Ingeniería Técnica en Informática de Gestión

Sistemas Operativos II

Prácticas

Área de Arquitectura y Tecnología de Computadores
Escuela Universitaria Politécnica de Teruel

Luis C. Aparicio Cardiel

Área de Arquitectura y Tecnología de Computadores
EUPT: Sistemas Operativos II

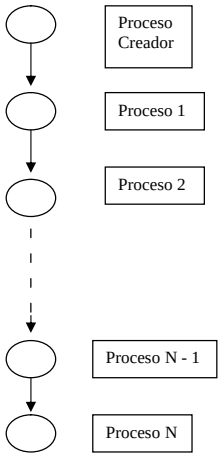
Sistemas Operativos II

Indice:

Ejercicios básicos de Sistemas Operativos I.

Problema 1

Escribe un programa en C utilizando llamadas al sistema que genere una lista de procesos en la que **como máximo haya 2 procesos a la vez en el sistema**. Cada uno de ellos debe indicar el número de orden que ocupa en la lista.



Los mensajes de una posible ejecución podrían ser:

$ listaP ↵

Soy el proceso creador 5031

Soy el nuevo proceso: 5032 mi padre es 5031
Mi numero de orden es 1
Mi padre ha finalizado, ahora soy el proceso maestro

Soy el nuevo proceso: 5033 mi padre es 5032
Mi numero de orden es 2
Mi padre ha finalizado, ahora soy el proceso maestro

Soy el nuevo proceso: 5034 mi padre es 5033
Mi numero de orden es 3
Mi padre ha finalizado, ahora soy el proceso maestro

La lista de procesos ha sido creada satisfactoriamente

Problema 2:

1º.- Comenta en detalle el siguiente programa. Utiliza la numeración de las líneas que aparece en el código para ir describiendo el programa. Haz un dibujo que muestre su funcionamiento.

2º.- ¿Cuántos procesos se crean? ¿Qué parentesco tienen entre ellos? Indica un orden de finalización de los procesos. ¿Podrías asegurar que el orden de finalización que has indicado se producirá siempre?. Razona las respuestas.

3º.- Muestra la traza de ejecución de las siguientes llamadas al programa:

Anillo 1 10

Anillo 3 10

Anillo 4 10

¿Qué hace le programa en realidad?

4º .- Modifica el programa para que ninguno de los procesos quede en estado zombie o huérfano. Cuando finalice un proceso hijo, se debe mostrar su identificador de proceso y su estado de finalización.

5º .- Supongamos que añadimos nuevo código a la función Realiza_Operacion. El tiempo de ejecución del nuevo código no es exacto y puede variar entre 3 y 5 seg.

El programa padre necesita obtener una solución antes de 10 seg. para que el resultado sea válido.

Modifica el programa de tal forma que si no se cumple el plazo de ejecución, el proceso padre finalice todos los procesos hijos pendientes y de por finalizado el programa con un mensaje de error que aparezca por la salida de error.

Nota: El proceso padre sólo debe finalizar los procesos hijos que todavía están activos en el sistema.

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

10      int Realiza_Operacion(int num_1, int num_2);

20      int main(int argc, char *argv[])
        {
30       pid_t pidaux, pidwait, hijo1, hijo2, hijo3;
40       int numero, alarma, nuevo_numero, nbytes;
50       int buf = 0, status = 0, cont_hijos_fin = 0;

60       int fpipe[3][2];

70       pipe(fpipe[0]);
80       pipe(fpipe[1]);
90       pipe(fpipe[2]);

100      numero = atoi(argv[1]);
110      alarma = atoi(argv[2]);
120      printf("\n Paso 1 del testigo: %d", numero);
130      printf("\n Tiempo máximo permitido para la ejecución %d seg.\n", alarma);
140      fflush(stdout);

150      if ((hijo1 = fork()) == 0)
         {
160       close(fpipe[0][1]);
170       close(fpipe[1][0]);
180       close(fpipe[2][0]);
190       close(fpipe[2][1]);

200       read(fpipe[0][0], &buf, 4);

210       nuevo_numero = Realiza_Operacion(numero, buf);
220       printf("\n Paso 2 del testigo: %d", nuevo_numero); fflush(stdout);

230       write(fpipe[1][1], &nuevo_numero, 4);
240       exit(1);
        }

250      if ((hijo2 = fork()) == 0)
         {
260       close(fpipe[0][0]);
270       close(fpipe[0][1]);
280       close(fpipe[1][1]);
290       close(fpipe[2][0]);

300       read(fpipe[1][0], &buf, 4);
```



```
310       nuevo_numero = Realiza_Operacion(numero, buf);
320       printf("\n Paso 3 del testigo: %d", nuevo_numero); fflush(stdout);

330       write(fpipe[2][1], &nuevo_numero, 4);
340       exit(2);
         }

350      if ((hijo3 = fork()) == 0)
         {
360       close(fpipe[0][0]);
370       close(fpipe[1][0]);
380       close(fpipe[1][1]);
390       close(fpipe[2][1]);

400       read(fpipe[2][0], &buf, 4);

410       nuevo_numero = Realiza_Operacion(numero, buf);
420       printf("\n Paso 4 del testigo: %d", nuevo_numero); fflush(stdout);

430       write(fpipe[0][1], &nuevo_numero, 4);
440       exit(3);
         }

450      sleep(1);      // Todos los procesos hijos deben estar en ejecución.

460      write(fpipe[0][1], &numero, 4);
470      close(fpipe[0][1]);

480      nbytes = read(fpipe[0][0], &buf, 4);

490      if (nbytes == 4)
500       printf("\n El resultado de la ejecución del anillo es: %d\n", buf);
510      else
520       printf("\n La ejecución del anillo ha sido INCORRECTA. \n");

530      exit(0);
        }

        int Realiza_Operacion(int num_1, int num_2)
        {
         int num_opera;

         /*   */
         // Nuevo código de la función;
         /*   */
         num_opera = num_1 * num_2;

         return(num_opera);
        }
```

Problema 3

Utilización de las llamadas al sistema exit() y wait() en procesos recursivos.

Dado el siguiente programa en C: DIMEPAR.C; se trata de descomponerlo en tres programas diferentes de manera que el funcionamiento sea equivalente.

Se escribirá un programa dimepar.c que será el programa que realice la llamada al ejecutable par o al ejecutable impar. Además informará por pantalla si el numero es par o impar.

Se escribirá un programa par.c que funcionará de forma equivalente a la función par

Se escribirá un programa impar.c que funcionará de forma equivalente a la función impar.

```
//
//                  DIMEPAR.C
//

#include <stdio.h>
#include <stdlib.h>

int par(int numpar);
int impar(int numimpar);

int main(int argc, char * argv[])
{
        int numero, result;

  numero = atoi(argv[1]);

  result = par(numero);

  if (result == 1) printf("\n El numero %d: es PAR", numero);
  else printf("\n El numero %d: es IMPAR", numero);

        return(0);
}
```



```
int par(int numpar)
{
        int presult;
        if(numpar == 0) return (1);
  else
  {
        numpar = numpar - 1;
    presult = impar(numpar);
    return(presult);
  }
}

int impar(int numimpar)
{
        int presult;
        if(numimpar == 0) return (0);
  else
  {
        numimpar = numimpar - 1;
    presult = par(numimpar);
    return(presult);
  }
}
```

Problema 4

Comenta brevemente el programa que aparece en la página siguiente.

Modifica el programa de forma que el padre escriba su mensaje después de que halla finalizado el hijo.

Modifica el programa de forma que el hijo escriba su mensaje después de que halla finalizado el padre.

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

void charatatime(char *str);

int main()
{
  pid_t pid;

  if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0) charatatime("salida del hijo \n");

  else charatatime("salida del padre \n");

  exit(0);
}

void charatatime(char *str)
{
  char *ptr;
  int c;

  setbuf(stdout, NULL);

  ptr = str;

  do
  {
    c = *ptr;
    putc(c, stdout);
  } while (*ptr++ != '\0');
}
```


Problema 5

¿Cuándo se genera la señal SIGPIPE?. ¿Qué procesos reciben esta señal?. ¿Cuál es el comportamiento por defecto de los procesos que reciben esta señal?.

Escribe un programa en el que se pueda observar claramente la generación de esta señal y su captura.

Problema 6

Utilizando llamadas al sistema UNIX, escribe un programa en C, comentado con todo detalle, que ejecute la siguiente orden:

```
sort < old.txt | wc >> new.txt
```

¿Cuál será el contenido del fichero new.txt después de ejecutar esta orden?

Práctica 0: Variables de entorno

Objetivo:

La siguiente práctica tiene como objetivo repasar los fundamentos de UNIX y conocer las variables de entorno.

0. Fundamentos de UNIX

Unix está en vías de contar con una norma común, aunque todavía existen variaciones de un fabricante a otro, en asuntos como formatos para documentación en línea, opciones de compilación de programas y localización de bibliotecas de sistemas.

¿Cómo obtener ayuda?

man <comando>

Los contenidos típicos de una tabla UNIX de páginas del manual, cada página del manual contiene algún aspecto de UNIX.

1. Comandos de usuario
2. Llamadas al sistema
3. Funciones de biblioteca de lenguaje C
4. Dispositivos e intérpretes de redes
5. Formatos de ficheros
6. Juegos y demostraciones
7. Entornos tablas y macros troff
8. Mantenimiento del sistema

Títulos:

HEADER: Título individual de una página del manual
NAME: Resumen de una línea
SYNOPSIS: Describe el uso
AVAILABILITY: Indica la disponibilidad sobre el sistema
DESCRIPTION: Describe lo que hace el comando o la función
RETURN VALUES: Los valores de regreso si son aplicables
ERRORS: Resume los valores de **errno** y las condiciones de error
FILES: Lista los ficheros de sistema que usan los comandos o funciones
SEE ALSO: Lista de otros comandos relevantes y secciones adicionales del manual
ENVIRONMENT: Lista de variables relevantes en el entorno
NOTES: Provee información de herramientas poco comunes en su uso e implemantación
BUGS: Lista de problemas conocidos y otras advertencias

Opciones interesantes:

-k man –k <comando>
HEADER relacionadas con las tres primeras secciones de las páginas del manual

-a man –a <comando>
HEADER relacionadas con todas las páginas del manual

-s man –s número ≅ número
man –s núm <comando> ≅ man núm <comando>

-f man –f <comando> ¡ ¡ pruébalo y verás ! !

Ejemplo: El comando **ps**, muestra información sobre procesos.
Resumen de los distintos campos:

F: banderas asociadas al proceso
S: estado del proceso
UID: ID del usuario propietario del proceso
PID: ID del proceso
PPID: ID del padre del proceso
C: utilización del procesador empleada para la administración del proceso
PRI: prioridad del proceso
NI: valor de amabilidad (nice value)
ADDR: dirección en memoria del proceso
SZ: tamaño de la imagen del proceso
WCHAN: dirección del evento si el proceso está suspendido
TTY: terminal de control
TIME: tiempo acumulado de ejecución
COMMAND: nombre del comando

Nota:- Este comando es diferente para los estándares POSIX.2 y Spec 1170

Los comandos de UNIX pueden tener tres clases de argumentos en línea de comandos:

- opciones: – [letras ó dígitos]
- opcion-argumentos: opciones seguidas de argumentos
- operandos: son argumentos que van seguidos de opciones u opción-argumentos.

Comandos relacionados con man:

apropos < .. >
whatis < .. >
which < .. >
find find – name / “ *.c ”

Compilación:

```
gcc –o mio mio.c

gcc –c mio.c
gcc -c milib.c
        gcc –o mio mio.o milib.o
```

Ficheros makefile

La utilidad make permite a los usuarios recompilar en forma incremental una colección de programas módulo objeto

Ejemplo:

```
Mio:    mio.o milib.o
<TAB> cc –o mio mio.o milib.o
```

Ejemplo:

```
my:     my.o mylib.o
<TAB> cc –o my my.o mylib.o

my.o:   my.c mysinc.h
<TAB> cc –c my.c
mylib.o:        mylib.c mysinc.h
<TAB> cc –c mylib.c
```

Definiciones de macros

```
OPTS = -O –H
my:     my.c my.h
<TAB> cc $(OPTS) –o my my.c
```

Actualización del objeto targer del fichero de descripción mymake

```
$ make –f mymake target
```

Nota:

"------.h"	ficheros de cabecera de usuario
<------.h>	ficheros de cabecera del sistema

Descriptores de fichero para E/S: <unistd.h> ("canales")

STDIN_FILENO	≅	0
STDOUT_FILENO	≅	1
STDERR_FILENO	≅	2

Apuntadores de fichero para E/S (ISO C) ("punteros a la estructura FILE")

```
stdin
stdout
stderr
```

Ambiente del usuario: Intérpretes de comandos UNIX:

- intérprete C csh, ejecuta comandos de arranque de un fichero llamado .cshrc

- intérprete Bourne sh, ejecuta comandos de arranque de un fichero llamado .profile

- intérprete KornShell ksh, ejecuta comandos de arranque de un fcihero llamado .profile.

¿Qué interprete de comandos usamos en "Linux"?

Variables de entorno:

```
.HOME
.SHELL
.PATH
.LONGNAME
.TERM
.USER
.MANPATH
.DISPLAY
```

Comandos para enlistar o actualizar las variables de entorno:

```
setenv
export
```

(Dependen del intérprete de comandos que se utilice).

Ejercicio 0.

Ejecuta el siguiente comando: man environ

Escribe un programa en C que muestre todas las variables de entorno que contiene la variable: extern char **environ.

Nota: Observa que la ayuda de Linux ofrece la definición de las variables de entorno.

Ejecuta el siguiente comando: man getenv

getenv - get an environment variable

#include <stdlib.h>

char *getenv(const char *name);
POSIX.1, Spec 1170

Determina si cierta variable de entorno está definida o no. El nombre de la variable de entorno debe ser pasada como una cadena de caracteres. La función getenv devuelve NULL si dicha variable no está definida. Si tiene un valor, getenv devuelve un puntero a la cadena de caracteres que contiene el valor.

Escribe un programa en C que imprima el valor de la variable de entorno DISPLAY. Modifica el programa para que imprima la variable de entorno que le pasemos como argumento al programa en su llamada.

Ejecuta el siguiente comando: env

Sirve para examinar las variables de entorno y modificarlas con el fin de ejecutar otro comando.

Cuando es invocado sin argumentos, el comando env muestra las variables de entorno en uso.

Los argumentos opcionales [name= value], indican las variables de entorno que serán modificadas.

El argumento opcional “utility” señala el comando que será ejecutado bajo el entorno modificado.

Área de Arquitectura y Tecnología de Computadores
EUPT: Sistemas Operativos II

Práctica 1: Control de Procesos

Objetivo:

La siguiente práctica tiene como objetivo repasar los elementos fundamentales sobre control de procesos en UNIX.

1. Funciones básicas para el control de procesos:

1.1 Identificadores de procesos

Todo proceso tiene asociados 6 o más identificadores asociados como se indica a continuación:

- Identificadores reales del proceso:
  real_user_ID, real_group_ID.

- Identificadores usados para verificaciones en los permisos de acceso a ficheros:
  effective_user_ID, effective_group_ID, supplementary_group_ID.

- Identificadores salvaguardados por la función exec:
  saved_set_user_ID, saved_set_group_ID.

Las funciones en UNIX que permiten obtener algunos de estos identificadores son:

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

pid_t getpid();      Devuelve: Process_ID del proceso llamador
pid_t getppid();     Devuelve: Process_ID del padre del proceso llamador
uid_t getuid();      Devuelve: Real_user_ID del proceso llamador
uid_t geteuid;       Devuelve: Effective_user_ID del proceso llamador
gid_t getgid();      Devuelve: Real_group_ID del proceso llamador
gid_t getegid();     Devuelve: Effective_group_ID del proceso llamador
```

POSIX.1, Spec 1170

Ejercicio 1.

Escribe y compila el programa prac1.c. Ejecuta el programa en los dos siguientes escenarios y explica los diferentes resultados obtenidos.

```
$ prac1
$ prac1 > temp.out
```

¿Por qué es importante que el proceso padre y los procesos hijos compartan los canales de los ficheros abiertos?

prac1.c

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

int main()
{
 int glob = 6;
 char buf[] = "Mensaje para la salida estándar \n";

 int var = 88;
 pid_t pid;

 if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
  perror("write error");
 printf("Mensaje antes de función fork \n");

 if ((pid = fork()) < 0) perror("fork error");
 else if (pid == 0)
 {
  glob++;
  var++;
 }
 else sleep(2);

 printf("pid = %d, glob = %d, var = %d \n", getpid(), glob, var);

 exit(0);
}
```

Modifica el programa para visualizar todos los identificadores de proceso posibles a través de las funciones enumeradas anteriormente y de otras que conozcas. Señala cuáles de estos identificadores son compartidos por padre e hijo. ¿Qué ventajas aporta el compartir los identificadores de proceso?

Modifica el programa para obtener el valor de las variables de entorno del padre y del hijo. Señala cuáles de estas son iguales. ¿Qué ventajas aporta la igualdad de las variables de entorno?

1.2 Funciones fork

La única manera de crear un nuevo proceso por el kernel de UNIX es mediante la ejecución de la función fork por parte de un proceso existente.

