción:

Inicializa `mutex` con los atributos pasados en `mutexattr`. Si `mutexattr` es `NULL`, se inicializa con los atributos por default.

También se pueden inicializar estáticamente, utilizando constantes predefinidas:

```
pthread_mutex_t mutex = CONSTANTE;
```

Donde `CONSTANTE` puede ser:

`PTHREAD_MUTEX_INITIALIZER` para mutexes rápidos

`PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP` para mutexes recursivos. El mismo thread puede adquirir múltiples veces el mutex y se va actualizando un contador de referencias, y debe realizar tantos **`pthread_mutex_unlock()`** como locks haya adquirido.

`PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP` para mutexes de verificación de errores

Argumentos:

`mutex` es la variable mutex.

`mutexattr` estructura con los atributos que desean setearse.

Devolución:

Siempre devuelve 0.

Prototipo de la función:

```
int pthread_mutex_lock( pthread_mutex_t *mutex );
```

Descripción:

Adquiere el lock sobre `mutex`.

Argumentos:

`mutex` es la variable mutex.

Devolución:

0 si fue exitoso, distinto de cero si hubo error. Si es un mutex rápido, se bloquea hasta que el mutex se libere. Si es un mutex de verificación de error, entonces retorna inmediatamente y setea `EDEADLK`.

Prototipo de la función:

```
int pthread_mutex_trylock( pthread_mutex_t *mutex );
```

Descripción:

Prueba la adquisición del lock sobre `mutex`. Es una función no bloqueante, si el mutex está tomado, no suspende al thread, retorna y continua con la ejecución.

Argumentos:

`mutex` es la variable mutex.

Devolución:

0 si fue exitoso, distinto de cero si hubo error o el mutex está bloqueado, en este caso setea `EBUSY`.

Prototipo de la función:

```
int pthread_mutex_unlock( pthread_mutex_t *mutex );
```

Descripción:

Libera `mutex`, si es del tipo rápido, si es recursivo decrementa el contador de referencias y cuando llega a cero entonces libera el mutex.

Argumentos:

`mutex` es la variable mutex.

Devolución:

0 si fue exitoso, distinto de cero si hubo error.

Prototipo de la función:

```
int pthread_mutex_destroy( pthread_mutex_t *mutex );
```

Descripción:

Destruye **`mutex`**, liberando los recursos que pudiesen estar asociados al mutex. El mutex debe estar desbloqueado.

Argumentos:

`mutex` es la variable mutex.

Devolución:

0 si fue exitoso, distinto de cero si hubo error.

Otras funciones

Para hacer *detach* de un thread que se había creado como *attached*, se usa pthread_dettach(). Una vez *detached*, no se puede volver atrás.

Prototipo de la función:

```
int pthread_detach( pthread_t th );
```

Argumentos:

`th` es el identificador del thread a hacer *detach*.

Devolución:

0 si fue exitoso, distinto de cero si hubo error.

Para obtener información del thread que se está ejecutando.

Prototipo de la función:

```
pthread_t pthread_self( void )
```

Devolución:

Devuelve una estructura `pthread_t` con la información del thread que la llama.

Para comparar los identificadores de threads, y así saber si son iguales.

Prototipo de la función:

```
int pthread_equal( pthread_t tid1, pthread_t tid2 );
```

Argumentos:

`tid1` es el identificador del primer thread a comparar.

`tid2` es el identificador del segundo thread a comparar.

Devolución:

0 si ambos TIDs son iguales, distinto de cero si son distintos.

Anexo

Convención de nombres

Prefijo de función	Grupo de funciones
pthread_	Threads en general y subrutinas varias
pthread_attr_	Atributos de objetos
pthread_mutex_	Mutex
pthread_mutexattr_	Atributos de objetos Mutex
pthread_cond_	Variables condicionales
pthread_condattr_	Atributos de variables condicionales
pthread_key_	Claves para los datos específicos del thread

Referencia de funciones

Funciones de Pthreads	
Administración de Threads	pthread_create
	pthread_exit

	pthread_join
	pthread_once
	pthread_kill
	pthread_self
	pthread_equal
	pthread_yield
	pthread_detach
Datos específicos de threads	pthread_key_create
	pthread_key_delete
	pthread_getspecific
	pthread_setspecific
Cancelación de threads	pthread_cancel
	pthread_cleanup_pop
	pthread_cleanup_push
	pthread_setcancelstate
	pthread_getcancelstate
	pthread_testcancel
Planificación de threads	pthread_getschedparam
	pthread_setschedparam
Señales	pthread_sigmask
Funciones de Atributos de Pthread	
Administración básica	pthread_attr_init
	pthread_attr_destroy
Detachable o Joinable	pthread_attr_setdetachstate
	pthread_attr_getdetachstate
Información específica del Stack	pthread_attr_getstackaddr
	pthread_attr_getstacksize
	pthread_attr_setstackaddr
	pthread_attr_setstacksize

