

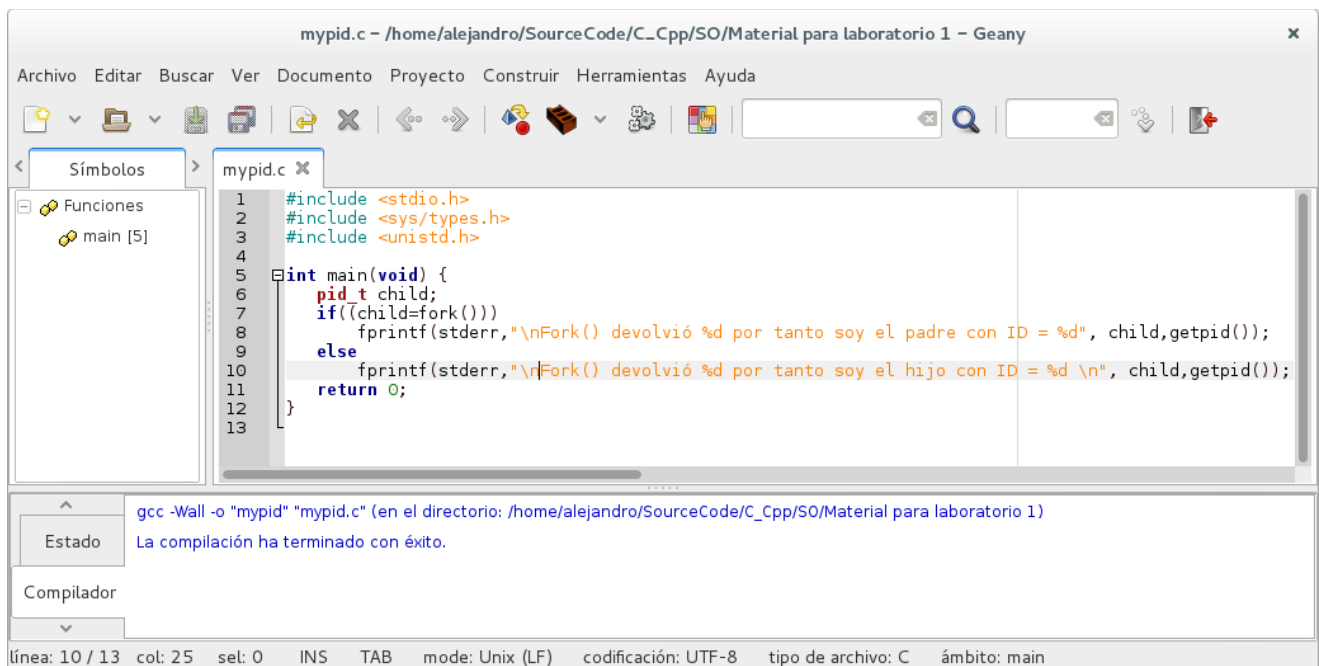
## Tema: Comunicación entre procesos relacionados

### PARTE A

**Resumen:** Llamadas al sistema que debe revisar: *fork()*, *wait()*, *waitpid()*, *fflush()*, *getpid()*, *getppid()*

#### Los primeros ejemplos:

1.- Este primer programa crea un proceso hijo y tanto el padre como el hijo muestran por pantalla su identificador de proceso (*pid*). Recuerde que en el momento que se ejecuta la llamada al sistema *fork()* son dos procesos en ejecución simultánea (multiprogramación) cuya línea de partida es el valor que devolvió el *fork()* tanto al padre como al hijo (el padre recibe el *pid* del hijo y el hijo recibe 0). De esta manera se puede bifurcar la ejecución para que el padre ejecute una acción diferente al hijo.

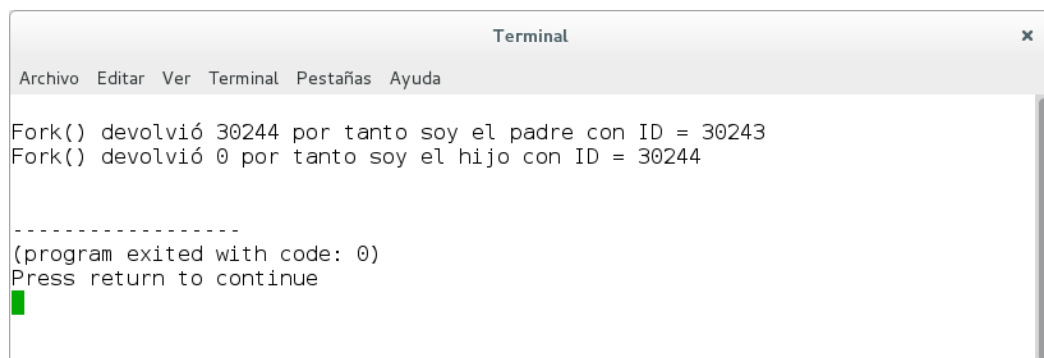


The screenshot shows the Geany IDE with the file 'mypid.c' open. The code is as follows:

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main(void) {
6     pid_t child;
7     if((child=fork()))
8         fprintf(stderr, "\nFork() devolvió %d por tanto soy el padre con ID = %d", child, getpid());
9     else
10        fprintf(stderr, "\nFork() devolvió %d por tanto soy el hijo con ID = %d \n", child, getpid());
11    return 0;
12 }
13
```

Below the editor, the 'Estado' (Status) panel shows the compilation command: `gcc -Wall -o "mypid" "mypid.c" (en el directorio: /home/alejandro/SourceCode/C_Cpp/SO/Material para laboratorio 1)` and the message: 'La compilación ha terminado con éxito.'

Al ejecutar el programa obtenemos la siguiente salida:

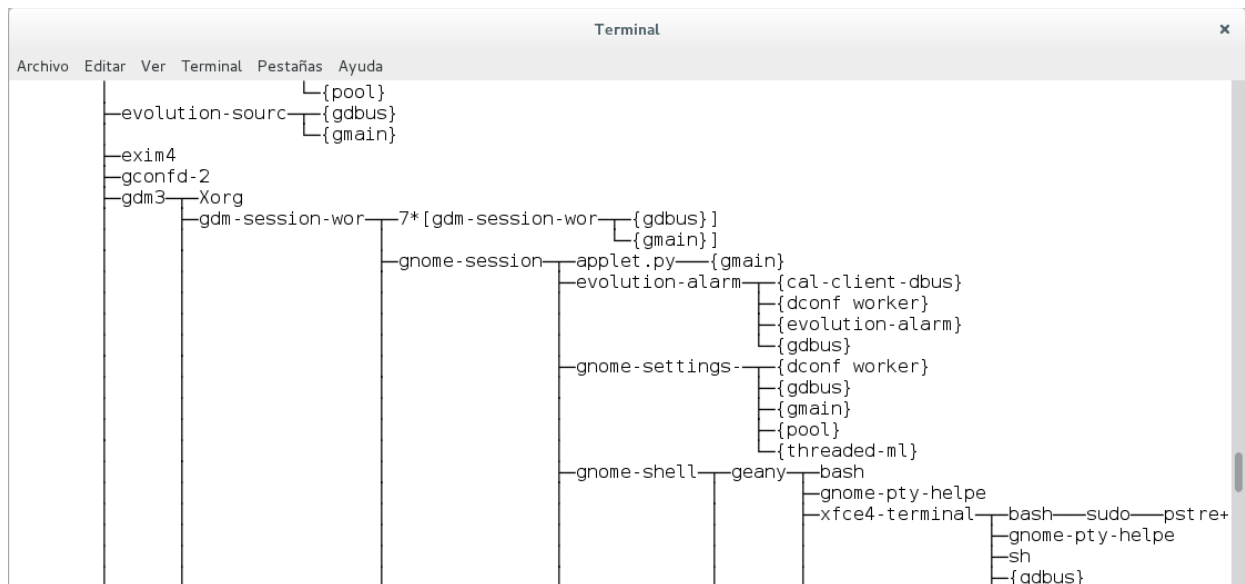


The terminal window shows the following output:

```
Fork() devolvió 30244 por tanto soy el padre con ID = 30243
Fork() devolvió 0 por tanto soy el hijo con ID = 30244

-----
(program exited with code: 0)
Press return to continue
```

Sería interesante ver de forma gráfica el árbol de procesos. Existe el programa *pstree* que muestra el árbol de procesos mostrando su dependencia. Observe el programa en ejecución:



Podemos incluir algunas línea en el código de forma que podamos ver el árbol de procesos. Nuestro programa quedaría modificado de la siguiente forma:

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5
6 int main(void) {
7     pid_t child;
8     if((child=fork()))
9         fprintf(stderr, "\nFork() devolvió %d por tanto soy el padre con ID = %d", child, getpid());
10    else {
11        fprintf(stderr, "\nFork() devolvió %d por tanto soy el hijo con ID = %d\n", child, getpid());
12        system("pstree > aprocesos.txt");
13    }
14    return 0;
15 }
  
```

gcc -Wall -o "mypid" "mypid.c" (en el directorio: /home/alejandro/SourceCode/C\_Cpp/SO/Material para laboratorio 1)  
La compilación ha terminado con éxito.

línea: 15 / 16 col: 12 sel: 0 INS TAB mode: Unix (LF) codificación: UTF-8 tipo de archivo: C ámbito: main

Compile y ejecute el programa. La salida será semejante a la anterior, pero se habrá creado un archivo con nombre *aprocesos.txt* conteniendo la salida del comando *ps*. Este archivo es creado en el mismo lugar donde se encuentra el ejecutable. A continuación mostramos el archivo en la parte que nos interesa:

```

|-gvfsd-http---{gdbus}
|-gvfsd-metadata---{gdbus}
|-gvfsd-trash---{gdbus}
|   |--{gmain}
|   |--2*[{pool}]
|-minissdpd
|-mission-control---{dconf worker}
|   |--{gdbus}
|--mypid---sh---pstree
|-nxserver.bin---nxd---5*[{nxd}]
|   |--nxnode.bin---nxclient.bin---9*[{nxclient.bin}]
|   |--16*[{nxnode.bin}]
|   |--5*[{nxserver.bin}]
|-oosplash---soffice.bin---{ICEConnectionWo}
  
```

Sin embargo aquí hay algo que no concuerda, nuestro programa debería crear un proceso hijo que tiene igual nombre pero diferente *pid*. Es decir debía aparecer algo así como:

```
mypid --- mypid --- sh --- pstree
```

La explicación es muy sencilla, uno de los procesos, en este caso el proceso padre ha terminado antes, y cuando el proceso hijo ejecuta *pstree*, solo aparece el proceso hijo.