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

pid_t fork();
```

POSIX.1, Spec 1170

Devuelve: 0 en el proceso hijo, process_ID del hijo en el padre, -1 en error.

El nuevo proceso creado por fork se llama proceso hijo. Ambos procesos, padre e hijo, continúan ejecutándose a partir de la instrucción siguiente a fork. El hijo es una copia del padre. Otra característica importante de la función fork es que después de su ejecución todos los descriptores de ficheros abiertos en el proceso padre son duplicados en la tabla de descriptores de ficheros de usuario en el proceso hijo (aunque ambos comparten la misma entrada en la tabla de ficheros del sistema)

Existe otra función en UNIX que permite crear un nuevo proceso: vfork

```
# include <sys/types.h>
# include <unistd.h>
# include <vfork.h>   /* Requerido en algunos sistemas */

pid_t vfork();
```

POSIX.1, Spec 1170

Devuelve: 0 en el proceso hijo, process_ID del hijo en el padre, -1 en error.

Esta función se utiliza para crear un nuevo proceso cuando el propósito del nuevo proceso es realizar una llamada exec para ejecutar un nuevo programa. Existen algunas diferencias con fork.

vfork crea el nuevo proceso de forma análoga a como lo hace fork, pero no hace una copia completa del espacio de direcciones del padre en el hijo, ya que el hijo no quiere hacer referencia al espacio de direcciones del padre (el hijo quiere ejecutar un exec), después de la llamada vfork. Mientras el hijo está en ejecución hasta que llama a exec o a exit el hijo se ejecuta en el espacio de direcciones del padre. Esta optimización proporciona una ganancia en eficiencia.

Otra diferencia entre vfork y fork es que la función vfork garantiza que el hijo se ejecuta primero hasta que ejecuta su correspondiente exec o exit. Cuando el hijo llama a una de estas dos funciones el padre continua. Esta forma de trabajar puede dar lugar a la aparición de bloqueos si el hijo antes de realizar exec o exit está esperando alguna acción del padre.

Ejercicio 2.

Ejecuta el siguiente programa y explica los resultados obtenidos.

¿Por qué no es necesaria la instrucción sleep(2) del Ejercicio 1?

¿Qué diferencia importante hay entre el proceso padre del Ejercicio 1 y el de este ejercicio en cuanto al valor de sus variables? ¿Por qué?

Modificar el programa sustituyendo la llamada exit(0) por _exit(0) en la parte del proceso hijo. ¿Qué diferencias hay con la anterior ejecución? ¿Cuál es la razón de la diferencia observada?

prac2.c

```
#include <sys/types.h>
#include <unistd.h>
#include <vfork.h>    /* Requerido por algunos sistemas */

int main()
{
  int glob = 6;
  int var = 88;
  pid_t pid;

  printf("Mensaje antes de funcion vfork \n");

  if ((pid = vfork()) < 0) perror("vfork error");
  else if (pid == 0)
  {
    glob++;
    var++;
    printf("pid = %d, glob = %d, var = %d \n", getpid(), glob, var);
    exit(0);
  }

  printf("pid = %d, glob = %d, var = %d \n", getpid(), glob, var);

  exit(0);
}
```

1.3 Funciones exit

Existen tres formas de acabar un proceso normalmente y dos formas de acabar un proceso anormalmente:

Terminación normal:
a) Ejecución de un return desde la función main (equivalente a llamar a exit)
b) Llamar a la función exit que llama a los manejadores de salida registrados a través de la función atexit y cierra todos los streams standard de E/S.
c) Llamar a la función _exit que es llamada al final por exit.

Terminación anormal:
a) Ejecución de abort.
b) Cuando el proceso recibe ciertas señales (signals). Las señales pueden ser generadas por el propio proceso, por algún otro proceso o por el propio kernel (ejemplos de señales generadas por el kernel son: señales generadas cuando un proceso referencia una posición de memoria fuera de su espacio de direcciones o señales generadas en una división por cero).

Independientemente de cómo termina un proceso, el código del kernel ejecutado es el mismo. Este código cierra todos los descriptores abiertos por el proceso, libera la memoria que estaba usando.

En cualquiera de los casos precedentes es deseable que el proceso que termina notifique al proceso padre cómo terminó el proceso. Las funciones exit y _exit realizan esto mediante el exit_status que pasan como argumento de las funciones. En el caso de una terminación anormal, el kernel (no el proceso) genera un status de terminación para indicar la razón de la terminación anormal. En cualquier caso, el padre del proceso puede obtener el estatus de terminación a través de las funciones wait o waitpid descritas más adelante.

Nótese que hay que diferenciar entre el “exit_status” que es el argumento de cualquiera de las funciones exit o _exit, o el valor del return desde main “termination_status”. El “exit_status” se convierte en el “terminacion_status” a través del kernel cuando se llama finalmente a la función _exit.

Después de la ejecución de un fork es bastante obvio que el proceso hijo tiene un padre. Pero en el caso de que el proceso padre termine antes que el proceso hijo esta relación de filiación no queda clara. Lo que ocurre es que en el sistema existe un proceso especial denominado init que se convierte en el proceso padre de aquellos procesos cuyo proceso padre terminó antes que ellos. Se dice que el proceso init heredó al proceso huérfano. Lo que sucede normalmente es que siempre que un proceso termina el kernel revisa todos los procesos activos para determinar si el proceso que ha terminado es padre de algún proceso que todavía existe. Si es así, el identificador del proceso padre de los procesos hijos existentes se cambia al valor 1 que es el identificador del proceso init. De esta manera se garantiza que todo proceso en el sistema tiene un padre.

Otra condición que hay que tener en cuenta se refiere al caso que un proceso hijo termine antes que el proceso padre. Si el hijo desapareciera completamente del sistema, el padre no sería capaz de encontrar su “terminación_status” cuando, si es que lo hace, el padre estuviera finalmente listo para analizar si el hijo ha terminado. En este caso el kernel tiene que mantener una cierta cantidad de información referente a todos aquellos procesos que han terminado de forma que esta información se encuentre disponible en el momento en el que el proceso padre llame a las funciones wait o waitpid. En su forma más reducida, esta información consta del identificador del proceso terminado, el “termination_status” del proceso y la cantidad de tiempo de CPU consumida por el proceso. No obstante el kernel ha liberado toda la memoria ocupada por el proceso terminado y cierra todos sus ficheros abiertos. En terminología UNIX el proceso que ha terminado, pero cuyo proceso padre todavía no ha llegado a la correspondiente función wait, se llama proceso zombie. La orden UNIX ps(1) muestra el estado de un proceso zombie con Z. Si se escribe un programa muy largo que crea muchos procesos hijos, a menos que se llame a la función wait para cada uno de ellos para buscar su “termination_status”, estos procesos hijos llegarán a ser zombies.

Una consideración final acerca de lo que sucede con un proceso heredado por el proceso init cuando termina. Estos procesos no se convierten en zombies ya que init está escrito de tal forma que siempre que uno de sus hijos termina, init llama a una de las funciones wait para obtener su “termination_status”. Al hacer esto el proceso init se evita que el sistema quede plagado de zombies.

1.4 Funciones wait

Cuando un proceso termina el kernel se lo notifica al padre mediante el envío de la señal SIGCHLD. Dado que la terminación de un proceso hijo es un evento asíncrono, esta señal es la notificación asíncrona del kernel al proceso padre. El padre puede ignorar esta señal o puede proveer una función que es llamada cuando ocurre la señal (un "signal handler"). La acción por defecto para esta señal es ignorarla. Un proceso que llama a las funciones wait o waitpid puede encontrarse en una de las tres siguientes situaciones:

a) Bloqueado, todos sus hijos están todavía en ejecución
b) Regreso inmediato de la función con el "termination_status" de un hijo, en el caso de que un hijo haya terminado y el padre esté esperando su "termination_status".
c) Regreso inmediato de la función con un error, si no existe ningún proceso hijo.

Si un proceso está llamando a la función wait porque recibió la señal SIGCHLD, se espera que la función wait retorne inmediatamente. Pero si se llama a la función en un momento arbitrario de tiempo, el proceso se puede bloquear.

wait, waitpid - wait for process termination

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

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
                        POSIX.1, Spec 1170
   Devuelven: Process_ID si es correcto, o, o –1 si error.
```

Las diferencias entre estas dos funciones son:

a) wait puede bloquear al proceso llamador hasta que un proceso hijo termine, mientras que waitpid tiene una opción que evita que el proceso se bloquee.
b) waitpid no espera la terminación del primer proceso hijo para terminar, tiene un conjunto de opciones que controlan cuál es el proceso hijo que se espera que termine.

Si el hijo ya ha terminado y es un proceso zombie, wait regresa inmediatamente con el status del hijo. En otro caso el proceso llamador se bloquea hasta que un hijo termina. Si el llamador se bloquea y tiene múltiples hijos, wait regresa cuando uno de ellos termina. Se puede identificar el proceso hijo que terminó por que la función devuelve su identificador de proceso.

Para ambas funciones el argumento status es un puntero a un entero. Si este argumento no es un puntero nulo, el "termination_status" del proceso terminado se almacena en la posición apuntada por el argumento. Si no se está interesado en el "termination_status", se pasa un puntero nulo como argumento.

Normalmente el entero que representa el status que es devuelto por estas dos funciones es dependiente de la implementación y se suele acompañar con una serie de bits que indican el status del exit, para regresos normales, bits que indican el número de señal, para regresos anormales, un bit indica si se ha genera un fichero "core", y así sucesivamente. POSIX .1 especifica que el "termination_status" debe ser analizado a través de un conjunto de macros que se encuentran definidas en <sys/wait.h>. Existen tres macros mutuamente exclusivas que permiten determinar como ha terminado un proceso, su nombre empieza por WIF. Una vez determinado cuál de estas tres macros devuelven un valor true, se pueden utilizar otras macros que permiten obtener el "exit_status", numero de señal, y así sucesivamente. La tabla siguiente describe someramente estas macros.

WIFEXITED (status)
Valor cierto si el status devuelto por un proceso hijo que terminó normalmente. En este caso se puede ejecutar WEXITSTATUS (status) para buscar los 8 bits más bajo del argumento que el hijo pasó a exit o _exit.

WIFSIGNALED (status)
Valor cierto si el status fue devuelto por un hijo que terminó anormalmente, por recepción de una señal que no pudo capturar. En este caso se puede ejecutar WTERMSIG (status) para buscar el número de señal que causó la terminación. Adicionalmente, SVR4 y $.3+BSD definen la macro WCOREDUMP (status) que devuelve valor cierto si se generó un fichero core del proceso terminado.

WIFSTOPPED (status)
Valor cierto si el status fue devuelto por un proceso hijo que se encuentra en ese momento detenido. En este caso se puede ejecutar la macro WSTOPSIG (status) para buscar el número de la señal que causó la detención del proceso hijo.

Ejercicio 3.

Ejecuta el siguiente programa y explica los resultados obtenidos.

prac3.c

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

void pr_exit(int estado);

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

```
  if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0) exit(7);

  if (wait(&status) != pid) perror("wait error");
  pr_exit(status);

  if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0) abort();

  if (wait(&status) != pid) perror("wait error");
  pr_exit(status);

 if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0) status = status / 0;

 if (wait(&status) != pid) perror("wait error");
  pr_exit(status);

  exit(0);
 }

 void pr_exit(int estado)
 {
  if (WIFEXITED(estado)) printf("Terminación normal, exit status = %d\n",
WEXITSTATUS(estado));
  else if (WIFSIGNALED(estado))
  {
   printf("Terminación anormal, numero de signal = %d ",
WTERMSIG(estado));
   if (WCOREDUMP(estado)) printf("(se genero fichero core)\n");
   else printf(" \n");
  }
  else if (WIFSTOPPED(estado)) printf("Hijo detenido, numero de signal =
%d\n", WSTOPSIG(estado));
 }
```

La función wait regresa cuando uno cualquiera de los procesos hijos termina. Cuando se desea esperar a que un proceso hijo específico termine y algunas posibilidades adicionales se puede usar la función waitpid.

La interpretación del argumento pid de la función waitpid depende de su valor:

pid = -1 Espera hasta que termine cualquier proceso hijo. (equivalente a wait)
pid > 0 Espera hasta que termine el proceso hijo cuyo ID de proceso sea igual a pid.
pid = 0 Espera hasta que termine cualquier proceso hijo cuyo ID de grupo sea igual al del proceso llamador.
pid < -1 Espera hasta que termine cualquier proceso hijo cuyo ID de grupo sea igual al valor absoluto de pid.

waitpid devuelve el ID del proceso hijo que terminó, y su “termination_status” es devuelto a través de status. Con la función wait el único error posible aparece cuando el proceso llamador no tiene procesos hijos. En el caso de la función waitpid, puede aparecer un error también en el caso que el proceso especificado o el grupo de procesos especificado no exista o no sea hijo del proceso llamador.

El argumento options es un control adicional que permite la función waitpid. Este argumento es 0 o es una operación OR entre las constantes que se definen a continuación:

Constante WNOHANG: La función waitpid no se bloquea en el caso de que el hijo especificado por pid no se encuentre disponible. En este caso el valor devuelto por la función es 0.

Constante WUNTRACED: Si la implementación soporta control de trabajos, se devuelve el status de cualquier proceso hijo especificado a través de pid que ha sido detenido, y cuyo status no ha sido comunicado desde que fue detenido.

La macro WIFSTOPPED determina si el valor devuelto corresponde a un proceso hijo detenido.

La función waitpid proporciona tres características más que la función wait.
a) waitpid permite espera la terminación de un proceso específico
b) waitpid proporciona una versión no bloqueante de función wait
c) waitpid soporta control de trabajos a través de la opción WUNTRACED

Ejercicio 4.

Determina si el siguiente programa crea un proceso zombie o no.

prac4_1.c

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

int main()
{
 pid_t pid;

 if ((pid = fork()) < 0) perror("fork error");
 else if (pid == 0) exit(0);
 sleep(4);
 system("ps -l");
 exit(0);
}
```

Ejecuta el siguiente programa y explicar su funcionamiento. ¿Se crean procesos zombies en este caso? ¿Por qué? ¿Cuál es la misión de la sentencia sleep(2) en el segundo hijo? ¿Se puede decir a priori qué proceso se ejecutará primero después del un fork?

prac4_2.c

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

int main()
{
  pid_t pid;

  if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0)
  {
    if ((pid = fork()) < 0) perror("fork error");
    else if (pid > 0) exit(0);

    sleep(2);
    printf("Second child, parent pid = %d \n", getppid());
    exit(0);
  }

  if (waitpid(pid, NULL, 0) != pid) perror("waitpid error");
  exit(0);
}
```

El sistema operativo 4.3+BSD proporciona dos funciones adicionales de tipo wait: wait3 y wait4. La única característica adicional que proporciona estas funciones respecto a las funciones wait anteriormente descritas es que poseen un argumento adicional que permite al kernel devolver un resumen de los recursos utilizados por el proceso terminado y todos sus procesos hijos.

wait3, wait4 - wait for process termination, BSD style

```
#define _USE_BSD
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3(int *status, int options, struct rusage *rusage)
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage)
```

POSIX.1, Spec 1170

Devuelven: Process_ID si es correcto, o, o –1 si error.

La información sobre recursos incluye información tal como cantidad de tiempo de CPU de usuario, tiempo de CPU de sistema, número de fallos de página, número de señales recibidas, etc.

Analiza la función **getrusage(2)** en el manual para obtener más detalles sobre la información de recursos. Esta información de recursos está disponible para procesos hijos terminados no para procesos hijos detenidos.

A continuación se explican algunos detalles sobre las **denominadas condiciones de carrera** (race conditions). Una condición de carrera se presenta cuando múltiples procesos están tratando de hacer algo con datos compartidos y el resultado final depende del orden en el que los procesos se ejecutan. La función fork es una de las funciones que puede dar lugar a condiciones de carrera si el código que aparece después de la ejecución de la función, implícita o explícitamente, depende de que el proceso padre o hijo se ejecute primero después del fork. En general no se puede predecir que proceso se ejecutará primero. Incluso aunque se conociese que proceso se ejecutará primero, lo que suceda después de que el primer proceso comience a ejecutarse depende de la carga del sistema y del algoritmo de planificación de procesos dentro del sistema. Los problemas derivados de las condiciones de carrera son difíciles de detectar puesto que dependen del entorno de ejecución.

Si un proceso quiere esperar a que termine un proceso hijo, éste debe utilizar una llamada a una función wait.

¿Qué debe hacer un proceso hijo si quiere espera a que el proceso padre termine?

El problema de esta clase de bucles (denominados polling) es que consumen inútilmente tiempo de CPU. Para evitar condiciones de carrera y bucles de espera como los anteriores se precisa el uso de señales entre múltiples procesos. También se pueden usar varias técnicas de comunicación entre procesos (IPC).

Ejercicio 5.

El programa presentado a continuación pretendía que el proceso padre escribiese “salida del padre” y a continuación el proceso hijo escribiese “salida del hijo”.

Verifica que existe una condición de carrera.

Modifica el programa de forma que el padre escriba primero su mensaje completo y a continuación el hijo escriba su mensaje.