Atributos de Planificación de Threads	pthread_attr_getschedparam
	pthread_attr_setschedparam
	pthread_attr_getschedpolicy
	pthread_attr_setschedpolicy
	pthread_attr_setinheritsched
	pthread_attr_getinheritsched
	pthread_attr_setscope
	pthread_attr_getscope
Funciones de Mutex	
Administración de Mutex	pthread_mutex_init
	pthread_mutex_destroy
	pthread_mutex_lock
	pthread_mutex_unlock
	pthread_mutex_trylock
Administración de Prioridades	pthread_mutex_setprioceiling
	pthread_mutex_getprioceiling
Funciones de Atributos de Mutex	
Administración básica	pthread_mutexattr_init
	pthread_mutexattr_destroy
Sharing	pthread_mutexattr_getpshared
	pthread_mutexattr_setpshared
Atributos de Protocolo	pthread_mutexattr_getprotocol
	pthread_mutexattr_setprotocol
Administración de Prioridades	pthread_mutexattr_setprioceiling
	pthread_mutexattr_getprioceiling
Funciones de Variables Condicionales	
Administración básica	pthread_cond_init
	pthread_cond_destroy
	pthread_cond_signal

	pthread_cond_broadcast
	pthread_cond_wait
	pthread_cond_timedwait
Funciones de Atributos de Variables Condicionales	
Administración básica	pthread_condattr_init
	pthread_condattr_destroy
Sharing	pthread_condattr_getpshared
	pthread_condattr_setpshared

Compilación

Plataforma	Comando del Compilador	Descripción
GNU/Linux	gcc -lpthread	GNU C
	g++ -lpthread	GNU C++
IBM AIX	xlc_r / cc_r	C (ANSI / non-ANSI)
	xlC_r	C++
	xlf_r -qnosave xlf90_r -qnosave	Fortran – usando la API Pthreads de IBM (no-portable)
Intel para Linux	icc -pthread	C
	icpc -pthread	C++
COMPAQ Tru64	cc -pthread	C
	cxx -pthread	C++
Otras Plataformas	gcc -pthread	GNU C
	g++ -pthread	GNU C++
	guidec -pthread	KAI C (si está instalado)
	KCC -pthread	KAI C++ (si está instalado)

Fuentes

- Distributed Systems Architecture Group. http://asds.dacya.ucm.es
- iSeries Information Center. IBM.

http://as400bks.rochester.ibm.com/html/as400/v4r5/ic2924/index.htm?info/RZAHWRZAHWCEXCODEX.HTM
- Mi Web LINUX: Tutorial pthreads. http://members.tripod.com/webprototype/tutorial_pthreads_1.html
- Operating System Examples. University of Arkansas. http://www.csce.uark.edu/~aapon/courses/os/examples/
- POSIX Threads Programming. Blaise Barney. Lawrence Livermore National Laboratory. http://www.llnl.gov/computing/tutorials/pthreads/
- A Pthreads Tutorial. Andrae Muys. New Mexico State University. http://www.cs.nmsu.edu/~jcook/Tools/pthreads/pthreads.html
- Páginas del man. Escuela Politécnica Superior, Universidad Autónoma de Madrid. http://www.ii.uam.es/~so2/pthread/index.html
- Threads: Basic Theory and Libraries. Dave Marshall. Cardiff School of Computer Science, Cardiff University http://www.cs.cf.ac.uk/Dave/C/node29.html

Créditos

Documento	Autor	Contribuyen
Estructura general de la guía	Juan Manuel Fera (jmfera@gmail.com)	Fernando Boettner (fboettner@gmail.com)
Scripts	Juan Manuel Fera (jmfera@gmail.com)	
AWK	Juan Manuel Fera (jmfera@gmail.com)	
Comando make y makefiles	Juan Manuel Fera (jmfera@gmail.com)	
Fork, wait y exec	Juan Manuel Fera (jmfera@gmail.com)	
Señales	Juan Manuel Fera (jmfera@gmail.com)	
Fifos	Juan Manuel Fera (jmfera@gmail.com)	
Semáforos y memoria compartida	Juan Manuel Fera (jmfera@gmail.com)	
Sockets	Juan Manuel Fera (jmfera@gmail.com)	
Threads	Ramiro de Lizarralde (rlizarralde@gmail.com)	

Feedback

En caso de encontrar errores en este documento, estaremos francamente agradecidos si nos lo hacen saber a yafuiste@yafuiste.com.ar