### Tarea 1:

a) Modifique el programa para que el padre espere al hijo, de forma que cuando el hijo “tome la foto” con *pstree*, el padre salga en ella. La salida del archivo *ejmp2.4.out* debe ser semejante a la siguiente:

```
e
--gnome-ptty-help
| -sh---mypid---mypid---sh---pstree
| -{gdbus}
| -{gmain}
| -{pool}
```

b) Explique la presencia del proceso *sh*.

c) Lea el manual de *pstree* y vea si hay alguna forma de sólo mostrar la rama correspondiente a un proceso proporcionándole su *pid*. Modifique el programa para obtener esta salida.

d) Modifique el programa para que el archivo *aprocesos.txt* sea creado en el directorio donde se encuentra sus programas fuentes y no en el *home*.

**Nota:** recuerde que al incluir nuevas funciones de librería o llamadas al sistema, también debe indicar los correspondientes archivos de cabecera (archivos *.h*)

2.- Escriba los siguientes programas (*chainprocesses.c* y *ejmp2.6.c*)

```
mypid.c x chainprocesses.c x
5
6  /* Este programa crea una cadena de procesos. Es decir el padre */
7  /* crea un hijo, este a su vez crea otro y así en forma sucesiva */
8  /* Ejm 2.5 del libro UNIX Programacion Practica - Kay Robbins */
9  /*                               Steve Robbins */
10 /* Modificado por Alejandro Bello Ruiz - Informática PUCP */
11
12 int main(void){
13     int i,status;
14     pid_t child;    /* pid_t es un tipo definido en types.h */
15
16     for (i=1;i<4;++i) if((child=fork())) break;
17     fprintf(stderr,"Esta es la vuelta Nro %d\n",i);
18     fprintf(stderr,"Recibi de fork el valor de %d\n",child);
19     fprintf(stderr,"Soy el proceso %d con padre %d\n\n",getpid(),getppid());
20     wait(&status);
21     return 0;
22 }
23
```

```

mypid.c x chainprocesses.c x fanprocesses.c x
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  /* Este programa crea un abanico de procesos. Es decir el padre crea */
7  /* tres hijo. */
8  /* Ejm 2.6 del libro UNIX Programacion Practica - Kay Robbins */
9  /* Steve Robbins */
10 /* Modificado por Alejandro Bello Ruiz - Informática PUCP */
11
12 void main(void){
13     int i,status;
14     pid_t child,pid_padre;
15
16     pid_padre=getpid();
17     for(i=1;i<4;++i)
18         if((child=fork())<=0) break;
19     else fprintf(stderr,"Este es el ciclo Nro %d y se esta creando el proceso %d\n",i,child);
20     if(pid_padre==getpid()) for(i=1;i<4;++i) wait(&status);
21     fprintf(stderr,"Este es el proceso %d con padre %d\n",getpid(),getppid());
22     return 0;
23 }
24

```

Compile y ejecute cada uno de los programas.

### Tarea 2

- Modifique el programa *chainprocesses.c* para que se genere el archivo *chainprocesses.txt* que contendrá la salida del programa *ps* mostrando el árbol de procesos creados por el proceso *padre*.
- Modifique el programa *fanprocesses.c* para que se genere el archivo *fanprocesses.txt* que contendrá la salida del programa *ps* mostrando el árbol de procesos creados por el proceso *padre*.

3.- Para obtener el árbol de procesos mostrados por *ps* está claro que todos los procesos que queremos que aparezcan en la salida deben estar ejecutandose. Pero hay situaciones en las que el proceso se crea e inmediatamente muere y al final se desea obtener el árbol de procesos que se creó, obviamente *ps* no nos servirá de mucho puesto que la mayoría de procesos ya murieron. Podría forzarse para que los procesos sigan ejecutandose, pero a veces el código no lo permite

Escriba, compile, enlace y ejecute los siguientes programas (*multifork.c* y *isengfork.c*):

```

mypid.c x chainprocesses.c x fanprocesses.c x multifork.c x
1  /* multifork.c (c) 2005-2009 Rahmat M. Samik-Ibrahim, GPL-like */
2  /* ***** */
3  #include <sys/types.h>
4  #include <sys/wait.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #define DISPLAY1 "PID INDUK** ** pid (%5.5d) ** *****\n"
9  #define DISPLAY2 "val1(%5.5d) -- val2(%5.5d) -- val3(%5.5d)\n"
10 /* ***** main ***** */
11
12 int main(void) {
13     pid_t val1, val2, val3;
14     printf(DISPLAY1, (int) getpid());
15     fflush(stdout);
16     val1 = fork();
17     waitpid(-1, NULL, 0);
18     val2 = fork();
19     waitpid(-1, NULL, 0);
20     val3 = fork();
21     waitpid(-1, NULL, 0);
22     printf(DISPLAY2, (int) val1, (int) val2, (int) val3);
23     return 0;
24 }

```

```

mypid.c x chainprocesses.c x fanprocesses.c x multifork.c x isengfork.c x
1  /* isengfork.c (c) 2007-2009 Rahmat M. Samik-Ibrahim, GPL-like */
2  /* ***** */
3  #include <sys/types.h>
4  #include <sys/wait.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  int main(void) {
9      int ii=0;
10     if (fork() == 0) ii++;
11     waitpid(-1,NULL,0);
12     if (fork() == 0) ii++;
13     waitpid(-1,NULL,0);
14     if (fork() == 0) ii++;
15     waitpid(-1,NULL,0);
16     printf ("Result = %3.3d \n",ii);
17     return 0;
18 }
19

```

### Tarea 3

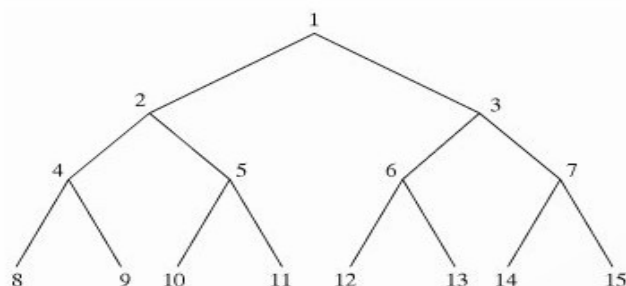
- Haciendo uso de los datos proporcionados por el programa *multifork.c* elabore un esquema que represente el árbol de procesos generados por el proceso padre.
- En el programa *multifork.c* cuál es la interpretación de su salida.
- Haciendo uso de los datos proporcionados por el programa *isengfork.c* elabore un esquema que represente el árbol de procesos generados por el proceso padre.
- En el programa *isengfork.c* cuál es la interpretación de su salida.

Si cree necesario puede agregar instrucciones adicionales para lograr su objetivo.

4.- (Pregunta 1 del primer laboratorio calificado del semestre 2010-1) Escriba un programa que reciba como argumento un número natural  $n$  (mayor que 0) y cree un **árbol binario** de procesos de profundidad  $n$ . Cuando sea creado el árbol, cada proceso deberá imprimir la frase “Soy el proceso  $x$ ” y luego terminar. Por ejemplo, si el usuario ingresa lo siguiente:

```
$ binarytree 4 ... construirá un árbol de profundidad 4.
```

luego el árbol de procesos debería tener una apariencia semejante a la siguiente:



y la salida debería ser:

```

Soy el proceso 1
Soy el proceso 2
Soy el proceso 3

```

Soy el proceso 4  
Soy el proceso 5  
Soy el proceso 6  
Soy el proceso 7  
Soy el proceso 8  
Soy el proceso 9  
Soy el proceso 10  
Soy el proceso 11  
Soy el proceso 12  
Soy el proceso 13  
Soy el proceso 14  
Soy el proceso 15

### Solución

```
binarytree.c ×
1  /* binarytree.c (c) 2010 Alejandro T. Bello Ruiz, GPL-like */
2  /* ***** */
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <math.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <sys/wait.h>
9
10 double final;
11
12 void crea_arbol(int);
13
14 int main(int narg, char *argv[])
15 { int n;
16
17     n=atoi(argv[1]);
18     final=pow(2,(n-1));
19     crea_arbol(1);
20     return 0;
21 }
22
23 void crea_arbol(int x)
24 { char cadena[60];
25
26     sprintf(cadena,"Soy el proceso %d con pid %d y ppid %d\n",x,getpid(),getppid());
27     write(1,cadena,strlen(cadena));
28     if(x >= final) return;
29     if(!fork()) { crea_arbol(2*x); exit(0); }
30     if(!fork()) { crea_arbol(2*x+1);exit(0);}
31     wait(NULL);
32     wait(NULL);
33 }
34
```

Se han añadido las llamadas al sistema *getpid()* y *getppid()*, para verificar que los procesos han sido creados formando un árbol binario tal como se ha solicitado en la pregunta.

### Tarea 4

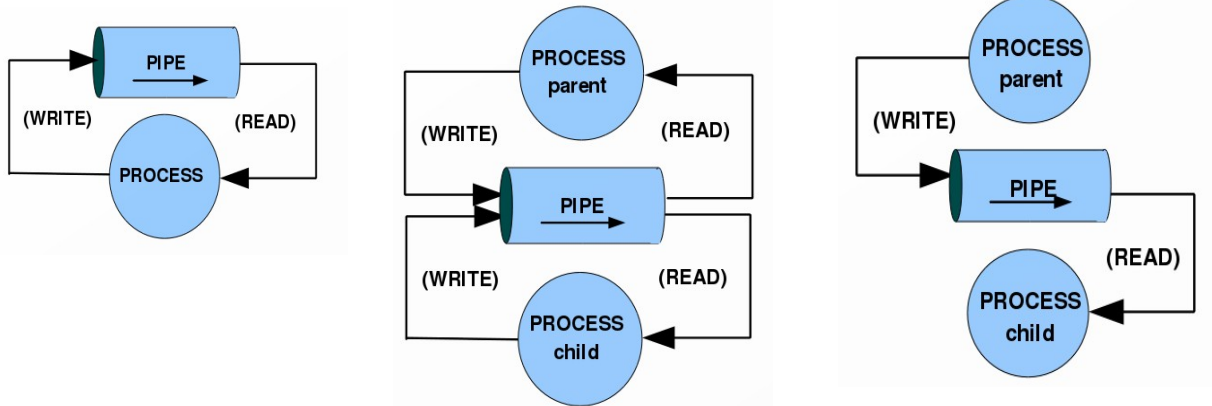
- Modifique el programa para obtener el archivo *binarytree.txt* de forma que contenga la salida del programa *ps tree*, mostrando el árbol de procesos.
- Escriba un programa que cree un árbol de procesos, semejante al anterior, pero que el recorrido que se realiza para la creación de procesos sea siguiendo el *esquema primero-en-profundidad*.