Modifica el programa de forma que el proceso hijo escriba primero su mensaje completo y a continuación el padre escriba su mensaje.

prac5.c

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

void charatatime(char *str);

int main()
{
  pid_t pid;

  if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0) charatatime("salida del hijo \n");

  else charatatime("salida del padre \n");

  exit(0);
}

void charatatime(char *str)
{
  char *ptr;
  int c;

  setbuf(stdout, NULL);

  ptr = str;

  do
  {
    c = *ptr;
    putc(c, stdout);
  } while (*ptr++ != '\0');
}
```


1.5 Funciones exec

La creación de un proceso hijo mediante la función fork con el objetivo de ejecutar un programa distinto del asociado al proceso padre requiere la utilización de una de las funciones exec. Cuando un proceso llama a una de las funciones exec, este proceso es completamente reemplazado por el nuevo programa, y el nuevo programa comienza a ejecutarse a partir de su función main. El identificador del proceso no cambia como consecuencia de la ejecución de la función exec porque no se ha creado un nuevo proceso, exec exclusivamente reemplaza el proceso actual, sus segmentos de código, datos, heap y pila con un nuevo programa extraído de disco.

Existen 6 funciones diferentes exec. Estas seis funciones exec completan las primitivas de control de procesos de UNIX. Con fork se crean nuevos procesos, y con las funciones exec se pueden iniciar nuevos programas. La función exit y las dos funciones wait nos permiten controlar la terminación de procesos y la espera por la terminación de procesos respectivamente.

execl, execlp, execle, execv, execvp, execve - execute a file

```
#include <unistd.h>

int execl(const char *path, const  char  *arg0,  ...,  const char *argn, char *
/*NULL*/);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char  *arg0,  ...,  const char *argn, char *
/*NULL*/, char *const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char  *arg0,  ...,  const char *argn, char *
/*NULL*/);
int execvp(const char *file, char *const argv[]);
```

POSIX.1, Spec 1170

Devuelven: -1 si error, no regresan si no error

La primera diferencia entre estas funciones es que las cuatro primeras tienen un primer argumento que es el pathname, mientras que las dos últimas tiene como primer argumento el nombre de un fichero. Cuando se especifica un nombre de fichero, si el nombre del fichero contiene el carácter “/”, este nombre es interpretado como un pathname, en cualquier otro caso, el fichero ejecutable se busca en los directorios especificados por la variable de entorno PATH.

La variable de entorno PATH contiene una lista de directorios denominada prefijos de paths que están separados por el carácter “:”; por ejemplo, la siguiente cadena de entorno especifica cuatro directorios donde buscar:

PATH = /bin: /usr/bin: /usr/local/bin/ : .

Si las funciones execlp o execvp, encuentra un fichero ejecutable utilizando uno de los prefijos de path, pero el fichero no es un fichero de código máquina que haya sido generado por el editor de enlaces, estas funciones suponen que es un “shell script” y tratan de invocar /bin/sh con el nombre del fichero como entrada del shell.

La siguiente diferencia entre las funciones exec se refiere a la forma de pasar la lista de argumentos, “l” se utiliza para indicar lista y “v” se utiliza para indicar vector. Las funciones execl, execlp y execle requieren la especificación de cada uno de los argumentos de la línea de órdenes para el nuevo programa como un argumento separado. El fin de la lista de argumentos se marca con un puntero nulo. Para las otras tres funciones execv, execvp y execve se construye un vector de punteros a los argumentos, y la dirección de este vector es el argumento de estas tres funciones. Los argumentos de la línea de orden para las tres funciones execl, execlp y execle es:

char *arg0, char *arg1, … … …, char *argn, (char *) 0

La diferencia final entre las seis funciones se refiere a la forma de pasar la lista de entorno al nuevo programa. Las dos funciones cuyo nombre termina con una “e”, execle y execve permiten pasar un puntero a un vector de punteros a las cadenas de entorno. Las otras cuatro funciones, utilizan la variable environ del proceso llamador para copiar el entorno existente para el nuevo programa. Normalmente un proceso permite propagar su entorno a sus hijos, pero hay casos en los que un proceso quiere especificar un determinado entorno para un proceso hijo. Un ejemplo de esto último es el programa login cuando se inicia un nuevo login shell. Los argumentos, por ejemplo, de la función execle son:

char *pathname, char *arg0, char *arg1, … , char argn, (char *)0, char *envp[]

El identificador de proceso no cambia después de la ejecución de la función exec. Existe un gran número de propiedades adicionales que el nuevo programa hereda del proceso llamador de la función exec, como por ejemplo: identificador del proceso, identificador del proceso padre, identificador del grupo de proceso, identificador de sesión, control del terminal, directorio actual de trabajo, directorio raíz, señales pendientes, límites de recursos, bloqueos de ficheros, máscara para los modos de creación de ficheros, etc.

El manejo de los ficheros abiertos depende del valor del close_on_exec flag para cada descriptor (flag FD_CLOEXEC) Todo descriptor abierto en un proceso tiene uno de esos flags. Si este flag está a uno, el descriptor se cierra al ejecutar la función exec. La opción por defecto es dejar el descriptor abierto después de la ejecución de exec, a menos que se ponga a uno el close_on_exec flag utilizando la función fcntl.

fcntl - manipulate file descriptor

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

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock * lock);
```

Notar que en muchas implementaciones UNIX sólo una de estas 6 funciones, execve, es una llamada al sistema dentro del kernel. Las otras cinco son funciones de biblioteca que eventualmente invocan a esta llamada al sistema.

Ejercicio 6.

Construye el siguiente programa que será ejecutado posteriormente a través de diferentes llamadas a las funciones exec.

prac6_1.c → echoall

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;
  char **ptr;
  extern char **environ;

  for (i = 0; i < argc; i++) printf("argv[%d]: %s \n", i, argv[i]);

  ptr = environ;

  while (*ptr != NULL)
  {
    printf("%s \n", *ptr);
    ptr++;
  }

  exit(0);
}
```

Ejecuta el siguiente programa y explica los diferentes resultados obtenidos.

prac6_2.c

```
/*     prac6_2.c     */

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

int main()
{
  char *env_init[] = {"USER=unknown", "PATH=/tmp", NULL};

  pid_t pid;

  printf("\n Inicio del primer hijo \n\n");
  fflush(stdout);

  if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0)
  {
    if (execle("/export/home0/practicas/atc/soii/echoall", "echoall", "my arg1",
"MY ARG2", (char *) 0, env_init) < 0) perror("execle error");
  }

  if (waitpid (pid, NULL, 0) < 0) perror("wait error");

  printf("\n\n Finalización del primer hijo \n\n\n");
  printf("\n Inicio del segundo hijo \n\n");
  fflush(stdout);

  if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0)
  {
    if (execlp("echoall", "echoall", "only 1 arg", (char *) 0) < 0) perror("execlp
error");
  }

  if (waitpid (pid, NULL, 0) < 0) perror("wait error");

  printf("\n\n Finalización del segundo hijo \n\n\n");
  fflush(stdout);

  exit(0);
}
```


Por último en este apartado se recomienda vivamente el investigar el funcionamiento y el significado de las siguientes funciones relacionadas con el cambio de los identificadores de usuario y de grupo de un proceso.

setuid - set user identity

```
#include <unistd.h>
int setuid(uid_t uid);
```

setgid - set group identity

```
#include <unistd.h>
int setgid(gid_t gid)
```

setreuid, seteuid - set real and / or effective user ID

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int seteuid(uid_t euid);
```

setregid, setegid - set real and / or effective group ID

```
#include <unistd.h>
int setregid(gid_t rgid, gid_t egid);
int setegid(gid_t egid);
```

Muy recomendable: Escribir un programa en el que se vea claramente el funcionamiento de estas funciones.

Práctica 2: Medición sobre procesos

Objetivo:

La siguiente práctica tiene como objetivo repasar los elementos fundamentales sobre control de procesos en UNIX.

2. Mediciones sobre procesos

Antes de introducir ciertas mediciones sobre procesos recordaremos una función bastante útil a la hora de programar: la función system. Esta función se usa para ejecutar una cadena de órdenes desde dentro de un programa. Por ejemplo, supóngase que se quiere imprimir la hora y fecha actual en un cierto fichero como marca de una cierta acción. Para ello se podrían usar las funciones time para obtener la hora y la fecha, a continuación llamar a la función localtime para convertir la hora, llamar a la función strftime para formatear los resultados y por fin escribir el resultado en un fichero. Desde un punto de vista de programación es mucho más fácil realizar esta tarea a través de la siguiente llamada a la función system: system("date > nombre_fichero");

ANSI C define la función system, pero su operación es fuertemente dependiente del sistema. Esta función no está definida por POSIX.1 ya que realmente no es una interfaz del S. O., sino que realmente es una interfaz con el shell. No obstante en la propuesta de POSIX.2 se está procediendo a su estandarización.

system - execute a shell command

#include <stdlib.h>

int system (const char * string);

POSIX.2

Si string es un puntero nulo, system devuelve un valor distinto de cero solo si un procesador de órdenes está disponible. Esta característica permite determinar si la función system es encuentra soportada en un S. O. determinado.

La función system se implementa haciendo uso de las funciones fork, exec y waitpid. Debido a esto, existen tres diferentes tipos de valores devueltos por la función:

a) Si la función fork falla o waitpid devuelve un error diferente que EINTR, la función system devuelve –1 con errno indicando el error.
b) Si exec falla (implicando que el shell no se puede ejecutar) el valor devuelto es como si el shell hubiera ejecutado exit(127).
c) En cualquier otro caso, las tres funciones se ejecutan correctamente (fork, exec y waitpid), el valor devuelto por la función system es "termination_status" del shell, en el formato especificado por waitpid.

Ejercicio 7.

El siguiente programa ilustra la utilización de la función system. Ejecútalo con diferentes ordenes UNIX y explica los comportamientos observados.

prac7.c

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

void pr_exit(int estado);

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

 if (argc < 2)
 {
  perror("Se requiere como argumento una línea de orden");
  exit(1);
 }

 if ((status = system(argv[1])) < 0) perror ("system() error");
 pr_exit(status);

 exit(0);
}

void pr_exit(int estado)
{
 if (WIFEXITED(estado)) printf("Terminación normal, exit status = %d\n",
WEXITSTATUS(estado));
 else if (WIFSIGNALED(estado))
 {
  printf("Terminación anormal, numero de signal = %d ",
WTERMSIG(estado));
  if (WCOREDUMP(estado)) printf("(se genero fichero core)\n");
  else printf(" \n");
 }
 else if (WIFSTOPPED(estado)) printf("Hijo detenido, numero de signal =
%d\n", WSTOPSIG(estado));
}
```

Ejercicio 8.

Escribe una implementación de la función system de acuerdo con las especificaciones anteriores utilizando las funciones fork, execl y waitpid. Llamar a ejecución al shell con la opción –c para que coja la siguiente línea de orden

```
int mi_system(char *cmdstring);
```

Ejercicio 9.

Prueba la implementación de la función system del ejercicio 8 mediante el siguiente programa. Comenta los resultados obtenidos.

prac9.c

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

int mi_system(char *cmdstring);

void pr_exit(int estado);


int main()
{
  int status;

  if ((status = mi_system("date")) < 0) perror("system() error");
  pr_exit(status);

  if ((status = mi_system("nosuchcommand")) < 0) perror("system() error");
  pr_exit(status);

  if ((status = mi_system("who; exit 44")) < 0) perror("system() error");
  pr_exit(status);

  exit(0);
}
```

La ventaja de utilizar la función system en lugar de utilizar fork y exec directamente, es que system realiza todas las funciones necesarias para manejar los errores.

2.1 Medias sobre procesos

Históricamente existen dos tipos diferentes de valores de tiempo mantenidos por los sistemas UNIX:

a) Tiempo de calendario. Este valor cuenta el número de segundos desde las 00:00:00 horas del 1 de enero de 1970 (Epoch), que se denomina Coordinated Universal Time. (UTC) Estos valores de tiempo se usan para registrar el momento en el que un fichero fue modificado por última vez, por ejemplo. El tipo de datos primitivo del sistema time_t almacena estos valores de tiempo.

b) Tiempo de proceso. Se denomina también tiempo de CPU y mide los recursos del procesador central utilizados por un proceso. El tiempo de proceso se mide en ticks de reloj, que históricamente han sido 50, 60 o 100 ticks por segundo. El tipo de datos primitivo del sistema clock_t mantiene estos valores de tiempo. Posteriormente, POSIX define la constante CLK_TCK para especificar el número de ticks por segundo.

Cuando se mide el tiempo de ejecución de un proceso, UNIX mantiene tres valores diferentes: tiempo de reloj, tiempo de CPU de usuario, tiempo de CPU de sistema.

El tiempo de reloj (también denominado “wall clock time”) es la cantidad de tiempo que consumió un proceso durante su ejecución. Su valor depende del número de procesos en ejecución en el sistema. Las medidas basadas en el tiempo de reloj que se presentan en algunos trabajos, siempre se realizan cuando no existe ninguna otra actividad en el sistema.

El tiempo de CPU de usuario es el tiempo de CPU atribuible a la ejecución de instrucciones de usuario. El tiempo de CPU de sistema es el tiempo de CPU atribuible al kernel cuando ejecuta funciones para el proceso. Por ejemplo, siempre que un proceso ejecuta un servicio del sistema, como read o write, el consumo de tiempo dentro del kernel realizando estos servicios del sistema es contabilizado en el tiempo de CPU de sistema de este proceso. La suma del tiempo de CPU de usuario y de sistema se denomina normalmente tiempo de CPU de un proceso.

El tiempo de calendario se puede obtener a través de la siguiente función:

time - get time in seconds

```
#include <time.h>
time_t time(time_t *t);
                                POSIX.1
```

Devuelven: -1 si error, no regresan sino error

El valor de tiempo se transmite como el valor de la función. Si el argumento es no nulo, el valor de tiempo se almacena también en la posición apuntada por t.

La función anterior devuelve un valor entero grande que representa el número de segundos transcurridos desde Epoch. En este punto es necesario llamar a una serie de funciones adicionales que son capaces de convertir este entero en una serie de datos legibles sobre la hora y fecha. Las dos funciones localtime y gmtime convierten el tiempo de calendario obtenido por time en una estructura tm como la descrita a continuación:

```
struct tm
        {
            int     tm_sec;         /* seconds: [0, 61] */
            int     tm_min;         /* minutes: [0, 59] */
            int     tm_hour;        /* hours: [0, 23] */
            int     tm_mday;        /* day of the month [1, 31] */
            int     tm_mon;         /* month [0, 11] */
            int     tm_year;        /* year: años desde 1900 */
            int     tm_wday;        /* day of the week: desde el domingo [0, 6] */
            int     tm_yday;        /* day in the year: desde el 1 de enero [0,365] */
            int     tm_isdst;       /* flag de la salvaguarda de tiempo */
        };
```

gmtime, localtime - transform binary date and time to ASCII

```
#include <time.h>
struct tm *gmtime(const time_t *timep);
struct tm *localtime(const time_t *timep);
                                POSIX.1
```

Devuelve: puntero a estructura tm

La diferencia entre estas dos funciones es que localtime convierte el tiempo de calendario a tiempo local (teniendo en cuenta la zona de tiempo local), mientras que gmtime obtiene el tiempo de calendario expresado como UTC.

A continuación se enumeran un conjunto de funciones relacionadas con el tiempo de calendario que permiten realizar diversas conversiones entre formatos y escritura de su valor en una salida. Se recomienda vivamente investigar estas funciones.

asctime, ctime, mktime - transform binary date and time to ASCII

```
#include <time.h>

char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timep);
                                POSIX.1
```

Devuelven: puntero a cadena de caracteres terminada con carácter null

```
time_t mktime(struct tm *timeptr);
                                POSIX.1
```

Devuelve: número de caracteres almacenados en el vector si hay espacio, o en otro caso.

strftime - format date and time

```
#include <time.h>
size_t strftime(char *s, size_t max, const char *format, const struct tm *tm);
                                POSIX.1
```

Devuelve: número de caracteres almacenados en el vector si hay espacio, 0 en otro caso.

El tiempo de proceso se obtiene a través de la función times que permite obtener las tres componentes anteriormente citadas: tiempo de reloj, tiempo de CPU de usuario y tiempo de CPU de sistema.

times - get process times

```
#include <sys/times.h>
clock_t times(struct tms *buf);
                                POSIX.1
```

Devuelve: tiempo de reloj en ticks de reloj si no error, -1 si error.

Esta función rellena la estructura tns apuntada por buf

```
struct  tms
        {
                clock_t tms_utime;  /* user time */
                clock_t tms_stime;  /* system time */
                clock_t tms_cutime; /* user time of children */
                clock_t tms_cstime; /* system time of children */
        };
```

La estructura no contiene ninguna medida relativa al tiempo de reloj ya que es el valor devuelto por la función cada vez que es llamada. Este valor se mide desde un punto arbitrario en el pasado, de forma que no se puede usar su valor absoluto, sino más bien su valor relativo. Por ejemplo, se llama a la función times y se memoriza el valor devuelto. En algún momento posterior se vuelve a llamar a la función times, y se resta del valor previamente obtenido el nuevo valor obtenido. La diferencia entre estos dos valores nos dará el tiempo de reloj.

Los dos últimos campos de la estructura referentes a procesos hijos contiene solo los valores para aquellos procesos hijos por los que el proceso padre está esperando su terminación.

Todos los valores de tipo clock_t devueltos por la función se convierten a segundos utilizando para ello el número de ticks de reloj por segundo. (el valor _SC_CLK_TCK devuelto por sysconf)

Ejercicio 10.