Páginas web del laboratorio que puede ayudarle:

## PARTE B

**Resumen:** Llamadas al sistema que debe revisar: *pipe()*, *dup()*, *dup2()*, *read()*, *write()*, *close()*.  
 Funciones de librería que debe revisar: *execl()*, *execv()*, *execle()*, *execlp()*, *execvp()*

1.- El programa *forknpipe.c* comunica un padre con su hijo, observe el dibujo.



(6.a) Un proceso después de invocar *pipe()*

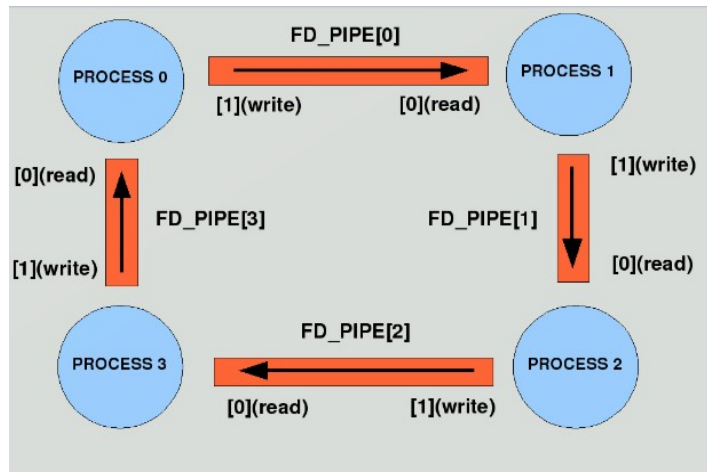
(6.b) El mismo proceso invoca *fork()*

(6.c) Cerrando los descriptores innecesarios

```

1  /* forknpipe.c (c) 2007-2009 Rahmat M. Samik-Ibrahim, GPL-like */
2  /* ***** */
3  #define BUFSIZE 64
4  #define WLOOP 5
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/types.h>
8  #include <sys/wait.h>
9  #include <unistd.h>
10 #include <string.h>
11
12 int main(void) {
13     char buffer[BUFSIZE];
14     char message[]="Hello, what's up?\n";
15     int ii, pipe_fd[2];
16     pipe(pipe_fd);
17     if (fork() == 0) { /* child ***** */
18         close(pipe_fd[0]);
19         printf("I am PID[%d] (child).\n", (int) getpid());
20         for (ii=0;ii<WLOOP;ii++)
21             write(pipe_fd[1], message, sizeof(message)-1);
22         close(pipe_fd[1]);
23     } else { /* parent ***** */
24         close(pipe_fd[1]);
25         printf("I am PID[%d] (parent).\n", (int) getpid());
26         memset(buffer, 0, sizeof(buffer));
27         while ((ii=read(pipe_fd[0],
28             buffer, BUFSIZE-1)) != 0) {
29             printf("PARENT READ[%d]:\n%s\n", (int) ii, buffer);
30             memset(buffer, 0, sizeof(buffer));
31         }
32         close(pipe_fd[0]);
33     }
34     return 0;
35 }
    
```

7.- El programa *forknpipe2.c* comunica 4 procesos.



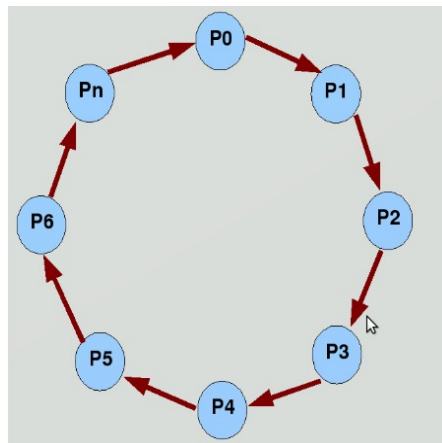
```

1  /* forknpipe2.c (c) 2007-2009 Rahmat M. Samik-Ibrahim, GPL-like */
2  #define BUFSIZE 64
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7  #include <unistd.h>
8  #include <string.h>
9
10 int main(void){
11     char buffer1[BUFSIZE],buffer2[BUFSIZE];
12     int p_this, p_prev, p_no1, p_no2;
13     int fd_pipe[4][2], ii, jj;
14     pid_t mypid;
15
16     memset(buffer1, 0, BUFSIZE);
17     memset(buffer2, 0, BUFSIZE);
18
19     for (ii=0;ii<4;ii++){
20         pipe(fd_pipe[ii]);
21     }
22
23     ii = (fork() != 0) ? 0 : 2;
24     jj = (fork() != 0) ? 0 : 1;
25
26     p_this = ii + jj;
27     close(fd_pipe[p_this][0]);
28
29     p_prev = (p_this + 3) % 4;
30     close(fd_pipe[p_prev][1]);
31
32     p_no1 = (p_this + 1) % 4;
33     close(fd_pipe[p_no1][0]);
34     close(fd_pipe[p_no1][1]);
35
36     p_no2 = (p_this + 2) % 4;
37     close(fd_pipe[p_no2][0]);
38     close(fd_pipe[p_no2][1]);
39
40     mypid = getpid();
41     sprintf(buffer1," A message from PID[%d].\n", (int) mypid);
42     write(fd_pipe[p_this][1], buffer1, BUFSIZE-1);
43     close(fd_pipe[p_this][1]);
44
45     while ((read(fd_pipe[p_prev][0], buffer2, BUFSIZE-1)) != 0) {
46         waitpid(-1,NULL,0);
47         printf("PID[%d] IS WAITING:\n%s\n", (int) mypid, buffer2);
48     }
49     close(fd_pipe[p_prev][0]);

```



8.- Escriba un programa para formar un anillo de  $n$  procesos, de forma que un mensaje es enviado por  $P_0$  es transmitido a  $P_1$ ,  $P_1$  lo transmite a  $P_2$ , y así sucesivamente. Al final  $P_0$  vuelve a recibir el mensaje.



### Solución

Programa tomado del libro *UNIX SYSTEMS Programming*, escrito por *Key A. Robbins* y *Steve Robbins* donde se explica el programa que ha continuación se presenta.

```

1  /******
2  /*      Programa anillo.c
3  /*      Tomado del libro: UNIX SYSTEMS Programming
4  /*      Autores : Kay A. Robbins y Steven Robbins
5  /******
6  #include <errno.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <unistd.h>
11 int main(int argc, char *argv[]) {
12     pid_t childpid; /* indicates process should spawn another */
13     int error; /* return value from dup2 call */
14     int fd[2]; /* file descriptors returned by pipe */
15     int i; /* number of this process (starting with 1) */
16     int nprocs; /* total number of processes in ring */
17     /* check command line for a valid number of processes to generate */
18     if ( (argc != 2) || ((nprocs = atoi(argv[1])) <= 0) ) {
19         fprintf(stderr, "Usage: %s nprocs\n", argv[0]);
20         return 1;
21     }
22     if (pipe(fd) == -1) { /* connect std input to std output via a pipe */
23         perror("Failed to create starting pipe");
24         return 1;
25     }
26     if ((dup2(fd[0], STDIN_FILENO) == -1) ||
27         (dup2(fd[1], STDOUT_FILENO) == -1)) {
28         perror("Failed to connect pipe");
29         return 1;
30     }
31     if ((close(fd[0]) == -1) || (close(fd[1]) == -1)) {
32         perror("Failed to close extra descriptors");
33         return 1;
34     }
35     for (i = 1; i < nprocs; i++) { /* create the remaining processes */
36         if (pipe(fd) == -1) {
37             fprintf(stderr, "[%ld]:failed to create pipe %d: %s\n",
38                     (long)getpid(), i, strerror(errno));
39             return 1;
40         }
41         if ((childpid = fork()) == -1) {
42             fprintf(stderr, "[%ld]:failed to create child %d: %s\n",
43                     (long)getpid(), i, strerror(errno));
44             return 1;
45         }

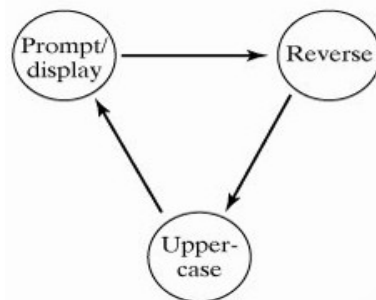
```

```

46     if (childpid > 0)                /* for parent process, reassign stdout */
47         error = dup2(fd[1], STDOUT_FILENO);
48     else                             /* for child process, reassign stdin */
49         error = dup2(fd[0], STDIN_FILENO);
50     if (error == -1) {
51         fprintf(stderr, "[%ld]:failed to dup pipes for iteration %d: %s\n",
52             (long)getpid(), i, strerror(errno));
53         return 1;
54     }
55     if ((close(fd[0]) == -1) || (close(fd[1]) == -1)) {
56         fprintf(stderr, "[%ld]:failed to close extra descriptors %d: %s\n",
57             (long)getpid(), i, strerror(errno));
58         return 1;
59     }
60     if (childpid)
61         break;
62 }                                     /* say hello to the world */
63 fprintf(stderr, "This is process %d with ID %ld and parent id %ld\n",
64     i, (long)getpid(), (long)getppid());
65 return 0;
66 }

```

9.- (Pregunta 2 del primer laboratorio calificado del semestre 2010-1) Escriba un programa que cree un anillo de 3 procesos conectados por *pipes*. El primer proceso deberá pedir al usuario que ingrese por teclado una cadena y enviarlo al segundo proceso. El segundo proceso deberá invertir la cadena y enviarla al tercer proceso. El tercer proceso deberá convertir los caracteres alfabéticos a mayúsculas y enviarlas de regreso al primer proceso. Cuando el primer proceso obtiene la cadena de procesos, deberá imprimirla en la terminal. Cuando se esto se lleve a cabo todos los procesos deberán terminar. A continuación una ilustración del anillo de procesos.



A continuación un ejemplo de la ejecución del programa.

```

$ ./ring3                ... corre el programa
  Por favor ingrese una cadena:ole
  La cadena procesada es: ELO
$ _

```

**Nota:** la cadena debe ser ingresada por teclado y tener como máximo 20 caracteres.

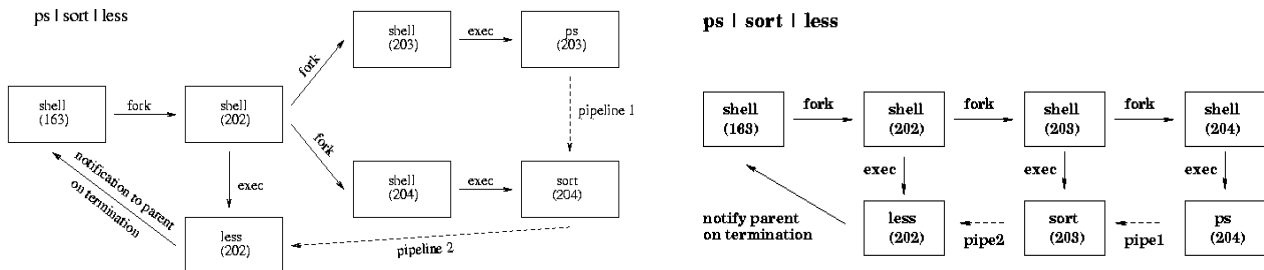
#### **Sugerencias:**

Para invertirla la cadena el segundo proceso debe leer, del primer *pipe*, toda la cadena en un arreglo, determinar la posición del último carácter de la cadena con la función *strlen()* y grabar por carácter en orden inverso en el segundo *pipe* (incluya el carácter cambio de línea). El tercer proceso lee carácter por carácter y lo convierte en mayúsculas con la función de librería *toupper()* y graba en el tercer *pipe* para que lo lea el proceso primero. Éste toma carácter por carácter y lo imprime, no haga cambio de línea en la impresión.

## Solución:

```
1 /* ring3.c (c) 2010 Alejandro T. Bello Ruiz, GPL-like */
2 /* ***** */
3
4 #include <stdio.h>
5 #include <unistd.h>
6 #include <ctype.h>
7 #include <stdlib.h>
8 #include <string.h>
9 #include <sys/wait.h>
10 #define N 3
11 #define PADRE 0
12 #define HIJO 1
13 #define NIETO 2
14
15 int pfd[N][2];
16 void cierra_descriptores_sin_uso(int);
17 void padre(void);
18 void hijo(void);
19 void nieto(void);
20
21 int main(void)
22 { int n;
23
24   for(n=0;n<N;n++) pipe(pfd[n]);
25   if(!fork()) {
26     if(!fork()) nieto();
27     else hijo();
28   }
29   padre();
30   return 0;
31 }
32
33 void padre(void)
34 { char c,cadena[20];
35   int i,n;
36
37   cierra_descriptores_sin_uso(PADRE);
38   printf("Ingrese una cadena:");
39   fgets(cadena, sizeof(cadena), stdin);
40   write(pfd[0][1], cadena, sizeof(cadena));
41   wait(NULL);
42   printf("La cadena original es : %s", cadena);
43   read(pfd[2][0], &n, sizeof(n));
44   printf("La cadena invertida es :");
45
46   for(i=0; i < n; i++) {
47     read(pfd[2][0], &c, sizeof(c));
48     printf("%c", c);
49   }
50
51 void hijo(void)
52 { char cadena[20];
53   int i,n;
54
55   cierra_descriptores_sin_uso(HIJO);
56   read(pfd[0][0], cadena, sizeof(cadena));
57   n=strlen(cadena);
58   write(pfd[1][1], &n, sizeof(n));
59   for(i=n-2; i >= 0; i--) write(pfd[1][1], &cadena[i], 1);
60   write(pfd[1][1], &cadena[n-1], 1);
61   wait(NULL);
62   exit(0);
63 }
64
65 void nieto(void)
66 { int i,k; char c;
67
68   cierra_descriptores_sin_uso(NIETO);
69   read(pfd[1][0], &k, sizeof(k));
70   write(pfd[2][1], &k, sizeof(k));
71   for(i=0; i < k; i++) {
72     read(pfd[1][0], &c, sizeof(c));
73     if(isalpha(c)) c=toupper(c);
74     write(pfd[2][1], &c, sizeof(c));
75   }
76   exit(0);
77 }
78
79
80 void cierra_descriptores_sin_uso(int i)
81 { int prev,ambos,sgte;
82
83   prev = (i - 1) < 0 ? N - 1 : i - 1; ambos = (i + 1) % N; sgte = i;
84   close(pfd[prev][1]);
85   close(pfd[ambos][0]);
86   close(pfd[ambos][1]);
87   close(pfd[sgte][0]);
88 }
```

10.- (Pregunta 5 del examen parcial del 2008-1) (Sandra Mamrak & Shaun Rowland) El mandato `ps | sort | less` para obtener la lista de procesos, ordenarla y paginarla interactivamente con avance o retroceso, requiere la construcción de dos tuberías (*pipes*) que puede implementarse por *shell* de varias formas. En las siguientes figuras se presentan dos formas diferentes. Presente los bosquejos solamente (no los programas completos, tampoco se necesita declarar todas las variables) que usen *fork()*, *exec()*, *pipe()*, *dup2()*, *close()* y *waitpid()* para estas dos implementaciones de arquitecturas de procesos. Preste una atención especial al cierre de todos los descriptores de tuberías y explique cómo cada proceso sabrá el momento de acabar su ejecución. Para los mandatos de *shell* que exigen la construcción de 3, 4 o más tuberías se observará que la segunda forma de implementación es más ventajosa. ¿Por qué?



### Solución:

Aunque la pregunta solicitaba un bosquejo, a continuación se ha implementado el programa para el primer caso.

```

1 /* other_ring3.c (c) 2009 Alejandro T. Bello Ruiz, GPL-like */
2 /* **** */
3 #include <stdio.h>
4 #include <unistd.h>
5 #include <sys/wait.h>
6
7 #define READ_STD 0
8 #define WRITE_STD 1
9
10 int main(int narg, char *argv[])
11 { int pfd1[2], pfd2[2];
12   int pid;
13
14   if(fork()) wait((int *)0);
15   else {
16     pipe(pfd1);
17     pipe(pfd2);
18     if((pid = fork()) && fork()) {
19       close(pfd1[0]);
20       close(pfd1[1]);
21       close(READ_STD);
22       dup(pfd2[0]);
23       close(pfd2[0]);
24       close(pfd2[1]);
25       execlp(argv[3], argv[3], NULL);
26     }
27     close(pfd2[0]);
28     switch (pid) {
29       case 0: close(WRITE_STD);
30              dup(pfd1[1]);
31              close(pfd1[0]);
32              close(pfd1[1]);
33              close(pfd2[1]);
34              execlp(argv[1], argv[1], NULL);
35       default: close(READ_STD);
36              dup(pfd1[0]);
37              close(WRITE_STD);
38              dup(pfd2[1]);
39              close(pfd1[0]);
40              close(pfd1[1]);
41              close(pfd2[1]);
42              execlp(argv[2], argv[2], NULL);
43     }
44   }
45   return 0;
46 }
47
-- INSERTAR --

```

11.- Elabore un programa que implemente el segundo esquema.

Páginas web del laboratorio que puede ayudarle:

<http://inform.pucp.edu.pe/~inf232/Semestre-1999-2/Laboratorio1/index.htm>  
<http://inform.pucp.edu.pe/~inf232/Semestre-2000-1/Laboratorio-2/index.htm>  
<http://inform.pucp.edu.pe/~inf232/Semestre-2000-2/Laboratorio-1/index.htm>  
<http://inform.pucp.edu.pe/~inf232/Semestre-2001-1/Laboratorio-3/index.htm>  
<http://inform.pucp.edu.pe/~inf232/Semestre-2002-1/Laboratorio-3/index.htm>  
<http://inform.pucp.edu.pe/~inf232/Semestre-2003-1/Laboratorio-3/index.htm>  
<http://inform.pucp.edu.pe/~inf232/Semestre-2005-1/Laboratorio-3/index.html>  
<http://inform.pucp.edu.pe/~inf232/Semestre-2007-2/Laboratorio-3/index.htm>  
<http://inform.pucp.edu.pe/~inf232/Semestre-2008-1/Laboratorio-3/index.htm>

### **Objetivos del laboratorio**

El laboratorio tiene los siguientes objetivos (estos serán los que marquen las pautas para las preguntas)

A.- Verificar que el alumno ha comprendido los programas que se presentaron como ejemplos o soluciones. Como no es posible evaluar todos los programas, se tomará solo uno de ellos.

B.- Comprobar que el alumno ha realizado las tareas dejadas en este material, aplicando el mismo procedimiento a problemas semejantes a los solicitados.

C.- Evaluar la capacidad de análisis y respuesta frente a situaciones nuevas que pueden ser solucionadas a partir de los ejemplos presentados en este material.