Ejecuta el siguiente programa con la siguiente línea de orden y explica los resultados obtenidos.

prac10 “sleep 5” “date”

Ejecuta el programa con la siguiente línea de orden. Explica los resultados obtenidos.

prac10 “grep main *.c > popo”

prac10.c

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

void do_cmd(char *cmd);

void pr_times(clock_t real, struct tms *tmsstar, struct tms *tmsend);

void pr_exit(int estado);

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

 for (i = 1; i < argc; i++) do_cmd(argv[i]);

 exit(0);
}

void do_cmd(char *cmd)
{
 struct tms tmsstart, tmsend;
 clock_t start, end;
 int status;

 fprintf(stderr, "\n command: %s \n", cmd);

 if ((start = times (&tmsstart)) == -1) perror("times error");

 if ((status = system(cmd)) < 0) perror("system() error");

 if ((end = times(&tmsend)) == -1) perror("times error");
```



```
 pr_times(end - start, &tmsstart, &tmsend);

 pr_exit(status);
}

void pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
 static long clktck = 0;

 if (clktck == 0)
  if ((clktck = sysconf(_SC_CLK_TCK)) < 0) perror("sysconf error");

 fprintf(stderr, " real: %7.2f \n", real / (double) clktck);

 fprintf(stderr, " user: %7.2f \n", (tmsend->tms_utime - tmsstart->tms_utime) /
(double) clktck);

 fprintf(stderr, " sys: %7.2f \n", (tmsend->tms_stime - tmsstart->tms_stime) /
(double) clktck);

 fprintf(stderr, " child user: %7.2f \n", (tmsend->tms_cutime – tmsstart->
tms_cutime) / (double) clktck);

 fprintf(stderr, " child sys: %7.2f \n", (tmsend->tms_cstime – tmsstart->
tms_cstime) / (double) clktck);
}

void pr_exit(int estado)
{
 if (WIFEXITED(estado)) printf("Terminación normal, exit status = %d\n",
WEXITSTATUS(estado));
 else if (WIFSIGNALED(estado))
 {
  printf("Terminación anormal, numero de signal = %d ",
WTERMSIG(estado));
  if (WCOREDUMP(estado)) printf("(se genero fichero core)\n");
  else printf(" \n");
 }
 else if (WIFSTOPPED(estado)) printf("Hijo detenido, numero de signal =
%d\n", WSTOPSIG(estado));
}
```

Práctica 3: Comunicación elemental entre procesos

Objetivo:

La siguiente práctica tiene como objetivo repasar y presentar los elementos fundamentales sobre comunicación de procesos en UNIX.

3. Comunicación entre procesos

En el punto 1 de las prácticas se han descrito las primitivas de control de procesos y se ha visto como crear múltiples procesos. Como se puede intuir fácilmente, con lo visto hasta la fecha, la única manera de que los procesos creados intercambien información es a través de los ficheros que permanecen abiertos a través de las llamadas fork o exec, o bien directamente a través de ficheros que se abran al efecto en el sistema de ficheros. En esta parte vamos a presentar otras técnicas de comunicación entre procesos UNIX denominadas genéricamente en terminología UNIX: IPC.

Las técnicas IPC en UNIX han sido y continúan siendo un cajón de sastre donde aparecen multitud de aproximaciones, y pocas de ellas poseen la característica de ser portables a través de diversas implementaciones. Las diferentes formas de IPC se resumen a continuación:

Tipos de IPC	SVR4	4.3+BSD
pipes (half duplex)	*	*
FIFOs (named pipes)	*	*
stream pipes (full duplex)	*	*
message queues	*	*
semaphores	*	
shared memory	*	
sockets	*	*
streams	*	

La única forma de IPC con la que se pude contar en todas las implementaciones de UNIX es la basada en pipes half-duplex. Las primeras 6 formas de IPC están restringidas a procesos que residen en la misma máquina. Las dos últimas formas son las únicas dos que están destinadas a soportar IPC entre procesos residentes en máquinas diferentes.

3.1 PIPES

Las “pipes” son mecanismos de comunicación entre procesos que permiten a dos o más procesos intercambiar información. Se utilizan muy frecuentemente dentro de los shells para conectar la salida estándar de una utilidad con la entrada estándar de otra. A continuación se muestra un ejemplo simple de una orden de shell que determina cuántos usuarios hay en el sistema:

```
$ who | wc –l
```


La utilidad who genera una línea de salida por usuario. Esta salida es “entubada” en la utilidad wc, que cuando se invoca con la opción –1, saca el número total de líneas de su entrada. Es importante señalar que ambos procesos, lector y escritor, relacionados por una pipe se ejecutan concurrentemente; una pipe automáticamente almacena la salida del escritor y suspende al escritor si la pipe llega a estar llena. De forma similar, si una pipe llega a vaciarse, el lector se suspende hasta que alguna salida adicional se encuentra disponible.

Todas las versiones de UNIX soportan “unnamed pipes”, que son la clase de pipes que usan los shell. System V también soporta una clase de pipes más potente denominada “named pipes”.

3.2 UNNAMED PIPES

Una “unnamed pipe” es un enlace de comunicación unidireccional (half-duplex) que almacena automáticamente su entrada hasta un máximo de 4 K (BSD) o 40 K (System V) y se crea mediante la llamada al sistema pipe().

pipe - create pipe

```
#include <unistd.h>
int pipe(int filedes[2]);
```

POSIX.1

Devuelve: 0 si no error, -1 si error

Cada terminación de una pipe tiene asociado un descriptor de fichero. La terminación de escritura de un pipe (filedes[1]) puede ser escrita utilizando write(), y la terminación de lectura (filedes[0]) puede ser leída utilizando read(). Cuando un proceso ha terminado de utilizar un descriptor de fichero asociado a una pipe, se debe cerrar utilizando close().

Se recomienda vivamente obtener y leer cuidadosamente, vía “man”, la información referente a la llamada al sistema pipe.

Las “unnamed pipes” se utilizan para la comunicación entre un proceso padre y su hijo, donde uno de los procesos es escritor y el otro proceso es un lector. La secuencia típica de eventos es la que sigue:

1) El proceso padre crea una “unnamed pipe” utilizando pipe()
2) El proceso padre ejecuta la llamada fork()
3) El escritor cierra su terminación de lectura del pipe, y el lector cierra su terminación de escritura pipe.
4) Los procesos se comunican utilizando write() y read().
5) Cada proceso cierra su descriptor de fichero activo cuando da por terminada la comunicación.

Nota: La comunicación bidireccional solo es posible utilizando 2 pipes.

Ejercicio 11.

Ejecuta el siguiente programa que crea un padre lector y un hijo escritor, y explica los resultados obtenidos. ¿Cuál es la finalidad de la escritura de un carácter NULL por parte del hijo? ¿Existe alguna estrategia adicional sustitutiva de la escritura de un carácter NULL?

prac11.c

```
#include <stdio.h>

#define READ 0    /* Indice de la terminación de lectura del pipe */
#define WRITE 1   /* Indice de la terminación de escritura del pipe */

char *frase = "Soy el proceso hijo y me encuentro bien";

int main()
{
 int fd[2];
 int bytesRead;

 char mensaje[100];  /* Buffer de mensajes del proceso padre */

 pipe (fd);      /* Creación de un unnamed pipe */

 if (fork() == 0)    /* El hijo escritor */
 {
  close(fd[READ]);  /* Cierro terminación del pipe no usada */
  write(fd[WRITE], frase, strlen(frase) + 1);  /* Incluye NULL */
  close(fd[WRITE]);
 }
 else  /* El padre lector */
 {
  close(fd[WRITE]);  /* Cierre terminación del pipe no usada */
  bytesRead = read(fd[READ], mensaje, 100);
  printf("El padre ha leído %d bytes: %s \n", bytesRead, mensaje);
  close(fd[READ]);
 }

 exit(0);
}
```


Cuando un proceso escritor envía más de un mensaje de longitud variable en una pipe, se debe usar un protocolo para indicar al lector el fin de mensaje. Los métodos para realizar esto incluyen:

a) Enviar la longitud del mensaje (en bytes) antes de enviar el mensaje propiamente dicho.
b) Terminar el mensaje con un carácter especial como “\n” o NULL.

Ejercicio 12.

Escribe un programa que implemente un proceso productor y un proceso consumidor con buffer intermedio de capacidad 5. El proceso productor será el proceso padre y el proceso consumidor será un proceso hijo creado por el proceso padre. La comunicación de información entre productor consumidor será a través de pipes. Los mensajes comunicados serán números naturales en secuencia empezando por el 0. La secuencia comunicada empezará en el 0 y terminará en el 100.

Determina la longitud máxima ocupada del buffer.

A continuación se presentan dos ejercicios adicionales sobre pipes que tratan de afianzar algunos de los conceptos presentados.

Ejercicio 13.

El programa presentado a continuación permite presentar los resultados de una aplicación o fichero página a página. La aplicación encamina todos los datos a una pipe que comunica directamente con los paginadores convencionales existentes en el S. O. UNIX. Para hacer esto se crea una pipe, se crea un proceso hijo, el hijo redirecciona la terminación de lectura de la pipe hacia su entrada estándar, y por fin el proceso hijo realiza un exec sobre el programa de paginación del sistema. Analiza en detalle el programa y ejecútalo. Explica los resultados obtenidos.

¿Por qué se realizan las operaciones presentadas en el programa relacionadas con la función dup2?

¿Por qué se consultan las variables de entorno?

prac13.c

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

```
#define DEF_PAGER "/usr/bin/more"    /* paginador por defecto */
#define MAXLINE 4096

int main(int argc, char *argv[])
{
  int n;
  int fd[2];

  pid_t pid;

  char line[MAXLINE];
  char *s;
  char *pager, *argv0;

  FILE *fp;

  if (argc != 2)
  {
    perror("usage: a.out <pathname>");
    exit(1);
  }

 if ((fp = fopen (argv[1], "r")) == NULL)
  {
    perror(argv[1]);
    exit(1);
  }

  if (pipe(fd) < 0) perror("Error en pipe");

  if ((pid = fork()) < 0) perror("Error en fork");

  else if (pid > 0)
  {
    close(fd[0]);

    while (fgets(line, MAXLINE, fp) != NULL)
    {
      n = strlen(line);
      if (write(fd[1], line, n) != n) perror("Error de escritura en la pipe");
    }

    if (ferror(fp)) perror("Error en fgets");

    close(fd[1]);
    if (waitpid(pid, NULL, 0) < 0) perror ("Error en waitpid");

    exit(0);
  }
```



```
  else
  {
    close(fd[1]);
    if (fd[0] != STDIN_FILENO)
    {
      if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO) perror("dup2 error to
stdin");
      close(fd[0]);
    }

    if ((pager = getenv("PAGER")) == NULL) pager = DEF_PAGER;

    if ((argv0 = strrchr(pager, '/')) != NULL) argv0++;
    else argv0 = pager;

    if (execl(pager, argv0, 0) < 0) perror(pager);
  }
}
```

Como se señaló en la práctica anterior existe una **condición de carrera** (race condition) cuando múltiples procesos están tratando de hacer algo sobre un conjunto de datos compartidos y el resultado final depende del orden en el que los procesos se ejecutan.

La función fork es una de las funciones que pueden dar lugar a condiciones de carrera si el código que aparece después de la llamada fork implícita o explícitamente depende de que el padre o el hijo se ejecuten primero. En general no se puede predecir a priori que proceso se ejecutará primero.

De cara a evitar condiciones de carrera, así como esperas activas (que consumen inútilmente CPU), se necesita un procedimiento basado en señales permitiendo a un proceso comunicar a otro su estado actual.

Ejercicio 14.

Estudia las siguientes funciones pensando en su utilización para evitar condiciones de carrera y esperas activas entre procesos padres e hijos. ¿Cómo se pueden utilizar estas funciones para evitar condiciones de carrera y esperas activas cuando hay que establecer un orden de ejecución estricto entre un proceso padre y un proceso hijo?

sincrpyh.h

```
#include <sys/wait.h>
#include <errno.h>
#include <stdio.h>
```

```
/* Declaración de las pipes */
int pfd1[2], pfd2[2];

/* Rutina de inicialización para las restantes rutinas */
void TELL_WAIT();

/* Notificar al proceso padre que el hijo ha realizado la tarea */
void TELL_PARENT(pid_t pid);

/* El hijo espera contestación del proceso padre */
void WAIT_PARENT();

/* Notificar al proceso hijo que el padre ha realizado la tarea */
void TELL_CHILD(pid_t pid);

/* El padre espera contestación del proceso hijo */
void WAIT_CHILD();
```

sincrpyh.c

```
#include "sincrpyh.h"

void TELL_WAIT()
{
  if (pipe(pfd1) < 0 || pipe(pfd2) < 0) perror("pipe error");
}

void TELL_PARENT(pid_t pid)
{
  if (write(pfd2[1], "c", 1) != 1) perror("Error de escritura");
}

void WAIT_PARENT()
{
  char c;

  if (read(pfd1[0], &c, 1) != 1) perror("Error de lectura");
  if (c != 'p')
  {
    perror("WAIT_PARENT: datos incorrectos");
    exit(1);
  }
}

void TELL_CHILD(pid_t pid)
{
  if (write(pfd1[1], "p", 1) != 1) perror("Error de escritura");
}
```



```
void WAIT_CHILD()
{
  char c;

  if(read(pfd2[0], &c, 1) != 1) perror("Error de lectura");
  if (c != 'c')
  {
    perror("WAIT_CHILD: datos incorrectos");
    exit(1);
  }
}
```

Ejercicio 15.

Modifica el programa del ejercicio 5 de forma que el padre escriba primero su mensaje completo y a continuación el hijo escriba su mensaje, utilizando las funciones diseñadas en el ejercicio 14.

Modifica el programa del ejercicio 5 de forma que el proceso hijo escriba primero su mensaje completo y a continuación el padre escriba su mensaje utilizando las funciones diseñadas en el ejercicio 14.

Ejercicio 16.

Los shells de UNIX usan unnamed pipes para construir pipelines. Utilizan un truco similar al mecanismo de redirección para conectar la salida estándar de un proceso a la entrada estándar de otro. Para ilustrar esta aproximación, se presenta el código fuente de un programa que ejecuta dos programas con nombre, conectando la salida estándar del primero a la entrada estándar del segundo. Se asume que ninguno de los programas se invoca con opciones y que los nombres de los programas se listan en la línea de orden. Si la aplicación se encuentra en el fichero ejecutable, prac16, utilizar como ejemplo de prueba: prac16 who wc

prac16.c

```
#include <stdio.h>

#define READ 0
#define WRITE 1

int main(int argc, char *argv[])
{
  int fd[2];

  pipe(fd);     /* Creación de unnamed pipe */
```

```
 if (fork() != 0)     /* Padre escritor */
 {
  close(fd[READ]);     /* Cierre terminal no usada de la pipe */
  dup2(fd[WRITE], 1);     /*Duplicar la terminación usada en stdout */
  close(fd[WRITE]);     /* Cierre terminación usada original */
  execlp(argv[1], argv[1], NULL);     /* Ejecución programa escritor */
  perror("connect");
 }

 else     /* Hijo lector */
 {
  close(fd[WRITE]);     /* Cierre terminal no usada de la pipe */
  dup2(fd[READ], 0);     /* Duplicar la terminación usada en stdin */
  close(fd[READ]);     /* Cierre terminación usada original */
  execlp(argv[2], argv[2], NULL);     /* Ejecución programa lector */
  perror("connect");
 }
}
```

3.2.1 Funciones popen y pclose

La operación de creación de una pipe para unir procesos es muy común, ya sea para leer su salida o para enviar información de entrada. Es por ello que la librería estándar de E/S históricamente ha proporcionado las funciones popen y pclose. Estas dos funciones realizan todo el trabajo sucio que se ha presentado en los ejercicios anteriores relacionados con pipes:

(1) Creación de una pipe.
(2) Crear un proceso hijo.
(3) Cerrar las terminaciones no usadas de la pipe.
(4) Cargar el nuevo programa en el espacio del hijo mediante llamadas a la función exec.
(5) Esperar para que la orden en ejecución termine.

popen, pclose - process I/O

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
                Devuelve: puntero a fichero si no error, NULL si error.

int pclose(FILE *stream);
                Devuelve: status de terminación de  command, -1 si error
```

La función popen realiza un fork y un exec para ejecutar el programa especificado mediante command, y devuelve un puntero de fichero de E/S estándar. Si el tipo es “r”, el puntero a fichero está conectado a la salida estándar de la aplicación especificada mediante command. Si el tipo es “w”, el puntero de fichero está conectado a la entrada estándar de la aplicación especificada mediante command.

La función pclose cierra el canal de comunicación, espera que la aplicación especificada mediante command termine, y devuelve el status de terminación del shell. Esto es también lo que hace la función system estudiada en la práctica anterior. Si el shell no puede ser ejecutado, el status de terminación devuelto por pclose es equivalente a la ejecución por el shell de exit(127).

Ejercicio 17.

Reescribe el programa del ejercicio 13 que permite presentar resultados página a página pero utilizando las funciones popen y pclose.

3.2.2 Coprocesos y filtros UNIX

Ejercicio 18.

Escribe una aplicación que escriba un prompt en la salida estándar y lea una línea de la entrada estándar. El prompt lo escribe el proceso padre y la entrada la recibe un proceso hijo creado mediante popen y que ejecutará una aplicación que actuará como filtro. La aplicación filtro obtiene la entrada y convierte toda letra mayúscula en letra minúscula y lo envía a través de la pipe creado mediante popen al proceso padre.

Ayuda para crear la aplicación filtro:
Utiliza las siguientes funciones de librería estándar de C:
int getchar(); int putchar(int c);
int tolower(int c); int isupper(int c)

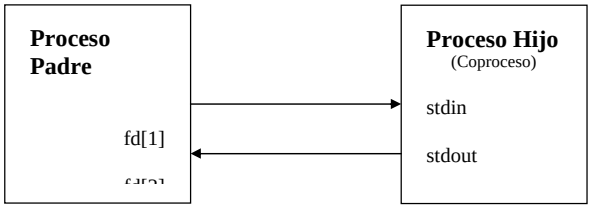
Un filtro UNIX es un programa que lee de la entrada estándar y escribe en la salida estándar. Los filtros normalmente se conectan linealmente en pipelines de shell. Un filtro llega a ser un coproceso cuando el mismo programa genera su entrada y lee su salida.

La función popen suministra una pipe unidireccional que conecta con otro proceso, uno a su entrada estándar y otro desde su salida estándar. El objetivo es escribir en la entrada estándar del coproceso, que el coproceso opere con los datos suministrados, y por fin los resultados aparecen en la salida estándar para que el proceso suministrador de los datos lea los resultados.

El siguiente ejercicio ilustra el principio de funcionamiento de los coprocesos y filtros de UNIX.

Ejercicio 19.

Un proceso crea dos pipes: una es la entrada estándar del coproceso y la otra es la salida estándar del coproceso.



A continuación se muestra el código del programa filtro que se convertirá en un coproceso. Este programa lee dos números de su entra estándar, los suma y deposita el resultado en su salida estándar. Compila este programa y nombra el ejecutable como "suma2".

suma2.c

```
/* Código del filtro que actúa como coproceso */
/* Lee dos enteros de la entrada estándar y devuelve su suma por la salida
estándar */

#include <stdio.h>
#include <unistd.h>

#define MAXLINE 4096          /* max line lengh */

int main()
{
  int n, int1, int2;
  char line[MAXLINE];

  while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0)
  {
    line[n] = 0;
    if (sscanf(line, "%d %d", &int1, &int2) == 2)
    {
      sprintf(line, "%d \n", int1 + int2);
      n = strlen(line);

      if (write(STDOUT_FILENO, line, n) != n) perror("Error de escritura");
    }
```



```
    else
    {
      if (write(STDOUT_FILENO, "Args. no válidos \n", 18) != 18) perror("Error
de escritura");
    }
  }

  exit(0);
}
```

A continuación se muestra el código del proceso padre que invoca al coproceso "suma2", después de haber leído dos números de su entrada estándar. El valor devuelto por el coproceso es escrito en la salida estándar.

prac19.c

```
/* Código del proceso padre */

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

#define MAXLINE 4096          /* max line lengh */

static void (sig_pipe(int signo));

int main()
{
  int n;
  int fd1[2], fd2[2];

  pid_t pid;

  char line[MAXLINE];

  if (signal(SIGPIPE, sig_pipe) == SIG_ERR) perror("signal error");

  if (pipe(fd1) < 0 || pipe(fd2) < 0) perror("pipe error");

  if ((pid = fork()) < 0) perror("fork error");

  else if (pid > 0)
  {
    close(fd1[0]);
    close(fd2[1]);
```

```
   while(fgets(line, MAXLINE, stdin) != NULL)
   {
    n = strlen(line);
    if (write(fd1[1], line, n) != n) perror("write error to pipe");

    if ((n = read(fd2[0], line, MAXLINE)) < 0) perror("read error from pipe");

    if (n == 0)
    {
        perror("chid close pipe");
      break;
    }

    line[n] = 0;
    if (fputs(line, stdout) == EOF) perror("fputs error");
   }

   if (ferror(stdin)) perror("fgets error on stdin");
   exit(0);
  }
  else
  {
   close(fd1[1]);
   close(fd2[0]);

   if (fd1[0] != STDIN_FILENO)
   {
    if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO) perror("dup2 error
to stdin");
    close(fd1[0]);
   }

   if (fd2[1] != STDOUT_FILENO)
   {
    if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO) perror("dup2
error to stdout");
    close(fd2[1]);
   }

   if (execl("./suma2", "suma2", 0) < 0) perror("execl error");
  }
}

static void (sig_pipe(int signo))
{
   printf("SIGPIPE caught \n");
   exit(1);
}
```


Prueba los programas anteriores y comenta los resultados. ¿Qué ocurre si se mata al coproceso antes de que escriba la suma en su salida estándar?.

¿Qué ocurre si se ejecuta el programa “suma2” directamente?

¿Funciona como se esperaba?

Ejercicio 20.

Sustituye el código del programa filtro del ejercicio anterior por el programa presentado a continuación. La única diferencia entre ambos radica en que se han sustituido las funciones read y write por funciones estándar de entrada salida. ¿Qué ocurre en este caso?.

Explica las diferencias observadas de comportamiento. ¿Se puede corregir el problema observado?.

pract20.c

```
#include <stdio.h>

#define MAXLINE 4096  /* max line lengh */

int main()
{
  int int1, int2;

  char line[MAXLINE];

  while (fgets(line, MAXLINE, stdin) != NULL)
  {
    if (sscanf(line, "%d %d", &int1, &int2) == 2)
    {
      if (printf("%d \n", int1 + int2) == EOF) perror("printf error");
    }
    else
    {
      if (printf("Args. Inválidos \n") == EOF) perror("printf error");
    }
  }

  exit(0);
}
```

El problema se puede solucionar si se añaden las siguientes líneas de código al programa anterior justo antes de entrar en el bucle.

```
if (setvbuf(stdin, NULL, _IOLBF, 0) != 0) perror("setvbuf error");
if (setvbuf(stdout, NULL, _IOLBF, 0) != 0) perror("setvbuf error");
```

Este código obliga a fgets a regresar cuando una línea se encuentra disponible, y obliga a printf a realizar un fflush cuando una nueva línea alcanza la salida.

Consulta en el man dicha función estándar de C

setbuf, setvbuf - assign buffering to a stream

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

3.3 NAMED PIPES

"Named pipes", a veces denominadas FIFOs, es un mecanismo de comunicación entre procesos menos restrictivo que las unnamed pipes, y ofrecen las siguientes ventajas:

1) Tienen un nombre que existe en el sistema de ficheros.
2) Pueden ser usadas por procesos no relacionados
3) Existen hasta que son explícitamente borradas.

Todas las reglas que se han mencionado para unnamed pipes se aplican a las named pipes, excepto que su capacidad es de alrededor de 40 K.

Named pipes existen como ficheros especiales en el sistema de ficheros, y se pueden crear de alguna de las siguientes formas:

1) Utilizando la utilidad UNIX mknod.
2) Utilizando la llamada al sistema mknod( ).
3) Utilizando la utilidad UNIX mkfifo.
4) Utilizando la función de C mkfifo( ).

Para crear una named pipe con la utilidad mknod hay que usar la opción p. El modo de una named pipe se puede poner a través de chmod, permitiendo a otros acceder a la pipe que cree un usuario. Observar que cuando se crea una named pipe el tipo que aparece mediante ls es p.

```
$ mknod miPipa p
$ chmod ug+rw miPipa
$ ls –l miPipa
```


Para crear una named pipe utilizando mknod(), hay que especificar S_IFIFO como modo del fichero. El modo de la pipe puede ser cambiado con chmod().

```
mknod ("miPipa", S_IFIFO, 0);
chmod("miPipa", 0660);
```

Independientemente de cómo sea creada la named pipe, el resultado final es el mismo: un fichero especial se añade al sistema de ficheros. Una vez que una pipe se ha abierto utilizando open(), write() añade datos al final de la cola FIFO, y read() elimina datos del principio de la cola FIFO. Cuando un proceso ha acabado de usar una named pipe, debe cerrarlo usando close(), y cuando ya no se va a usar más debe ser eliminado del sistema de ficheros usando unlink().

De forma parecida a las unnamed pipes, una named pipe es unidireccional. Los procesos lectores deben abrir la pipe para sólo lectura y los procesos escritores deben abrir la pipe para sólo escritura.

Reglas especiales:

a) si un proceso trata de abrir una named pipe para sólo lectura y no existe un proceso que la haya abierto para escritura, el lector esperará hasta que un proceso abra para escribir a menos que se señale la opción O_NDELAY en cuyo caso open() tiene éxito inmediatamente.

b) Si un proceso trata de abrir una named pipe para sólo escritura y no existe un proceso que la haya abierto para lectura, el escritor esperará hasta que un proceso la abra para lectura, a menos que se señale la opción O_NDELAY en cuyo caso open() falla inmediatamente.

Nota: Las named pipes no se pueden usar en red.

Ejercicio 21.

A continuación se presentan dos programas que hacen uso de un named pipe. El primero da lugar a un proceso lector que extrae datos de la named pipe, así mismo se encarga de la creación de la pipe en caso de que esta no exista. El segundo programa da lugar a un proceso escritor que escribe datos en la named pipe. Explica el principio de funcionamiento de estos dos programas. ¿Se puede construir una aplicación que permita comunicar a dos usuarios en dos cuentas diferentes mediante la named pipe?

¿Qué hay que modificar en estos programas?

prac21_l.c

```
/* Proceso lector de named pipe denominado "miPipa" */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int readLine(int fdaux, char *straux);

int main()
{
  int fd;
  char str[100];

  unlink("miPipa");
  mknod("miPipa", S_IFIFO, 0);
  chmod("miPipa", 0770);

  fd = open ("miPipa", O_RDONLY);

  while (readLine(fd, str) > 0) printf("%s\n", str);
  close(fd);
}

int readLine(int fdaux, char *straux)
{
  int n;

  do
  {
    n = read(fdaux, straux, 1);
  } while (n > 0 && *straux++ != NULL);

  return (n);
}
```

prac21_e.c

```
/* Proceso escritor de named pipe denominado "miPipa" */

#include <stdio.h>
#include <fcntl.h>

int main()
{
  int fd, messageLen, i;
```



```
  char message[100];

  sprintf(message, "Hola de PID %d", getpid());
  messageLen = strlen(message) + 1;

  do
  {
    fd = open ("miPipa", O_WRONLY);
    if (fd == -1)
    {
      printf("\n Error, la FIFO no existe.\n");
      sleep(1);
    }
  } while (fd == -1);

  printf("\n OK, la FIFO ya existe.\n Enviando mensajes ...\n");

  for (i = 1; i <= 3; i++)
  {
    write(fd, message, messageLen);
    sleep(3);
  }

  close(fd);
}
```

Como se ha señalado una FIFO es un tipo de fichero. Uno de los códigos de la componente st_mode de la estructura stat sirve para indicar que un fichero se trata de una FIFO. Esto se puede determinar a través de la macro S_ISFIFO.

La creación de una FIFO también se puede realizar a través de la siguiente función de C.

mkfifo - make a FIFO special file (a named pipe)

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

int mkfifo ( const char *pathname, mode_t mode );
```

Devuelve: 0 si no error, -1 si error.

La especificación del argumento mode para mkfifo es la misma que para la función open. Una vez creada la FIFO se pueden utilizar las funciones convencionales para tratar con ficheros.

Práctica 4: Señales.

Objetivo:

La siguiente práctica tiene como objetivo presentar los elementos fundamentales relacionados con las señales en UNIX.

4. Señales en UNIX

Las señales en UNIX (signals) son interrupciones software. La mayor parte de los programas de aplicación, no triviales, necesitan tratar con algún aspecto relacionado con señales. Las señales, en general, proporcionan un método para manejar eventos asíncronos: un usuario que pulsa en un terminal las teclas adecuadas para parar un proceso, por ejemplo.

4.1 Conceptos relacionados con señales

En primer lugar hay que decir que toda señal tiene un nombre. Estos nombres empiezan con las letras: SIG. Por ejemplo, SIGABRT es la señal para abortar que se genera cuando un proceso llama a la función abort. SIGALRM es la señal de alarma que se genera cuando un timer inicializado mediante la función alarm alcanza su consigna. SVR4 y 4.3+BSD tienen 31 señales diferentes.

Estos nombres se definen mediante constantes enteras positivas (el número de la señal) en el fichero <signal.h>. Ninguna señal tiene asociado el número de señal 0 que esta reservado para un uso especial de la función kill.

Existen numerosas condiciones que dan lugar a la generación de una señal.

a) Las señales generadas por el terminal ocurren cuando los usuarios pulsan determinadas teclas (o combinaciones de las mismas). Por ejemplo, la pulsación de la tecla DELETE de la terminal provoca, normalmente, la generación de la señal de interrupción (SIGINT o SIGQUIT).

b) Las excepciones hardware generan señales: división por cero, referencia no válida a memoria, etc. Estas condiciones son detectadas normalmente por el hardware, y el kernel es informado. El kernel entonces genera la señal apropiada para el proceso que se encontraba en ejecución en el momento que se produjo la condición. Por ejemplo, SIGSEGV se genera hacia un proceso que trató de realizar una referencia a memoria no válida.

c) La llamada al sistema kill(2) permite a un proceso enviar una señal a otro proceso o grupo de procesos. Naturalmente existen ciertas limitaciones, se debe ser el propietario del proceso al que se manda la señal o tener privilegios de superusuario.

man –s 2 kill

kill - send a signal to a process or a group of processes

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
                POSIX.1
      Devuelven: -1 si error, 0 si no error
```

d) El comando kill(1) permite enviar señales a otros procesos. Este programa no es más que una interfaz de la función kill. Esta orden se utiliza a veces para terminar un proceso que se está ejecutando en "background".

kill - terminate a process

```
kill [ -s signal | -p ]  [ -a ] pid ...
kill -l [ signal ]
```

e) Condiciones software pueden generar señales, principalmente cuando sucede algo ante lo que un proceso debe reaccionar. Ejemplos de estas señales son: SIGPIPE (generada cuando un proceso escribe en una pipe después de que el proceso lector ha terminado) o SIGALRM (generada cuando un reloj alarma inicializado por un proceso se dispara).

Las señales son ejemplos clásicos de eventos asíncronos. Estas ocurren en lo que a la vista de un proceso puede parecer momentos de tiempo seleccionados aleatoriamente. El proceso no puede analizar una variable (como por ejemplo errno) para ver si una señal ha ocurrido o no, en lugar de esto el proceso debe instruir al kernel con sentencias tales como: "Si la señal x ocurre, entonces hacer lo siguiente...".

Existen tres formas diferentes de reaccionar el kernel ante la aparición de una señal de acuerdo a como se le haya instruido. Estas distintas formas se denominan dentro del argot de UNIX disposición de la señal o acción asociada con una señal.

Disposicón ante una señal:

1. Ignorar la señal. Esta es la forma normal de reaccionar ante la mayor parte de las señales, pero existen dos señales que nunca pueden ser ignoradas: SIGKILL Y SIGSTOP. La razón por la que estas dos señales no pueden ser ignoradas es que de esta manera se proporciona al superusuario un método seguro para terminar un proceso o pararlo. Por otra parte, conviene no olvidar que si se ignoran algunas señales que pueden ser generadas por excepciones hardware el comportamiento del proceso no está definido.

2. Capturar la señal. Para reaccionar de esta manera, debemos instruir al kernel para que ejecute una determinada función, que el usuario habrá diseñado, cada vez que la señal en cuestión suceda. La función de usuario puede realizar todo lo que se considere oportuno para manejar la condición.

3. Dejar que se aplique la acción por defecto a la ocurrencia de una señal. Toda señal tiene asociada una acción por defecto (en la mayor parte de los casos, la acción por defecto consiste en terminar un proceso).

A continuación se presenta una tabla que contiene las señales presentes en SVR4 y 4.3+BSD. Cuando en la tabla aparece "terminate w/core" significa que una imagen de la memoria del proceso se coloca en el fichero denominado core que se ubica en el directorio de trabajo del proceso. Este fichero core puede ser utilizado por la mayor parte de los debuggers para examinar el estado del proceso en el momento en que terminó.

Señales en UNIX

Nombre	Descripción	SV	BSD	Acción por defecto
SIGABRT	Terminación anormal(abort)	*	*	Terminate w/core
SIGALRM	Time-out (alarm)	*	*	Terminate
SIGBUS	Hardware fault	*	*	Terminate w/core
SIGCHLD	Cambio de estado del hijo	*	*	Ignore
SIGCONT	Proceso continua parado	*	*	Continue/ignore
SIGEMT	Hardware fault	*	*	Terminate w/core
SIGFPE	Excepción aritmética	*	*	Terminate w/core
SIGHUP	Hangup	*	*	Terminate
SIGILL	Instrucción máquina ilegal	*	*	Terminate w/core
SIGINFO	Petición de estado desde teclado		*	Ignore
SIGINT	Carácter de interrupción desde terminal	*	*	Terminate
SIGIO	E/S asíncrona	*	*	Terminate/ignore
SIGIOT	Hardware fault	*	*	Terminate w/core
SIGKILL	Terminación	*	*	Terminate
SIGPIPE	Escritura en pipe sin lectores	*	*	Terminate
SIGPOLL	Pollabel even (poll)	*		Terminate
SIGPROF	Porfiling time alarm (setitimer)	*	*	Terminate
SIGPWR	Fallo/reinicio de alimentación	*		Ignore
SIGQUIT	Carácter quiet desde terminal	*	*	Terminate w/core
SIGSEGV	Referencia a memoria no válida	*	*	Terminate w/core
SIGSTOP	Stop	*	*	Stop process
SIGSYS	Llamada al sistema no válida	*	*	Terminate w/core
SIGTERM	Terminación	*	*	Terminate
SIGTRAP	Hardware fault	*	*	Terminate w/core
SIGTSTP	Carácter stop desde el terminal	*	*	Stop process
SIGTTIN	Lectura en background desde la tty	*	*	Stop process
SIGTTOU	Escritura en background a la tty	*	*	Stop process
SIGURG	Condición urgente	*	*	Ignore
SIGUSR1	Señal definida por el usuario	*	*	Terminate
SIGUSR2	Señal definida por el usuario	*	*	Terminate
SIGWTALRM	Alarma de tiempo virtual (setitimer)	*	*	Terminate
SIGWINCH	Cambio tamaño de ventana de terminal	*	*	Ignore
SIGXCPU	Excedido límite de CPU (setrlimit)	*	*	Terminate w/core
SIGXFSZ	Excedido límite tamaño fichero (setrlimit)	*	*	Terminate w/core

4.2 Función signal

La interfaz más simple con las cuestiones relacionadas con señales en UNIX la proporciona la función signal.

signal - ANSI C signal handling

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

El argumento signo es el nombre de la señal tal y como se encuentra la primera columna de la tabla anterior.