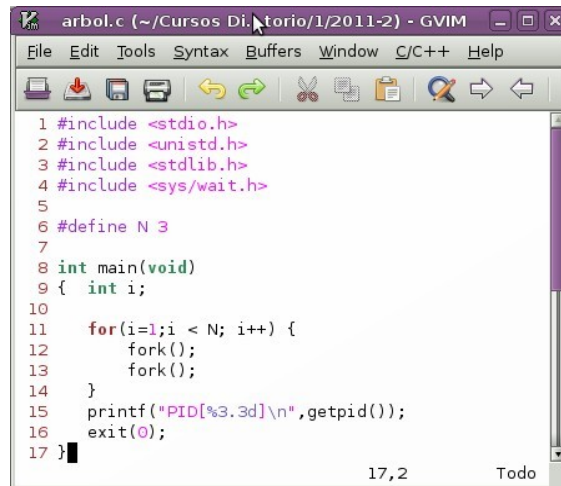
### **Importante**

No necesita traer código adicional. Si hubiera necesidad de escribir algún código base, este le será proporcionado en el laboratorio.

**Prof: Alejandro T. Bello Ruiz**

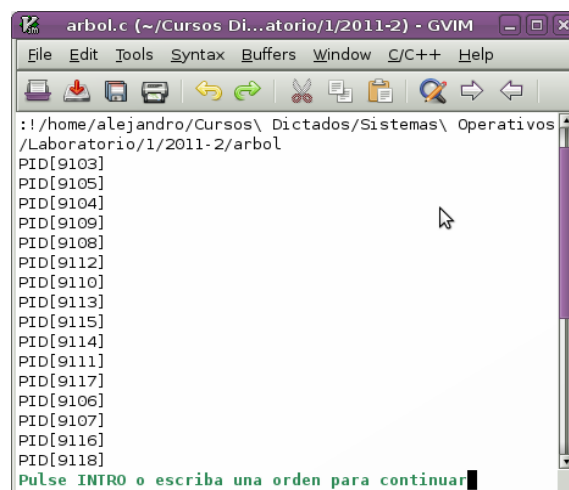
## Solución al Laboratorio Nro 1 (2do Período del 2011)

1.- (5 puntos) Se tiene el programa *arbol.c* (usted ha descargado junto con este documento una copia) que después de ser ejecutado muestra mensajes de los *pids* de todos los procesos creados incluyendo el identificador del padre.



```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <sys/wait.h>
5
6 #define N 3
7
8 int main(void)
9 { int i;
10
11   for(i=1; i < N; i++) {
12     fork();
13     fork();
14   }
15   printf("PID[%3.3d]\n", getpid());
16   exit(0);
17 }
```

Salida



```
::/home/alejandra/Cursos/Dictados/Sistemas/Operativos
/Laboratorio/1/2011-2/arbol
PID[9103]
PID[9105]
PID[9104]
PID[9109]
PID[9108]
PID[9112]
PID[9110]
PID[9113]
PID[9115]
PID[9114]
PID[9111]
PID[9117]
PID[9106]
PID[9107]
PID[9116]
PID[9118]
Pulse INTRO o escriba una orden para continuar
```

En total son 16 procesos. Se pide insertar líneas apropiadas en el programa, sin modificar en esencia el mismo, de manera que al ejecutarse, muestre el árbol completo de procesos.

Sugerencias:

a) Para no mostrar el árbol de procesos del sistema (desde *init*) puede emplear la siguiente línea:

```
pstree -p pid
```

donde *pid* indica el identificador del proceso padre cuyo árbol se quiere mostrar.

b) Para obtener una cadena a partir de datos dinámicos, puede emplear la función de librería *sprintf()*.

```
id = getpid(); /* id y cad variables previamente definidas */
sprintf(cad, "Mi pid es %d", id);
```

```
printf("%s\n",cad); /* Imprimirá algo como 'Mi pid es 1948' */
```

### Solución

El objetivo de esta pregunta es evaluar si el alumno es capaz de concluir que no se puede hacer presunción alguna de los procesos creados. Es decir no se puede asumir que la ejecución de procesos siguen siempre un orden determinado, esto es un mal supuesto. Debido a esto lo más seguro era dejar que los hijos se ejecutaran “for ever” y el padre después de un tiempo pequeño, un segundo quizás, pueda tomarles la foto, en la que definitivamente salieran todos.

A continuación el código solución y su salida.

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

#define N 3

int main(void)
{
    int i,ppid;
    char cadena[15];

    ppid = getpid();
    for(i=1; i < N; i++) {
        fork();
        fork();
    }
    printf("PID[%3.3d]\n",getpid());
    fflush(stdout);
    if (ppid == getpid()) {
        sleep(1);
        sprintf(cadena,"pstree -p %d",ppid);
        system(cadena);
        wait(NULL);
    } else {
        for(;;);
    }
    exit(0);
}
```

### Salida

```
:/home/alejandro/Cursos\ Dictados/Sistemas\ Operativos/Laboratorio/1/2011-2/arbol_sol
PID[7469]
PID[7470]
PID[7475]
PID[7476]
PID[7473]
PID[7474]
PID[7471]
PID[7472]
PID[7478]
PID[7477]
PID[7479]
PID[7480]
PID[7481]
PID[7482]
PID[7483]
PID[7484]
arbol_sol(7469)---arbol_sol(7470)---arbol_sol(7472)---arbol_sol(7481)---arbol_sol(7484)
|                                     |
|                                     |---arbol_sol(7482)
|                                     |---arbol_sol(7478)
|                                     |---arbol_sol(7474)
|                                     |---arbol_sol(7476)
|---arbol_sol(7471)---arbol_sol(7479)---arbol_sol(7483)
|                                     |---arbol_sol(7480)
|---arbol_sol(7473)---arbol_sol(7477)
|---arbol_sol(7475)
|---sh(7485)---pstree(7486)
```

Recuerde que para considerar el programa correcto, en el árbol deben aparecer los 16 procesos creados. Pueden existir diferentes programas.

**2.- (9 puntos)** Para el programa *forknpipe2.c*, del cuál usted ya ha hecho un análisis previo, se pide hacer un seguimiento del mismo, tal como se indican en el archivo *trace.odt* (archivo proporcionado con este documento).