El valor del parámetro func es:

a) la constante SIG_ING
b) la constante SIG_DFL
c) la dirección de la función que se ejecutará en el momento en el que la señal ocurra.

Si se especifica SIG_ING se está instruyendo al kernel para que ignore la señal (recuérdese que las señales SIGKILL y SIGSTOP no pueden ser ignoradas). Si se especifica SIG_DFL se está instruyendo al kernel para que ante la ocurrencia de la señal reaccione realizando la acción por defecto asociada con la señal. Cuando se especifica la dirección de una función que será llamada cuando la señal ocurra, se dice que el kernel ha sido instruido para "capturar" la señal. En el argot de UNIX se denomina a esta función "signal handler" 0 "signal-catching function".

Ejercicio 22.

El siguiente programa muestra un "signal handler" que permite capturar cualquiera de las dos señales definidas por el usuario. Estudiar específicamente las distintas declaraciones relacionadas con los parámetros de la función signal. Estudiar la definición de SIG_ERR. Ejecutar el programa en "background" y estudiar la orden kill(1) desde la línea de orden para enviar señales al proceso que tiene asociado el programa mostrado a continuación.

¿Qué sucede si se envía la señal SIGTERM al proceso? ¿Por qué? ¿Qué misión tiene la llamada a la función pause?

prac22.c

```
#include <unistd.h>
#include <signal.h>   /* Requerida por pause */

static void sig_usr(int signo);/* one handler for both signal */
```

```
int main()
{
 if (signal(SIGUSR1, sig_usr) == SIG_ERR)
 {
   perror("can't catch SIGUSR1");
   exit(1);
 }
 if (signal (SIGUSR2, sig_usr) == SIG_ERR)
 {
   perror("can´t catch SIGUSR2");
   exit(1);
 }

 while (1) pause();

}


static void sig_usr(int signo)
{

 if (signo == SIGUSR1) printf("received SIGUSR1 \n");
 else if (signo == SIGUSR2) printf("received SIGUSR2 \n");
 else
 {
   printf("recieved signal %d \n", signo);
   exit(1);
 }
 return;
}
```

Cuando un programa es cargado en memoria y lanzada su ejecución a través de una llamada a una función de tipo exec, el estado de todas las señales es o bien el estado de realizar la acción por defecto o el estado de ignorar su ocurrencia. Normalmente, todas las señales se encuentran en el estado de reaccionar con la acción por defecto. De una manera más específica, la llamada a la función exec cambia la disposición de todas las señales que estaban siendo rastreadas (capturadas) al estado de reaccionar con la acción por defecto, el resto de las señales quedan como estaban.

Una limitación que tiene la función signal se refiere a que no se puede determinar la disposición actual de una señal sin cambiar su disposición. Un poco más adelante veremos como la función sigaction permite determinar la disposición de una señal sin cambiarla.

Cuando un proceso llama a la función fork, el proceso hijo hereda las disposiciones de las señales del proceso padre. Aquí, a diferencia de la función exec, el proceso hijo arranca con una copia del proceso padre por lo que las direcciones de los "signal handlers" tienen sentido (están definidas) en el contexto del proceso hijo.

4.3 "Signal handlers" y funciones reentrantes

Ante la ocurrencia de una señal, es un proceso el que maneja esta señal cuando se ha instruido al kernel para ejecutar una función ("signal handler") cuando ocurra la señal. Este manejo de la señal por parte del proceso se traduce en que la secuencia normal de instrucciones que estaba ejecutando el proceso se interrumpe temporalmente para ejecutar la función que representa al "signal handler". Si el "signal handler" regresa mediante un return, entonces se recupera la ejecución de la secuencia normal de instrucciones del proceso en el punto donde se dejo antes de la ejecución del "signal handler". Este comportamiento es similar a la secuencia de actividades que se desencadenan cuando se atienden las interrupciones hardware.

Uno de los problemas fundamentales que aparece durante la ejecución del "signal handler" es que se puede llamar a funciones que se estaban ejecutando (y han sido interrumpidas para ejecutar el "signal handler") en un proceso. Para que estas llamadas dentro del "signal handler" no produzcan efectos perniciosos (en particular, resultados impredecibles de la llamada a la función interrumpida) se requiere que las funciones sean reentrantes.

Además de las consideraciones anteriores sobre la conveniencia que desde un "signal handler" tan sólo se llame a funciones que sean reentrantes es necesario recordar que en un proceso existen algunas variables, como por ejemplo errno, que pueden ser modificadas por las funciones llamadas desde el "signal handler" falseando el comportamiento del proceso que en principio no debería verse afectado por la ejecución del "signal handler". Por lo tanto, como regla general, cuando se llame a una función desde un "signal handler", previamente se deberá salvaguardar el valor de errno para a la vuelta de la función restaurarlo.

Funciones reentrantes que pueden ser llamadas en un "singal handler"

_exit	fork	pipe	stat
abort	fstat	read	sysconf
access	fetegid	rename	tcdrain
alarm	geteuid	rmdir	tcflow
cfgetspeed	getgid	setgid	tcflush
cfgetospeed	getgroups	setpgid	tcgetarrt
cfsetispeed	getpgrp	setsid	tcgetpgrp
cfsetospeed	getpid	setuid	tcsendbreak
chdir	getppid	sigaction	tcsetarrt
chmod	getuid	sigaddset	tcsegpgrp
chwon	kill	sigdelset	time
close	link	sigemptyset	times
creat	longjmp	sigfillset	umask
dup	lseek	sigismember	uname
dup2	mkdir	signal	unlink
execle	mkfifo	sigpending	utime
execve	open	sigprocmask	wait
exit	pathconf	sigsuspend	waitpid
fcntl	pause	sleep	write

Ejercicio 23.

El siguiente programa muestra un "signal handler" que llama a una función que no es reentrante (getpwnam). Estudia los posibles resultados que se pueden producir.

prac23.c

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

static void my_alarm();

int main()
{
 struct passwd *ptr;

 signal(SIGALRM, my_alarm);
 alarm(1);

 while (1)
 {
  if ((ptr = getpwnam("luispa")) == NULL)
  {
    perror("getpwnam error");
    exit(1);
  }

  if (strcmp(ptr->pw_name, "luispa") != 0)
    printf("return value corrupted !, pw_name = %s \n", ptr->pw_name);
 }
}


static void my_alarm()
{
 struct passwd *rootptr;

 printf("in signal handler \n");
 if ((rootptr = getpwnam("root")) == NULL)
 {
  perror("gerpwnam(root) error");
  exit(1);
 }

 alarm(1);
 return;
}
```


4.4 Señales SIGCLD Y SIGCHLD

Estas dos señales habitualmente generan cierta confusión. SIGCLD es el nombre en System V y es una señal que tiene una semántica diferente a la señal en BSD denominada SIGCHLD

La semántica de la señal BSD SIGCHLD es la normal y muy similar a la de otras muchas señales. Cuando esta señal ocurre, el estado de un proceso hijo ha cambiado y es necesario llamar a una función wait para determinar que es lo que ha sucedido.

La señal System V SIGCLD tiene un funcionamiento diferente al resto de las señales que pueden existir en el sistema. Estas diferencias se concretan en los siguientes puntos:

a) Si el proceso coloca específicamente la disposición de la señal a valor SIG_IGN, el proceso hijo del proceso llamador no generará procesos zombies. Obsérvese que esto es diferente a lo que se realiza en la acción por defecto (SIG_DFL) acometida cuando la señal ocurre, que es ignorarla. Si el proceso llamador posteriormente llama a una de las funciones wait, se bloqueará hasta que sus hijos hayan terminado, y entonces la función wait devolverá el valor -1 con errno igual a ECHILD.

b) Si el proceso coloca específicamente la disposición de la señal para su captura, el kernel inmediatamente analiza si existe algún proceso hijo listo para que se le pueda aplicar alguna función wait y si esto es así, entonces llama al "handler" de la señal SIGCLD. Esto provoca que haya que escribir con especial cuidado el "handler" para esta señal.

Ejercicio 24.

El siguiente programa muestra un "signal handler" para la señal SIGCHLD. ¿Por qué se vuelve a llamar a la función signal al entrar al handler?

prac24.c

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

static void sig_cld();

int main()
{
  pid_t pid;

  if (signal(SIGCHLD, sig_cld) == -1) perror ("signal error");

  if ((pid = fork()) < 0) perror("fork error");
  else if (pid == 0)
  {
    sleep(2);
    _exit(0);
  }

  pause();
  exit(0);
}

static void sig_cld()
{
  pid_t pid;
  int status;

  printf("SIGCHLD received \n");
  if (signal(SIGCHLD, sig_cld) == -1) perror("signal error");

  if ((pid = wait(&status)) < 0) perror("wait error");

  printf("pid = %d \n", pid);
  return;
}
```


Una señal se dice que se ha generado para un proceso (o que se ha enviado a un proceso) cuando el evento que causa la señal ha ocurrido. El evento puede ser una excepción hardware (e.g. una división por 0), una condición software (e.g. la expiración del temporizador asociado a una alarma), una señal generada por un terminal, o una llamada a la función kill. Cuando una señal se ha generado, el kernel normalmente pone a uno un flag dentro de la entrada en la tabla de procesos correspondiente al proceso o procesos a los que va dirigida la señal.

Una señal se dice que ha sido liberada a un proceso cuando la acción asociada a la señal ha comenzado a realizarse. Durante el tiempo que transcurre entre la generación de una señal y su liberación se dice que la señal está pendiente.

Un proceso tiene la opción de bloquear la liberación de una señal. Si una señal que se encuentra bloqueada es generada para un proceso, y si la acción a realizar para esta señal es la acción por defecto o capturar la señal, entonces la señal permanece en estado pendiente para el proceso hasta que el proceso o bien desbloquea la señal o bien cambia la acción a considerar con esta señal a valor ignorar. El sistema determina lo que hay que hacer con una señal bloqueada en el momento de ser librada no en el momento de ser generada. Esto permite al proceso cambiar la acción asociada a la señal antes de su liberación. La función sigpending se llama desde un proceso para determinar que señales bloqueadas y pendientes existen.

¿Qué sucede si una señal bloqueada es generada más de una vez antes de que el proceso desbloquee la señal?.

POSIX.1 permite al sistema liberar la señal una o más veces. Si el sistema libera la señal más de una vez, se dice que las señales son encoladas en una cola. No obstante, la mayor parte de los sistemas UNIX no encolan las señales, sino que el kernel libera exactamente cada señal una vez.

¿Qué sucede si más de una señal se encuentra lista para ser liberada a un proceso?.

POSIX.1 no especifica el orden en el que las señales serán liberadas al proceso.

Cada proceso tiene una máscara de señales que define el conjunto de señales que actualmente se encuentran bloqueadas para su liberación a un proceso. Esta máscara se puede ver como un valor de bits cada uno asociado con una señal diferente. Un proceso puede examinar y cambiar su máscara de señales llamando a la función sigprocmask.

Dado que es posible que el número de señales exceda el número de bits presente en un entero, POSIX.1 define un nuevo tipo de datos, sigset_t que permite manejar un conjunto de señales.

4.5 Funciones kill y raise

La función kill envía una señal a un proceso o conjunto de procesos. La función raise permite que un proceso se envía una señal a sí mismo.

kill - send a signal to a process or a group of processes

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
        POSIX.1
    Devuelve: -1 si error, 0 si no error
```

raise - send a signal to the current process

```
#include <signal.h>
int raise (int sig);
        POSIX.1
    Devuelve: -1 si error, 0 si no error
```

The raise function sends a signal to the current process. It is equivalent to kill(getpid(), sig)

Existen cuatro condiciones diferentes para el argumento pid de la función kill.

pid > 0 La señal es enviada al proceso cuyo PID es pid.

pid == 0 La señal es enviada a todos los procesos cuyo PID de grupo es igual al del proceso generador de la señal y para los que el emisor tiene permiso para enviar.

pid < -1 La señal es enviada a todos aquellos proceso cuyo PID de grupo coincide con el valor absoluto de pid y para los que el emisor tiene permiso para enviar la señal.

pid = -1 POSIX.1 deja sin especificar esta condición. SVR4 y 4.3+BSD usan esta condición para lo que llaman "broadcast signal".

Como se ha señalado anteriormente un proceso debe tener permiso para poder enviar una señal a otro proceso. El superusuario puede enviar señales a cualquier otro proceso. En cualquier otro caso, la regla básica es que el identificador de usuario real o efectivo del receptor.

POSIX.1 define la señal número 0 como la señal nula. Si el argumento signo de la función kill es 0, entonces la función realiza los análisis de errores pertinentes pero no envía ninguna señal. Esto se usa a veces para determinar si un proceso determinado existe todavía. Si se envía una señal nula a un proceso que no existe, la función kill devuelve el valor -1 y la variable errno es igual a ESRCH.

Si la llamada a la función kill provoca la generación de una señal para el proceso llamador y si la señal no se encuentra bloqueada, entonces o bien signo o bien alguna otra señal no bloqueada pendiente será liberada al proceso antes de regresar de la función kill.

4.6 Funciones alarm y pause

La función alarm permite inicializar un temporizador que expirará en un tiempo futuro. Cuando el temporizador expira, se genera la señal SIGALRM. Si se ignora o no se captura esta señal, la acción por defecto asociada a la ocurrencia de esta señal es la terminación del proceso.

alarm - set a process alarm clock

```
#include <unistd.h>
unsigned int alarm(unsigned int segundos);
        POSIX.1
```

Devuelve: 0 o el número de segundos hasta una alarma colocada previamente.

El valor segundos se refiere al número de segundos en el futuro cuando se debe generar la señal. Cuando este tiempo se alcanza, el kernel genera la señal, pero puede ocurrir que transcurra una cierta cantidad de tiempo adicional hasta que el proceso toma el control para manejar la señal, debido a los retrasos inherentes al scheduling del procesador.

Existe un único reloj alarma por proceso. Si, cuando se llama a la función alarm, existe un valor registrado para generar la señal SIGALRM que todavía no ha expirado (debido a una llamada previa a la función alarm), la función devuelve el número de segundos que quedan hasta la expiración de esta alarma previa. El valor previamente registrado para la alarma es sustituido por el nuevo valor suministrado en la actual llamada a la función.

Si existe un reloj alarma registrado previamente para el proceso que no ha expirado todavía y si el parámetro segundos de la llamada actual a la función alarm tiene el valor 0, se cancela el reloj alarma previamente registrado. No obstante, la función devuelve todavía el número de segundos que quedan hasta la expiración del reloj alarma previamente registrado.

Aunque la acción por defecto asociada a la ocurrencia de la señal SIGALRM es terminar el proceso, la mayor parte de los procesos que usan un reloj alarma capturan esta señal. Si el proceso quiere terminar entonces puede realizar el conjunto de operaciones que le permiten terminar ordenadamente.

La función pause suspende al proceso llamador hasta que se captura una señal.

pause - suspend process until signal

```
#include <unistd.h>
int pause(void);
```

POSIX.1

Devuelve: -1 con errno igual a EINTR

La única ocasión en la que la función pause regresa es cuando se ejecuta un "signal handler" y éste regresa. En este caso, pause devuelve -1 con error a valor EINTR.

Ejercicio 25.

El siguiente programa muestra una implementación de la función sleep utilizando las funciones pause y alarm. A la hora de llamar a esta función, determinar:

a) ¿Qué ocurre si el llamador de la función tiene un reloj alarma previamente registrado? ¿Cómo se puede evitar el problema?
b) ¿Qué ocurre si con la llamada a la función signal hemos cambiado la disposición de la señal SIGALRM? ¿Cómo podemos evitar este problema?
c) Si se pueden dar condiciones de carrera entre la primera llamada a alarm y la llamada a pause. Justificalo con un ejemplo.

prac25.c

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

static void sig_alrm();
unsigned int sleep2(unsigned int nsecs);

int main()
{
 printf("Acabo de empezar y me voy a dormir \n");
 sleep2(10);
 printf("Me acabo de despertar \n");

 exit(0);
}

static void sig_alrm()
{
 printf("Estoy en el signal handler\n");
 return;
}
```



```
unsigned int sleep2(unsigned int nsecs)
{
 printf("Estoy en sleep2 \n");

 if(signal (SIGALRM, sig_alrm) == SIG_ERR) return(nsecs);

 alarm(nsecs);/* start the timer */
 pause();        /* next caught signal wakes us up */

 printf("Salgo de sleep2 \n");
 return(alarm(0));      /* turn off timer, return unslept time */
}
```

Un uso bastante habitual de la función alarm, adicional a la implementación de la función sleep, se refiere a la fijación de límites superiores de tiempo a operaciones que suponen el bloqueo del proceso. El programa presentado a continuación es un ejemplo de esto, donde se realiza una operación de lectura sobre un dispositivo que puede bloquear. El programa lee una línea de la entrada estándar y la escribe por la salida estándar.

Ejercicio 26.

Analiza el código del ejemplo y determina los problemas que pueden aparecer: ¿Existen condiciones de carrera entre la primera llamada a alarm y la llamada a read? ¿Siempre aborta la función read cuando se supera la cota de tiempo impuesta en la operación?

prac26.c

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

#define MAXLINE 4096

static void sig_alrm();

int main()
{
 int n;
 char line[MAXLINE];
```

```
  if (signal(SIGALRM, sig_alrm) == SIG_ERR)
  {
    perror("signal(SIGALRM) error");
    exit(1);
  }

  alarm(10);

  if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
  {
    perror("read error");
    exit(1);
  }

  alarm(0);
  write(STDOUT_FILENO, line, n);
  exit(1);
}

static void sig_alrm()
{
  printf("\n Estoy en el hadler\n");
  return;
}
```


Área de Arquitectura y Tecnología de Computadores
EUPT: Sistemas Operativos II

Práctica 5: Sincronización y semáforos

Objetivo:

La siguiente práctica tiene como objetivo presentar los elementos fundamentales relacionados con la utilización de semáforos en UNIX y su utilización para resolver problemas de sincronización/comunicación entre procesos concurrentes.

Semáforos en UNIX

Los semáforos no pueden ser implementados enteramente dentro del código de usuario por la simple razón de la atomicidad de las operaciones que se realizan sobre ellos. La ayuda necesaria se consigue mediante llamadas al sistema que manipulan los semáforos que son mantenidos dentro del kernel del S. O. los semáforos UNIX son más complejos que los semáforos binarios descritos en clase. Las diferencias básicas son:

1. Se puede crear un vector de semáforos con una sola llamada al sistema.
2. Los semáforos son multivaluados, no binarios. La idea es restringir el número máximo de usuarios simultáneos de un recurso a un límite máximo, que no sea necesariamente cero. El valor del semáforo indica el número de copias disponibles del recurso asociado. Cuando se usan como semáforos binarios trabajan como de costumbre.
3. Se puede especificar un vector de operaciones sobre el vector de semáforos.
4. UNIX proporciona mecanismos que permiten liberar automáticamente todos los cerrojos que un proceso mantiene sobre sus semáforos cuando sale.
5. Los semáforos tienen propietarios, modos de acceso y estampillas de tiempo, de forma parecida a los bloques de memoria compartida.

Toda esta flexibilidad es muy superior a la que la mayor parte de los usuarios pueden necesitar, y esto hace que las llamadas al sistema para manipular semáforos parezcan confusas.

A continuación se presenta con un mínimo de detalle una interfaz simplificada que permite crear, hacer operaciones wait y signal sobre un vector de semáforos binarios. Para más detalles estudia las llamadas al sistema semget(2), semop(2) y semctl(2) que a continuación se detallan tal y como aparecen en el manual del sistema operativo UNIX SunOS 5.8

semget - get set of semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```


La figura 1 muestra la llamada semget(2), utilizada para crear un vector nuevo de semáforos u obtener un handle de un conjunto existente previamente. Al igual que los segmentos de memoria compartida, los conjuntos de semáforos se identifican mediante una clave numérica. Los semáforos existen independientemente de cualquier proceso particular, y su existencia permanece después de que el proceso creador termina. Un proceso debe llamar a semget() con el bit IPC_CREAT a 1 en el argumento semflag de cara a crear el conjunto de semáforos. Posteriormente, otro proceso obtiene acceso al conjunto de semáforos llamando a semget() con la misma clave numérica, y el argumento semflag igual a cero.

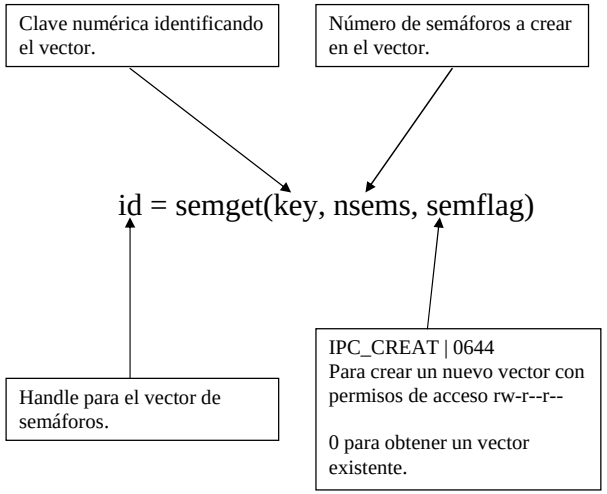


Figura 1. Creación de un vector de semáforos.

semop, semtimedop - semaphore operations

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, size_t nsops);

int semtimedop(int semid, struct sembuf *sops, size_t nsops, const struct timespec
*timeout);
```

Each sembuf structure contains the following members:

```
short   sem_num;    /* semaphore number */
short   sem_op;     /* semaphore operation */
short   sem_flg;    /* operation flags */
```

La figura 2 muestra la llamada semop(), utilizada para manipular los valores de semáforos. semop() maneja un vector de operaciones sobre semáforos en una única llamada, y se puede utilizar para realizar múltiples operaciones wait y signal sobre ellos. El sistema garantiza que el vector completo de operaciones se realiza de forma atómica, esto es, la llamada realizará el conjunto total de las operaciones o ninguna de ellas. Esto puede ser importante si se necesita realizar varias operaciones wait simultáneas sobre distintos semáforos antes de acceder a la sección crítica. (Problema de los filósofos sin bloqueos)

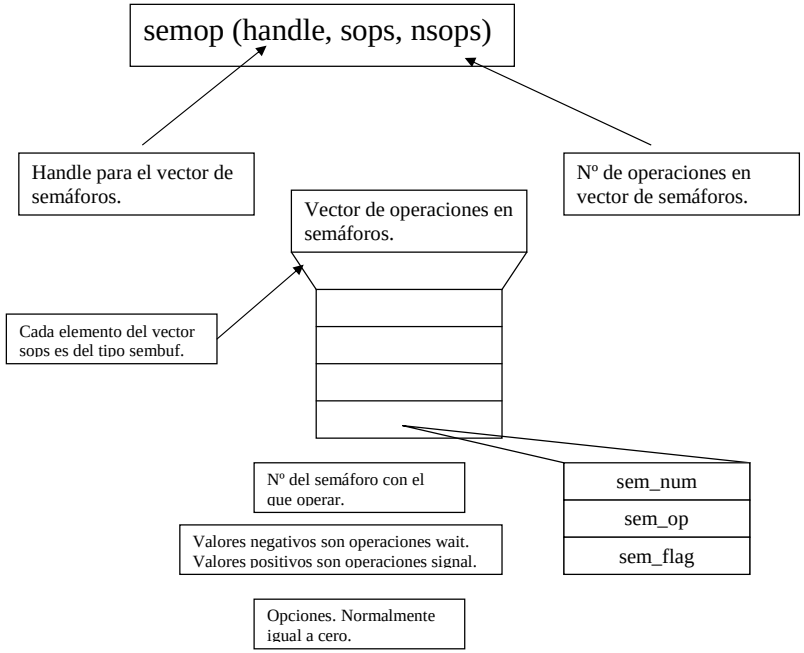


Figura 2. Operaciones sobre semáforos.

semctl - semaphore control operations

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```


Se puede ocultar la complejidad del trabajo con semáforos binarios por medio de funciones que permiten crear, hacer operaciones wait y signal sobre estos semáforos.

A continuación se detallan ejemplos (no necesariamente útiles para resolver problemas propuestos en esta práctica) de estas funciones.

/* Creación de un vector de n semáforos con clave k */

```
int crear_sem(key_t k, int n)
{
        int semid, i;

        /* Tratar de obtener un conjunto de semáforos existentes que tenga la clave. Si tiene existo, existirá ya un conjunto, que se debe destruir en primer lugar */

        /* Destruir el conjunto de semáforos preexistentes */

        if ((semid = semget(k, n, 0)) != -1) semctl(semid, 0, IPC_RMID);

        /* Los valores de los semáforos se inicializan a cero. A continuación se presenta un ejemplo de cómo inicializar todos a valor 1. La función semctl() con la orden SETALL es una manera más eficiente de hacer esto */

        if ((semid=semget(k, n, IPC_CREAT | 0600)) != -1)
        {
                arg.val = 1;
                for (i = 0; i < n; i++) semctl(semid, i, SETVAL, arg);
        }

        return(semid);
}
```

/* Operación wait sobre el semáforo n del conjunto sem */

```
void op_wait(int sem, int n)
{
        struct sembuf sop;      /* Construir un elemento de vector de operaciones */

        sop.sem_num = n;
        sop.sem_op = -1;        /* Operación wait u obtención una copia recurso */
        sop.sem_flg = 0;
        semop(sem, &sop, 1);            /* Operación wait propiamente dicha */
}
```

/* Operación signal sobre el semáforo n del conjunto sem */

```
void op_signal(int sem, int n)
{
        struct sembuf sop;      /* Construir un elemento de vector de operaciones */

        sop.sem_num = n;
        sop.sem_op = 1;         /* Operación signal o liberar una copia recurso */
        sop.sem_flg = 0;
        semop(sem, &sop, 1);            /* Operación signal propiamente dicha */
}
```

Estas funciones proporcionan simplicidad sacrificando flexibilidad. En particular las restricciones más importantes son que sólo soportan semáforos binarios que sólo permiten una operación simultánea sobre un semáforo.

Algunas estructuras relacinadas con semáforos muy útiles:

```
struct  sembuf
{
        ushort  sem_num;        /* Número de lugar en el conjunto de semáforos  */
        short  sem_op;          /* Operación;  < 0, 0, > 0                      */
        short  sem_flag;        /* IPC_NOWAIT, SEM_UNDO                         */
};

union  semun
{
        int  val;                       /* Para  SETVAL                         */
        struct semid_ds  *buf;          /* Para IPC_STAT  y IPC_SET             */
        ushort  *array;                 /* Para GETALL y SETALL                 */
};

struct  semi_ds
{
        struct ipc_perm  sem_perm;
        struct sem  *sem_base;          /* Puntero al 1º semáforo del conjunto  */
        ushort  sem_nsems;              /* Número de semáforos en el conjunto   */
        time_t  sem_otime;              /* Tiemo de la última operación semop( ) */
        time_t sem_ctime;               /* Tiempo del último cambio             */
};

struct  sem
{
        ushort  semval;         /* Valor del semáforo, siempre ≥  0             */
        pid_t  sempid;          /* PID de la última operación en el semáforo    */
        ushort  semncnt;        /* Número de procesos esperando semval ≥ 0      */
        ushort  semzcnt;        /* Número de procesos esperando semval = 0      */
};
```


PROBLEMAS A RESOLVER

Problema 1.

Construye un sistema de procesos que simule el problema productor/consumidor a través de un buffer compartido de capacidad limitada igual a 10.

La solución sólo debe contener el esquema de sincronización entre los procesos. La solución contendrá 2 procesos productores y 3 procesos consumidores.

La solución al problema de sincronización se realizará a través de semáforos contadores (no binarios).

Obtener una realización libre de bloqueos.

Obtener una traza, ordenada temporalmente, del estado de ocupación del buffer y de los procesos que acceden (utilizar como información para su ordenación, las estampillas de tiempo de la última operación sobre los semáforos). Analizar la traza y determinar si existen procesos que se encuentren accediendo concurrentemente al buffer. Analizar la traza y determinar si existen problemas de espera no limitadas por parte de algún proceso (starvation problems).

Comenta las decisiones de diseño más importantes.

Problema 2.

Construye un sistema de procesos que simule el problema de los cinco filósofos. Los tenedores se considerarán recursos cuya implementación se realizará a través de semáforos binarios.

Obtener una realización que esté libre de bloqueos.

Obtener una traza, ordenada temporalmente, de filósofos que alcanzan el estado "comiendo" (utiliza como información para su ordenación las estampillas de tiempo de la última operación sobre los semáforos). Analizar la traza y determinar si existen filósofos que se encuentren concurrentemente en el estado "comiendo". Analizar la traza y determinar si existen problemas de esperas no limitadas por parte de algún filósofo (starvation problems).

Apunta una solución a los problemas de esperas no limitadas.

Problema 3.

Construye un sistema de procesos que simule el problema de lectores y escritores donde un escritor accede a la base de datos en exclusión mutua con el resto de los procesos y no existen más restricciones de acceso.

La solución sólo debe contener el esquema de sincronización entre los procesos. La solución contendrá 2 escritores y 5 lectores.

Obtener una realización libre de bloqueos.

La sincronización se realizará mediante semáforos.

Obtener una traza, ordenada temporalmente, de los procesos que acceden a la base de datos (utilizar como información para su ordenación las estampillas de tiempo de la última operación sobre los semáforos). Analizar la traza y determinar si existen procesos que se encuentren accediendo concurrentemente a la base de datos. Analizar la traza y determinar si existen problemas de esperas no limitadas por parte de algún problema (starvation problems).

Comenta las decisiones de diseño más importantes.

ANEXO SEMÁFOROS

System Calls semget(2)

NAME
semget - get set of semaphores

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

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

DESCRIPTION
The semget() function returns the semaphore identifier associated with key.

A semaphore identifier and associated data structure and set containing nsems semaphores (see intro(3)) are created for key if one of the following is true:

- o key is equal to IPC_PRIVATE.

- o key does not already have a semaphore identifier associated with it, and (semflg&IPC_CREAT) is true.

On creation, the data structure associated with the new semaphore identifier is initialized as follows:

- o sem_perm.cuid, sem_perm.uid, sem_perm.cgid, and sem_perm.gid are set equal to the effective user ID and effective group ID, respectively, of the calling process.

- o The access permission bits of sem_perm.mode are set equal to the access permission bits of semflg.

- o sem_nsems is set equal to the value of nsems.

- o sem_otime is set equal to 0 and sem_ctime is set equal to the current time.

RETURN VALUES
Upon successful completion, a non-negative integer representing a semaphore identifier is returned. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS
The semget() function will fail if:

EACCES
A semaphore identifier exists for key, but operation permission (see intro(3)) as specified by the low-order 9 bits of semflg would not be granted.

EEXIST
A semaphore identifier exists for key but both (semflg&IPC_CREAT) and (semflg&IPC_EXCL) are both true.

EINVAL
The nsems argument is either less than or equal to 0 or greater than the system-imposed limit; or a semaphore identifier exists for key, but the number of semaphores in the set associated with it is less than nsems and nsems is not equal to 0.

ENOENT
A semaphore identifier does not exist for key and (semflg&IPC_CREAT) is false.

ENOSPC
A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores or semaphore identifiers system-wide would be exceeded.

SEE ALSO
ipcrm(1), ipcs(1), intro(3), semctl(2), semop(2), ftok(3C)

System Calls semctl(2)

NAME
semctl - semaphore control operations

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

int semctl(int semid, int semnum, int cmd, ...);
```

DESCRIPTION
The semctl() function provides a variety of semaphore control operations as specified by cmd. The fourth argument is optional, depending upon the operation requested. If required, it is of type union semun, which must be explicitly declared by the application program.

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

The permission required for a semaphore operation is given as {token}, where token is the type of permission needed.
The types of permission are interpreted as follows:

00400	READ by user
00200	ALTER by user
00040	READ by group
00020	ALTER by group
00004	READ by others
00002	ALTER by others

See the Semaphore Operation Permissions subsection of the DEFINITIONS section of intro(2) for more information. The following semaphore operations as specified by cmd are executed with respect to the semaphore specified by semid and semnum.

GETVAL
Return the value of semval (see intro(2)). {READ}

SETVAL
Set the value of semval to arg.val. {ALTER} When this command is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.

GETPID
Return the value of (int) sempid. {READ}

GETNCNT
Return the value of semncnt. {READ}

GETZCNT
Return the value of semzcnt. {READ}

The following operations return and set, respectively, every semval in the set of semaphores.

GETALL
Place semvals into array pointed to by arg.array. {READ}

SETALL
Set semvals according to the array pointed to by arg.array. {ALTER}. When this cmd is successfully executed, the semadj values corresponding to each specified semaphore in all processes are cleared.

The following operations are also available.

IPC_STAT
Place the current value of each member of the data structure associated with semid into the structure pointed to by arg.buf. The contents of this structure are defined in intro(2). {READ}

IPC_SET
Set the value of the following members of the data structure associated with semid to the corresponding value found in the structure pointed to by arg.buf:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode  /* access permission bits only */
```

This command can be executed only by a process that has an effective user ID equal to either that of super-user, or to the value of sem_perm.cuid or sem_perm.uid in the data structure associated with semid.

IPC_RMID
Remove the semaphore identifier specified by semid from the system and destroy the set of semaphores and data structure associated with it. This command can only be executed by a process that has an effective user ID equal to either that of super-user, or to the value of sem_perm.cuid or sem_perm.uid in the data structure associated with semid.