```
1 /* forknpipe2.c (c) 2007-2009 Rahmat M. Samik-Ibrahim, GPL-like */
2 #define BUFSIZE 64
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/types.h>
6 #include <sys/wait.h>
7 #include <unistd.h>
8 #include <string.h>
9
10 int main(void){
11     char buffer1[BUFSIZE],buffer2[BUFSIZE];
12     int p_this, p_prev, p_nol, p_no2;
13     int fd_pipe[4][2], ii, jj;
14     pid_t mypid;
15
16     memset(buffer1, 0, BUFSIZE);
17     memset(buffer2, 0, BUFSIZE);
18
19     for (ii=0;ii<4;ii++){
20         pipe(fd_pipe[ii]);
21     }
22
23     ii = (fork() != 0) ? 0 : 2;
24     jj = (fork() != 0) ? 0 : 1;
25
26     p_this = ii + jj;
27     close(fd_pipe[p_this][0]);
28
29     p_prev = (p_this + 3) % 4;
30     close(fd_pipe[p_prev][1]);
31
32     p_nol = (p_this + 1) % 4;
33     close(fd_pipe[p_nol][0]);
34     close(fd_pipe[p_nol][1]);
35
36     p_no2 = (p_this + 2) % 4;
37     close(fd_pipe[p_no2][0]);
38     close(fd_pipe[p_no2][1]);
39
40     mypid = getpid();
41     sprintf(buffer1, " A message from PID[%d].\n", (int) mypid);
42     write(fd_pipe[p_this][1], buffer1, BUFSIZE-1);
43     close(fd_pipe[p_this][1]);
44
45     while ((read(fd_pipe[p_prev][0], buffer2, BUFSIZE-1)) != 0) {
46         waitpid(-1,NULL,0);
47         printf("PID[%d] IS WAITING:\n%s\n", (int) mypid, buffer2);
48     }
49     close(fd_pipe[p_prev][0]);
50     exit(0);
51 }
```

-- INSERTAR -- 51,25 Comienzo

### Solución

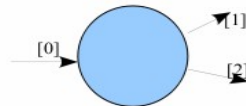
A diferencia de escribir programas en los que este depende de la lógica de cada alumno, en este ejercicio si hay respuesta única puesto que se hace análisis de un sólo código. El objetivo era hacer un “trace” al programa mostrado arriba. Se le presento el estado de procesos mediante unos bloques ya preparados para que el alumno continúe simplemente copiando los bloques según la secuencia de ejecución.



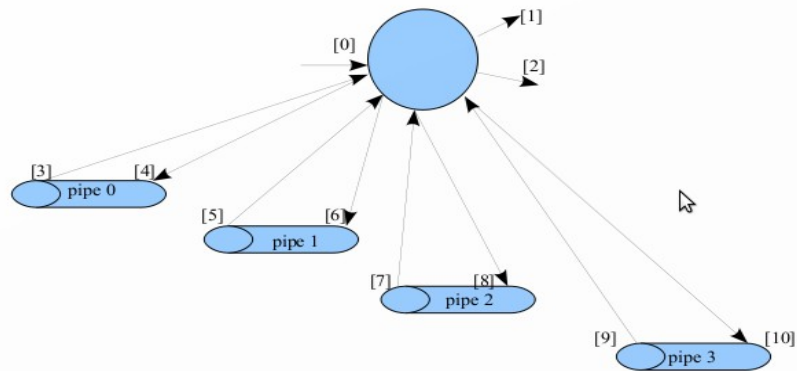
A continuación se muestra lo que se les presentó como base en el archivo *trace.odt*

(9 puntos) Trazo del programa *forknpipe2.c*

El proceso al inicio se puede representar con el siguiente gráfico:

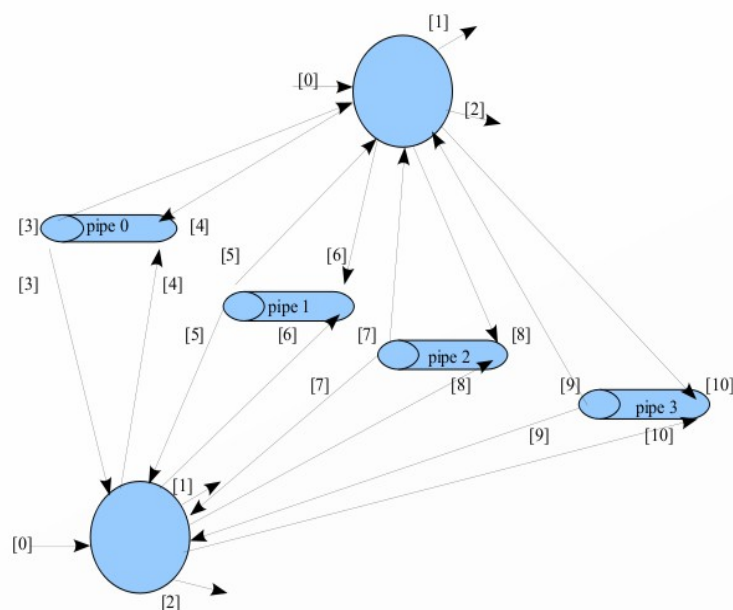


Luego hasta la línea 22 la ejecución se puede representar con el siguiente gráfico:



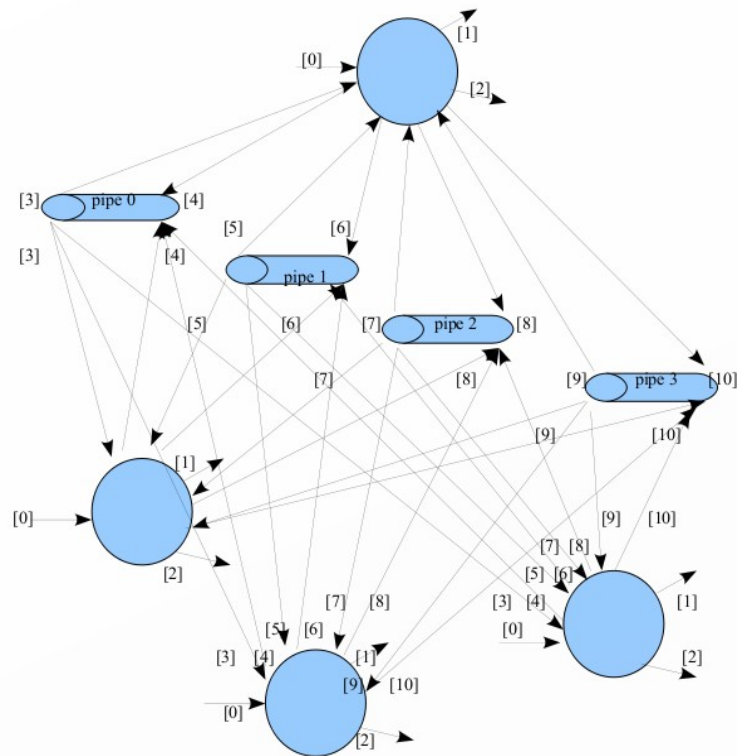
**Solución parte a)**

a) (2 punto) Luego de ejecutar el programa hasta la línea 23 se puede representar con el siguiente gráfico:



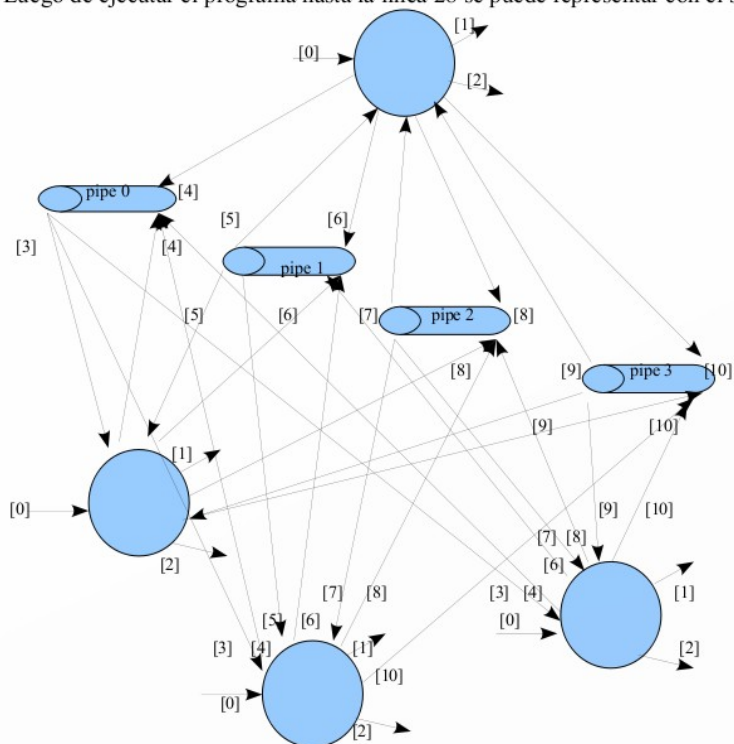
**Solución parte b)**

b) (3 puntos) Luego de ejecutar el programa hasta la línea 24 se puede representar con el siguiente gráfico:



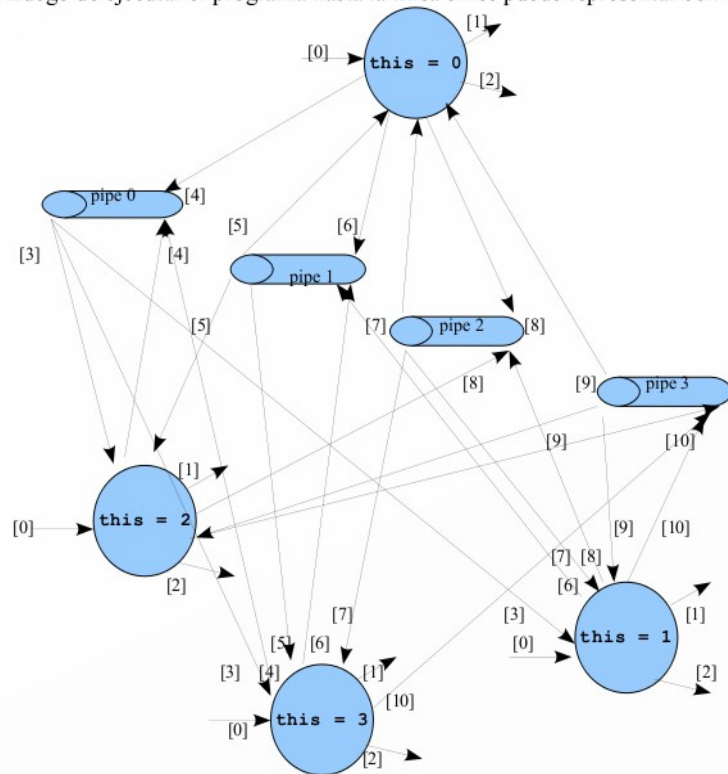
**Solución parte c)**

c) (1 punto) Luego de ejecutar el programa hasta la línea 28 se puede representar con el siguiente gráfico:



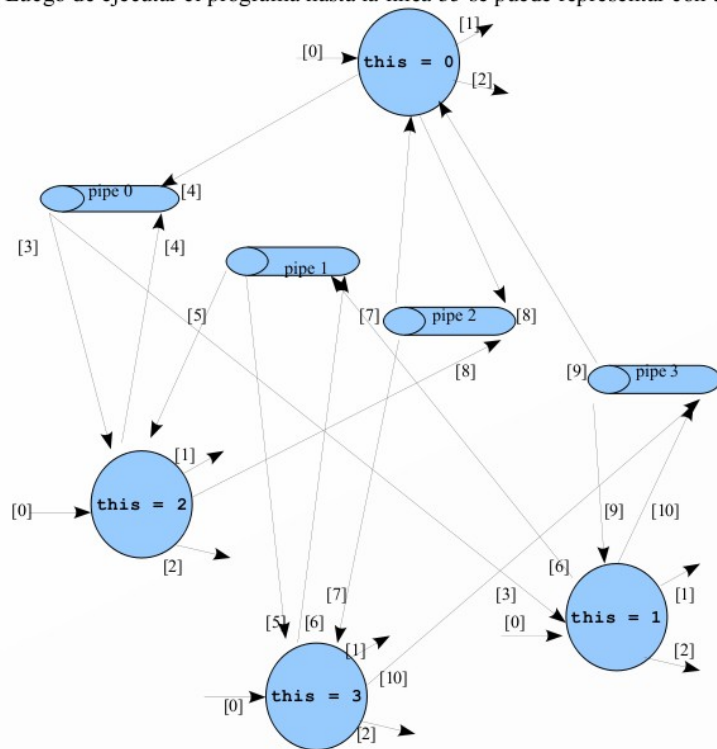
### Solución parte d)

d) (1 punto) Luego de ejecutar el programa hasta la línea 31 se puede representar con el siguiente gráfico:

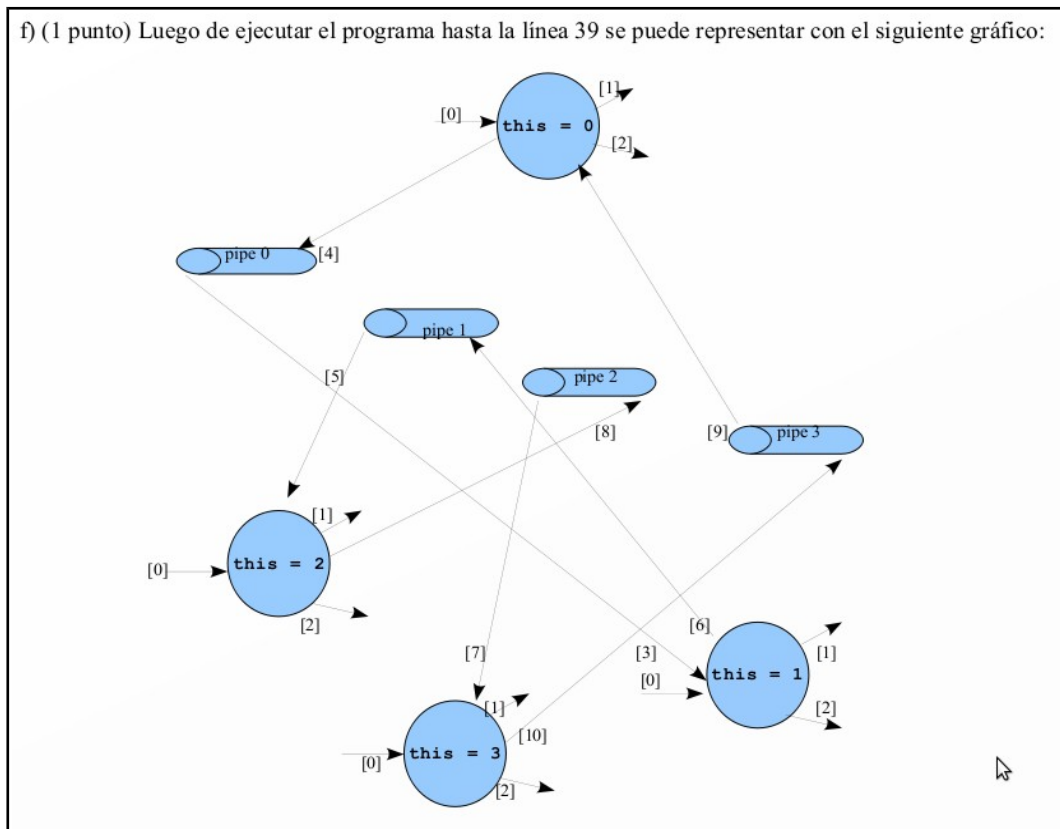


### Solución parte e)

e) (1 punto) Luego de ejecutar el programa hasta la línea 35 se puede representar con el siguiente gráfico:



### Solución parte f)



Las únicas partes difíciles son a) y b), el resto de diagramas son sencillas por que sólo hay que borrar descriptors que están representados por las flechas. Osea para c), d), e) y f) se hacían copias de la anterior y se procedía a borrar los descriptors respectivos. Al final se debe llegar a un anillo de 4 procesos, pero se debe tener un orden con los *pipes*.

**3.- (6 puntos)** Se desea escribir un programa con nombre *mybackup.c* que recibe como argumento por la línea comandos el nombre de un directorio. El programa debe mostrar al usuario los nombres de los archivos (regulares) que se encuentran en dicho directorio, preguntando si desea hacer una copia o no de cada uno de ellos. Si la respuesta es (Y/y/S/s) el programa debe copiar el archivo con el mismo nombre añadiéndole *.bak*, en caso contrario no se lleva a cabo acción alguna.

Se plantea la siguiente solución inconclusa. Mostrar los archivos regulares que se encuentran en el directorio indicado en `argv[1]`. Esto se logra con el comando *ls* ejecutado con la función de librería *system()*, y filtrado con *grep* para obtener sólo los archivos regulares. Esta estrategia tiene un inconveniente: la salida se mostrará por pantalla, todo en su conjunto. Lo ideal es que el programa tenga acceso a dicha lista y pueda mostrar al usuario el nombre de cada archivo para que responda si se desea hacer un *backup* o no de este.

A continuación se muestra el programa y su ejecución (el archivo fuente también es proporcionado con este documento).