RETURN VALUES
Upon successful completion, the value returned depends on cmd as follows:

GETVAL
the value of semval

GETPID
the value of (int) sempid

GETNCNT
the value of semncnt

GETZCNT
the value of semzcnt

All other successful completions return 0; otherwise, -1 is returned and errno is set to indicate the error.

ERRORS
The semctl() function will fail if:

EACCES
Operation permission is denied to the calling process (see intro(2)).

EINVAL
The semid argument is not a valid semaphore identifier; the semnum argument is less than 0 or greater than sem_nsems -1; or the cmd argument is not a valid command or is IPC_SET and sem_perm.uid or sem_perm.gid is not valid.

EPERM The cmd argument is equal to IPC_RMID or IPC_SET and the effective user of the calling process is not super-user, or cmd is equal to the value of sem_perm.cuid or sem_perm.uid in the data structure associated with semid.

EOVERFLOW
The cmd argument is IPC_STAT and uid or gid is too large to be stored in the structure pointed to by arg.buf.

ERANGE
The cmd argument is SETVAL or SETALL and the value to which semval is to be set is greater than the system imposed maximum.

SEE ALSO
ipcs(1), intro(2), semget(2), semop(2)

System Calls semop(2)

NAME
semop, semtimedop - semaphore operations

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

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

int semtimedop(int semid, struct sembuf *sops, size_t nsops, const struct timespec *timeout);
```

DESCRIPTION
The semop() function is used to perform atomically an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by semid. The sops argument is a pointer to the array of semaphore-operation structures. The nsops argument is the number of such structures in the array.

Each sembuf structure contains the following members:

```
short   sem_num;   /* semaphore number */
short   sem_op;    /* semaphore operation */
short   sem_flg;   /* operation flags */
```

Each semaphore operation specified by sem_op is performed on the corresponding semaphore specified by semid and sem_num. The permission required for a semaphore operation is given as {token}, where token is the type of permission needed.
The types of permission are interpreted as follows:

```
00400   READ by user
00200   ALTER by user
00040   READ by group
00020   ALTER by group
00004   READ by others
00002   ALTER by others
```

See the Semaphore Operation Permissions section of intro(3) for more information.

The sem_op member specifies one of three semaphore operations:

1. The sem_op member is a negative integer; {ALTER}

o If semval (see intro(3)) is greater than or equal to the absolute value of sem_op, the absolute value of sem_op is subtracted from semval. Also, if(sem_flg&SEM_UNDO) is true, the absolute value of sem_op is added to the calling process's semadj value (see exit(2)) for the specified semaphore.

o If semval is less than the absolute value of sem_op and (sem_flg&IPC_NOWAIT) is true, semop() returns immediately.

o If semval is less than the absolute value of sem_op and (sem_flg&IPC_NOWAIT) is false, semop() increments the semncnt associated with the specified semaphore and suspends execution of the calling process until one of the following conditions occur:

o The value of semval becomes greater than or equal to the absolute value of sem_op. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of sem_op is subtracted from semval and, if (sem_flg&SEM_UNDO) is true, the absolute value of sem_op is added to the calling process's semadj value for the specified semaphore.

o The semid for which the calling process is awaiting action is removed from the system (see semctl(2)). When this occurs, errno is set to EIDRM and -1 is returned.

o The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in signal(3C).

2. The sem_op member is a positive integer; {ALTER}

The value of sem_op is added to semval and, if (sem_flg&SEM_UNDO) is true, the value of sem_op is subtracted from the calling process's semadj value for the specified semaphore.

3. The sem_op member is 0; {READ}

o If semval is 0, semop() returns immediately.

o If semval is not equal to 0 and (sem_flg&IPC_NOWAIT) is true, semop() returns immediately.

o If semval is not equal to 0 and (sem_flg&IPC_NOWAIT) is false, semop() increments the semzcnt associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

o The value of semval becomes 0, at which time the value of semzcnt associated with the specified semaphore is decremented.

o The semid for which the calling process is awaiting action is removed from the system. When this occurs, errno is set to EIDRM and -1 is returned.

o The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in signal(3C).

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by sops is set to the process ID of the calling process.

The semtimedop() function behaves as semop() except when it must suspend execution of the calling process to complete its operation. If semtimedop() must suspend the calling process after the time interval specified in timeout expires, or if the timeout expires while the process is suspended, semtimedop() returns with an error. If the timespec structure pointed to by timeout is zero-valued and semtimedop() needs to suspend the calling process to complete the requested operation(s), it returns immediately with an error. If timeout is the NULL pointer, the behavior of semtimedop() is identical to that of semop().

RETURN VALUES

Upon successful completion, 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

The semop() and semtimedop() functions will fail if:

E2BIG The nsops argument is greater than the system-imposed maximum.

EACCES

Operation permission is denied to the calling process (see intro(3)).

EAGAIN

The operation would result in suspension of the calling process but (sem_flg&IPC_NOWAIT) is true.

EFAULT

The sops argument points to an illegal address.

EFBIG The value of sem_num is less than 0 or greater than or equal to the number of semaphores in the set associated with semid.

EIDRM A semid was removed from the system.

EINTR A signal was received.

EINVAL

The semid argument is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the limit.

ENOSPC

The limit on the number of individual processes requesting an SEM_UNDO would be exceeded.

ERANGE

An operation would cause a semval or a semadj value to overflow the system-imposed limit.

The semtimedop() function will fail if:

EAGAIN

The timeout expired before the requested operation could be completed.

The semtimedop() function will fail if one of the following is detected:

EFAULT

The timeout argument points to an illegal address.

EINVAL

The timeout argument specified a tv_sec or tv_nsec value less than 0, or a tv_nsec value greater than or equal to 1000 million.

SEE ALSO

ipcs(1), intro(3), exec(2), exit(2), fork(2), semctl(2), semget(2)

Práctica 6: IPC: Sockets

Objetivo:

La práctica tiene como objetivo el análisis de una aplicación y la programación de diversas soluciones cliente-servidor utilizando como mecanismos de comunicación y/o sincronización **sockets**.

6. Sockets en UNIX

La aplicación estará basada en un servidor que realizará el trabajo especificado en las funciones **cuenta_primos(), encuentra_primos()** y **esprimo()**, descritas en el programa secuencial presentado más abajo. Los clientes podrán pedir los servicios correspondientes a la funcionalidad expresada en las funciones **cuenta_primos()** y/o **encuentra_primos()** al servidor. La función **esprimo()** es interna al servidor y no visible por los clientes. La función **main()** del programa secuencial presentado a continuación pude considerarse como el núcleo de los clientes a construir.

```
/*        funcpri.h        */

#include <stdio.h>

#define MINIMO 1L
#define MAXIMO 1000000L


/* Cuenta los primos entre min y max */

long cuenta_primos(long min, long max);


/* Cuenta y encuentra los primos entre min y max y los devuelve en un vector */

long encuentra_primos(long min, long max, long vector[]);


/* Devuelve TRUE si n es primo */

int esprimo(long n);
```



```
/*        funcpri.c        */

#include "funcpri.h"


long cuenta_primos(long min, long max)
{
  long i, contador;

  contador = 0;

  for (i = min; i <= max; i++)
    if (esprimo(i)) contador = contador + 1;

  return (contador);
}


long encuentra_primos(long min, long max, long vector[])
{
  long i, contador;

  contador = 0;

  for (i = min; i <= max; i++)
    if (esprimo(i)) vector[contador++] = i;

  return (contador);
}


int esprimo(long n)
{
  long i;

  for (i = 2; i*i <= n; i++)
    if ((n % i) == 0) return (0);

  return (1);
}
```

```
/*       servpri.c        */

#include "funcpri.h"

/* Escribe una lista de números primos entre 1 y 1000000 */

int main()
{
  long i, cuantos;
  long primos[10000];

  cuantos = cuenta_primos(MINIMO, MAXIMO);
  printf("\n El número de primos es: %ld \n", cuantos);

  cuantos = encuentra_primos(MINIMO, MAXIMO, primos);

  for (i = 0; i < cuantos; i++)
    printf("\n %ld es primo \n", primos[i]);

  exit(0);
}
```

La práctica consistirá en las siguientes realizaciones:

1) Diseña un servidor que atienda peticiones de un cliente para determinar cuántos primos hay en el intervalo especificado o bien devolver la lista de primos existentes en dicho intervalo. Diseña también un cliente que utilice los servicios del servidor. En el guión de prácticas se incluirá el análisis realizado para determinar el tipo de interacción cliente-servidor por paso de mensajes seleccionada.

   En el diseño de la aplicación cliente-servidor se incluirán mecanismos para implementar las siguientes especificaciones adicionales:

   a) Recuperación de errores debidos a la transmisión. Se realizará un análisis de las posibles fuentes de error y los mecanismos implementados para recuperar los errores. Adicionalmente se incluirá el código desarrollado para probar dichos mecanismos y que se encargará de inyectar errores, de los tipos considerados, a la aplicación.

   b) Concurrencia/paralelismo a considerar en la atención de múltiples clientes por parte del servidor. Los mecanismos a introducir deberán tener en cuenta las características de la aplicación y deberán estar dirigidos a obtener ganancia en el tiempo de cálculo con relación al caso de un servidor iterativo.

2) Diseña la aplicación considerando la existencia de múltiples servidores. En este caso, un cliente dividirá el trabajo (el intervalo en el cual hay que calcular el número de primos o los números primos) en subintervalos y cada uno de éstos será enviado a un servidor diferente. El cliente dará por concluido el trabajo en el momento que se hayan recibido todas las respuestas de los servidores para todos lo paquetes de trabajo en los que el cliente ha dividido el trabajo total. Si un servidor ha terminado de procesar el subintervalo solicitado por el cliente, y al cliente todavía le quedan por calcular subintervalos, el cliente enviará a este servidor una nueva petición con el subintervalo pendiente.

Práctica 7: IPC: RPC

Objetivo:

La práctica tiene como objetivo el análisis de una aplicación y la programación de diversas soluciones cliente-servidor utilizando como mecanismos de comunicación y/o sincronización **RPC**.

7. RPC en UNIX

La práctica tiene como objetivo la construcción de una aplicación cliente-servidor basada en RPC. La aplicación estará basada en un servidor que contendrá los procedimientos **cuenta_primos( )**, **encuentra_primos( )** y **esprimo( )** con una funcionalidad como la descrita en el programa secuencial presentado más abajo. Los clientes podrán llamar a las funciones **cuenta_primos( )** y/o **encuentra_primos( )** contenidas en el servidor. La función **esprimo()** es interna al servidor y no visible por los clientes. La función **main( )** del programa secuencial presentado a continuación puede considerarse como ejemplo de los potenciales clientes.

/* Escribe una lista de números primos entre 1 y 1000 */

```
int main()
{
  int i, cuantos;
  int primos[1000];

  cuantos = cuenta_primos(1, 1000);
  printf("\n El número de primos es: %d \n", cuantos);

  cuantos = encuentra_primos(1, 1000, primos);

  for (i = 0; i < cuantos; i++)
    printf("\n %d es primo \n", primos[i]);

  exit(0);
}
```


/* Contar los primos entre min y max */

```
int cuenta_primos(int min, int max)
{
  int i, contador;

  contador = 0;

  for (i = min; i <= max; i++)
    if (esprimo(i)) contador = contador + 1;

  return (contador);
}
```

/* Encontrar los primos entre min y max, y devolverlos en vector */

```
int encuentra_primos(int min, int max, int vector[])
{
  int i, contador;

  contador = 0;

  for (i = min; i <= max; i++)
    if (esprimo(i)) vector[contador++] = i;

  return (contador);
}
```

/* Devuelve TRUE si n es primo */

```
int esprimo(int n)
{
  int i;

  for (i = 2; i*i <= n; i++)
    if ((n % i) == 0) return (0);

  return (1);
}
```

Como especificaciones adicionales se tendrán en cuenta los siguientes puntos:

1) El servidor realizará una autenticación tipo UNIX soportada en las llamadas RPC. Además, el servidor contendrá una política de control de acceso basada en una lista de usuarios autorizados mantenida en un fichero.

2) Las funciones contenidas en el servidor deberán realizar un control de errores en el ámbito de usuario y devolver un informe de los errores producidos al cliente. Este control de errores consistirá exclusivamente en verificar que min < max.
   Ayuda: utilícense uniones discriminadas para devolver diferentes datos en los casos de llamadas culminadas con éxito y llamadas culminadas con error a la funciones del servidor.)

Para la realización de esta práctica se recomienda seguir los siguientes pasos:

a) Diseño del fichero de interfase (**primos.x**) El fichero contendrá la declaración de los tipos de los parámetros y resultados de las funciones del servidor visibles por los clientes, así como la declaración de las funciones visibles contenidas en el servidor. El lenguaje que se utilizará para escribir este fichero es el lenguaje XDR. Téngase en cuenta que probablemente hay que cambiar la declaración de las funciones dadas en el programa anterior para adaptarse a las reglas de construcción de servidores RPC.

b) Mediante la utilidad **rpcgen** genera los ficheros: **primos.h**, **primos_xdr.c**, **primos_clnt.c** y **primos_svc.c**. Se recomienda estudiar los contenidos de estos ficheros e interpretarlos. En particular es sumamente importante conocer los contenidos del fichero **primos.h**, ya que en él se encuentra la especificación de los tipos de los parámetros y resultados de las funciones contenidas en el servidor y que harán falta a la hora de construir el código adaptado de las funciones y de los clientes.

c) Estudia el método de autenticación tipo UNIX soportado por las llamadas RPC. A continuación se comentan los puntos más importantes, recomendándose contrastar estas notas con la información contenida en el sistema donde se realicen estas prácticas.

   1) Para pasar las credenciales de AUTH_UNIX, el cliente debe adjuntar una structura de autorización al manejador obtenido de la llamada **clnt_create().** Para crear esta estructura se puede utilizar la función de librería RPC: **authunix_create_default().** El código para realizar esto es:

```
cl = clnt_create( … )
cl -> cl_auth = authunix_create_default();
```


2) La función contenida en el servidor puede recuperar las credenciales enviadas por el cliente por medio del segundo parámetro que se pasa en la llamada. Este argumento, que no se incluye en los casos de aplicaciones sin autenticación, es del tipo (struct svc_req *).

   La estructura svc_req es como sigue:

```
struct svc_req
{
        u_long rq_prog;              /* número de programa servidor */
        u_long rq_vers;              /* versión de programa servidor */
        u_long rq_proc;              /* número de función deseada */
        struct opaque_auth rq_cred;  /* credenciales en bruto */
        caddr_t rq_clntcred;         /*credenciales ad hoc sólo lectura */
        SVCXPRT *rq_xprt;            /* transporte asociado */
};
```

Los campos de interés son rq_cred que mantiene una estructura conteniendo entre otras cosas el tipo de autenticación, y el campo rq_clntcred que apunta a una estructura (dependiente del tipo de autentificación) conteniendo los datos reales de autentificación.

3) El servidor necesita en primer lugar verificar que tipo de autenticación se está considerando. Esto se representa mediante una de las tres siguientes constantes: AUTH_NONE, AUTH_UNIX Y AUTH_DES (definidas en el fichero <rpc/auth.h>)

4) Una vez localizado el tipo de autentiación, el servidor debe localizar en la estructura svc_req el nombre de la máquina del cliente, el identificador de usuario, etc. Antes de que se llame a la función contenida en el servidor, se extrae toda esta información del mensaje RPC y se deposita en la estrucutra authunix_parms que se encuentra declarada en el fichero <rpc/auth_sys.h>:

```
struct authunix_parms
{
        u_long aup_time;
        char *aup_machname;      /* nombre del host cliente */
        uid_t aup_uid;           /* ID de usuario del cliente */
        gid_t aup_gid;           /* ID de grupo del cliente */
        u_int aup_len;           /* longitud de la lista del grupo */
        gid_t *aup_gids;         /* lista de grupos */
};
```

En algunas versiones de RPC, la estructura authunix_parms se denomina authsys_parms y la constante AUTH_UNIX se denomina AUTH_SYS.

5) A continuación se muestra un trozo de código que ilustra una función contenida en el servidor y cómo ésta recupera las credenciales para una autenticación tipo AUTH_UNIX.

```
int *mi_funcion_servidora(mi_rep *r, struct svc_req *rqstp)
{
        struct authunix_parms *ucred;
        fprintf(stderr, "flavor = %d \n", rqstp->rq_cred.oa_flavor);

        if (rqstp->rq_cread.oa_flavor == AUTH_UNIX)
        {
                ucred = (struct authunix_parms *) (rqstp->rq_clntcred);
                fprintf(stderr, "host = %s \n", ucred->aup_machname);
                fprintf(stderr, "uid = %d \n", ucred->aup_uid);
                fprintf(stderr, "gid = %d \n", ucred->aup_gid);
        }

        /* Cuerpo de la función servidora */
}
```

d) Construye el servidor incluyendo el código de las funciones visibles desde los clientes, así como el código de las funciones auxiliares (este fichero se puede llamar por ejemplo, primos_funcs.c).

e) Construye el cliente en un fichero que se llame por ejemplo primos_main.c
gcc –o servidor primos_funcs.o primos_svc.o primos_xdr.o

f) Compila y monta los enlaces pertienetes ("link") del cliente y el servidor:
gcc –o cliente primos_main.o primos_clnt.o primos_xdr.o

g) Verifica la aplicación, incluso con varios clientes.