```

mybackup.c + (~Cursos Dictados...tivos/Laboratorio/1/2011-2) - GVIM
File Edit Tools Syntax Buffers Window C/C++ Help
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void usage(char *);
6
7 int main(int nargs, char *argv[])
8 { char comando[30];
9
10 if (nargs != 2) usage(argv[0]);
11 sprintf(comando,"ls -l %s | grep ^-.",argv[1]);
12 system(comando);
13 return 0;
14 }
15
16 void usage(char *prog)
17 {
18 fprintf(stderr,"Uso: %s <ruta_del_directorio>\n",prog);
19 exit(1);
20 }
21
-- INSERTAR --
21,1 Todo

```

Con la opción <S-F9> puede ingresar el argumento de la línea de comando. Para este ejemplo se ingresa el directorio /home/alejandro/Documentos

```

mybackup.c (~Cursos Dictados/S...tivos/Laboratorio/1/2011-2) - GVIM
File Edit Tools Syntax Buffers Window C/C++ Help
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void usage(char *);
6
7 int main(int nargs, char *argv[])
8 { char comando[30];
9
10 if (nargs != 2) usage(argv[0]);
11 sprintf(comando,"ls -l %s | grep ^-.",argv[1]);
12 system(comando);
13 return 0;
14 }
15
16 void usage(char *prog)
17 {
18 fprintf(stderr,"Uso: %s <ruta_del_directorio>\n",prog);
19 exit(1);
20 }
21
command line arguments for "Cursos Dictados/Sistemas Operativos/Laboratorio/1/2011-2/mybackup" : /home/alejandro/Documentos

```

Escribir el directorio que se desea como argumento en la línea de comandos

Luego ejecute el programa con <C-F9>. Observe que en la salida se muestran solo archivos regulares.

```

mybackup.c (~Cursos Dictados/Sistemas Operativos/Laboratorio/1/2011-2) - GVIM
File Edit Tools Syntax Buffers Window C/C++ Help
17 {
18 fprintf(stderr,"Uso: %s <ruta_del_directorio>\n",prog);
19 exit(1);
20 }
21
~/
~/
~/
~/home/alejandro/Cursos\ Dictados\Sistemas\ Operativos\Laboratorio/1/2011-2/mybackup /home/alejandro/Documentos
-rw-r--r-- 1 alejandro alejandro 125645 sep  8 11:36 From logic programming to Prolog_Introduction_ - Krzysztof R. Apt.pdf
-rw-r--r-- 1 alejandro alejandro 56404 mar 18 09:19 fx3600p.odt
-rw-r--r-- 1 alejandro alejandro 3281851 jun 22 21:55 modulo_05-openoffice_base.pdf
-rw-r--r-- 1 alejandro alejandro 896 ago 19 2010 prog1.pl
-rw-r--r-- 1 alejandro alejandro 461 mar  7 2011 prog2.pl
-rw-r--r-- 1 alejandro alejandro 285 ago 19 2010 prog3.pl
-rw-r--r-- 1 alejandro alejandro 449 ago 19 2010 prog4.pl
-rw-r--r-- 1 alejandro alejandro 80 ago 19 2010 prog5.pl
-rw-r--r-- 1 alejandro alejandro 2541 ago 19 2010 prog6.pl
-rw-r--r-- 1 alejandro alejandro 2000186 nov  8 2001 S750_UG_ES.PDF
-rw-r--r-- 1 alejandro alejandro 187516 sep  8 11:47 Seven Languages in Seven Weeks_seleccion2.pdf
-rw-r--r-- 1 alejandro alejandro 268143 sep  8 11:47 Seven Languages in Seven Weeks_seleccion.pdf

La orden fue terminada
Pulse INTRO o escriba una orden para continuar

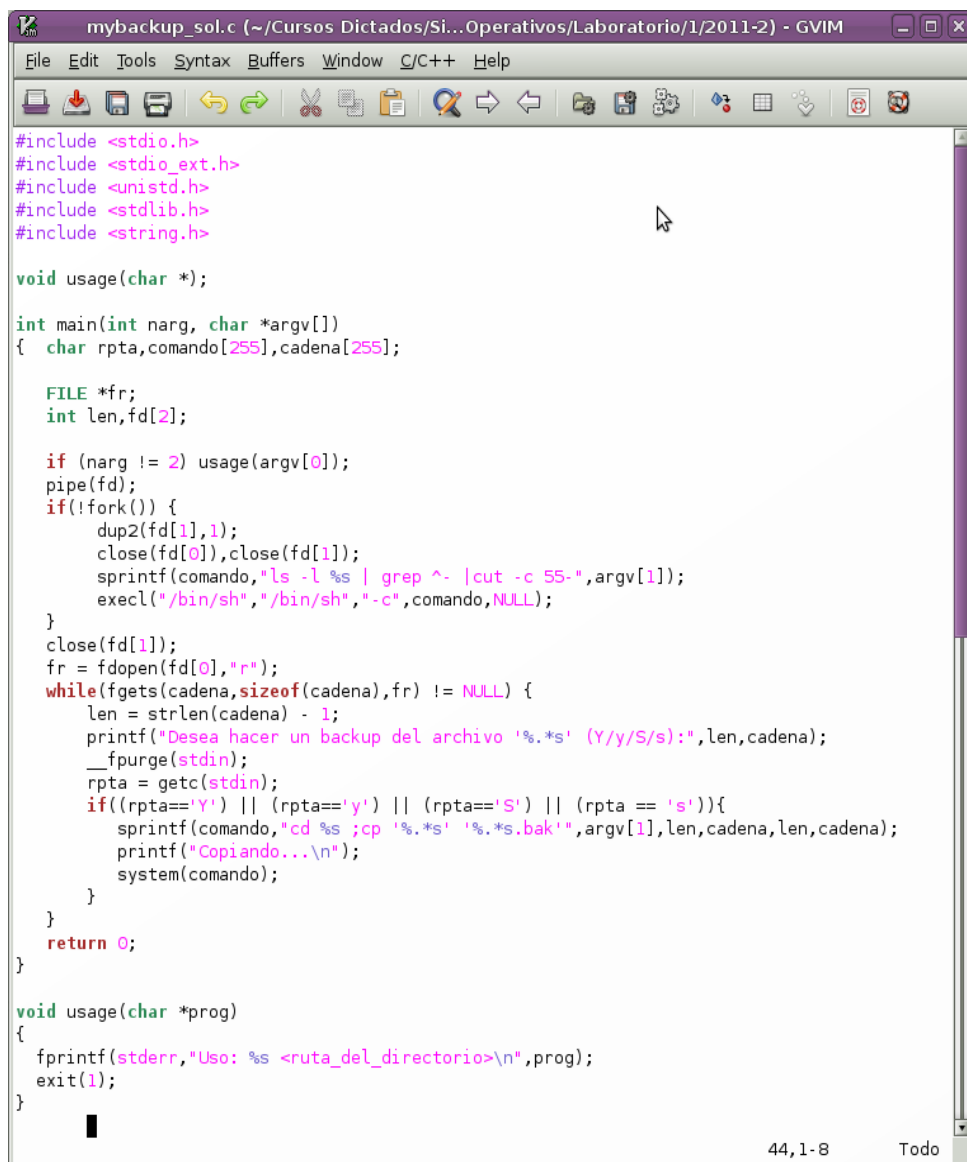
```

Se pide modificar el programa con la siguiente estrategia: el proceso padre crea un *pipe* y luego hace *fork()*. El proceso hijo redirecciona su salida estándar por el de escritura del *pipe*. De forma que todo lo que el hijo imprima a la salida estándar, vaya al *pipe*. A continuación el hijo invoca la función de librería *execv()* pidiendo que se ejecute un *shell* con la siguiente cadena como argumento `ls -l dir | grep ^-` y termina. El padre convierte el descriptor de lectura del *pipe* en un `FILE *` con la función *fdopen()* (la finalidad es facilitar la lectura de cadenas ya que *read()* lee flujo de bytes). Luego con *fgets()* se lee del *pipe* lo que el hijo escribió. A continuación muestra al usuario el nombre y si el usuario responde (Y/y/S/s) se se lleva a cabo la copia con *system()* invocando el comando *cp*. Este proceso se repite hasta que no haya nada que leer del *pipe*.

Puede consultar el manual en línea para los formatos de *fdopen()* y *fgets()*.

### Solución

#### A continuación el programa y su ejecución:



```
#include <stdio.h>
#include <stdio_ext.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

void usage(char *);

int main(int narg, char *argv[])
{
    char rpta, comando[255], cadena[255];

    FILE *fr;
    int len, fd[2];

    if (narg != 2) usage(argv[0]);
    pipe(fd);
    if (!fork()) {
        dup2(fd[1], 1);
        close(fd[0]), close(fd[1]);
        sprintf(comando, "ls -l %s | grep ^- | cut -c 55-", argv[1]);
        execl("/bin/sh", "/bin/sh", "-c", comando, NULL);
    }
    close(fd[1]);
    fr = fdopen(fd[0], "r");
    while (fgets(cadena, sizeof(cadena), fr) != NULL) {
        len = strlen(cadena) - 1;
        printf("Desea hacer un backup del archivo '%s' (Y/y/S/s):", len, cadena);
        __fpurge(stdin);
        rpta = getc(stdin);
        if ((rpta == 'Y') || (rpta == 'y') || (rpta == 'S') || (rpta == 's')) {
            sprintf(comando, "cd %s ; cp '%s' '%s.bak'", argv[1], len, cadena, len, cadena);
            printf("Copiando...\n");
            system(comando);
        }
    }
    return 0;
}

void usage(char *prog)
{
    fprintf(stderr, "Uso: %s <ruta_del_directorio>\n", prog);
    exit(1);
}
```

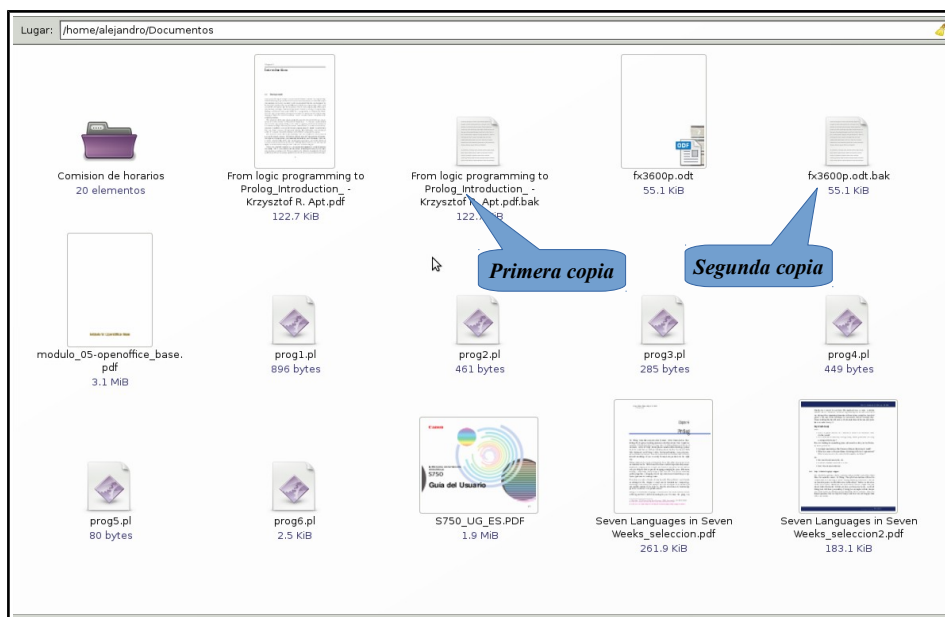
### Antes de mostrar la ejecución algunos comentarios.

- La línea `"ls -l %s | grep ^- | cut -c 55-"` fue puesta en la pizarra durante el laboratorio, para facilitar la obtención de los nombres de los archivos regulares en el *pipe*.
- Hubiera sido mejor controlar el comando `exec1()` si retornase `< 0`, en caso por ejemplo que la ruta del directorio no existiera. Esto no está controlado en mi solución.
- Hay dos exquisiteces que no hay que tener en cuenta, primero es muy conocido el problema de ingreso de caracteres con la función `getc(stdin)` ó `getchar()` por el retorno de carro que se almacena en la entrada estándar. La función `fflush()` solo limpia la salida estándar, por ese motivo empleo la función `__fpurge(stdin)`. Otra forma hubiera sido hacer un `while(getchar()=='\n');` pero no se veía muy elegante, así que emplee `__fpurge()`. La segunda tiene que ver con armar el comando para la copia, hay que recordar primero que se debe de encontrar en el directorio donde se encuentra los archivos, por eso primero se debe hacer un `cd` y luego ejecutar el comando `cp`. Para este comando se emplea los apóstrofes (') para poder copiar nombres de archivos conteniendo caracteres especiales y espacios en blanco.

Antes de ejecutar el programa se ha pasado como argumento por la línea de comandos el directorio `/home/alejandro/Documentos`

```
:/home/alejandro/Cursos\ Dictados/Sistemas\ Operativos/Laboratorio/1/2011-2/mybackup_sol /h
ome/alejandro/Documentos
Desea hacer un backup del archivo 'From logic programming to Prolog_Introduction_ - Krzyszto
f R. Apt.pdf' (Y/y/S/s):Y
Copiando...
Desea hacer un backup del archivo 'fx3600p.odt' (Y/y/S/s):Y
Copiando...
Desea hacer un backup del archivo 'modulo_05-openoffice_base.pdf' (Y/y/S/s):n
Desea hacer un backup del archivo 'prog1.pl' (Y/y/S/s):n
Desea hacer un backup del archivo 'prog2.pl' (Y/y/S/s):n
Desea hacer un backup del archivo 'prog3.pl' (Y/y/S/s):n
Desea hacer un backup del archivo 'prog4.pl' (Y/y/S/s):n
Desea hacer un backup del archivo 'prog5.pl' (Y/y/S/s):n
Desea hacer un backup del archivo 'prog6.pl' (Y/y/S/s):n
Desea hacer un backup del archivo 'S750_UG_ES.PDF' (Y/y/S/s):n
Desea hacer un backup del archivo 'Seven Languages in Seven Weeks_seleccion2.pdf' (Y/y/S/s):
n
Desea hacer un backup del archivo 'Seven Languages in Seven Weeks_seleccion.pdf' (Y/y/S/s):n

Pulse INTRO o escriba una orden para continuar
```



Prof. Alejandro T. Bello Ruiz



**SISTEMAS OPERATIVOS**  
**Laboratorio Nro 1 (Solución)**  
**(Segundo Semestre del 2015)**



**1.- (4 puntos) Orphan process:** An *orphan process* is a computer process whose parent process has finished or terminated, though it remains running itself. In a Unix-like operating system any orphaned process will be immediately adopted by the special **init** system process. This operation is called *re-parenting* and occurs automatically. Even though technically the process has the "init" process as its parent, it is still called an orphan process since the process that originally created it no longer exists. (*From Wikipedia*).

Escriba un programa con nombre *huerfano.c* que cree un proceso hijo, que luego quedará huérfano. El proceso hijo en su código deberá ejecutar *pstree -spl* antes de quedar huérfano y después, cuando ya es adoptado por "init". La salida deberá ser semejante a la siguiente:

```
alejandro@abMint ~/Documentos $ ./huerfano > salida
alejandro@abMint ~/Documentos $ cat salida
init(1)---mdm(1010)---mdm(1155)---cinnamon-session(2292)---cinnamon-launch(2796)---cinnamon(2798)---gnome-terminal(11611)---bash(11618)---huerfano(21834)---huerfano(21835)---sh(21836)---pstree(21837)
init(1)---huerfano(21835)---sh(21838)---pstree(21839)
alejandro@abMint ~/Documentos $
```

**Solución**

Un proceso huérfano, es aquel proceso que estando aún ejecutándose, su padre ha muerto. Y que posteriormente será "adoptado" por **init**.

**Primer intento**

Una solución sencilla consistiría en escribir un programa que, primero cree un proceso con *fork()*, luego el proceso hijo ejecutaría la orden *pstree*, el proceso hijo esperaría (con un *sleep*) a que el proceso padre muera y en ese momento se ejecutaría por segunda vez la orden *pstree*. A continuación el código:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#define OK 0

int main(void) {
    int id;

    if(!fork()) { /* Hijo */
        char cad[20];

        id = getpid();
        sprintf(cad, "pstree -spl %d", id);
        system(cad);
        sleep(1);
        system(cad);
        exit(OK);
    }
    /* Padre */
    exit(OK);
}
```

Al ejecutar este código no devuelve el resultado esperado:

```
alejandro@abMint ~/Documentos $ ./intento1
alejandro@abMint ~/Documentos $ init(1)---intento1(31713)---sh(31714)---pstree(31715)
init(1)---intento1(31713)---sh(31716)---pstree(31717)
```

La explicación es muy sencilla. El proceso padre se ha ejecutado primero (después de la llamada *fork()*) por ese motivo el proceso hijo cuando ejecutó ambos *pstree* ya no encontró al proceso padre.



## Segundo intento

¿Qué sucede si se añade un *sleep* al proceso padre?

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#define OK 0

int main(void) {
    int id;

    if(!fork()) { /* Hijo */
        char cad[20];

        id = getpid();
        sprintf(cad, "pstree -spl %d", id);
        system(cad);
        sleep(1);
        system(cad);
        exit(OK);
    }
    /* Padre */
    sleep(1);
    exit(OK);
}
```

Observemos su salida:

```
alejandro@abMint ~/Documentos $ ./intento2
init(1)—gnome-terminal(28436)—bash(28444)—intento2(31827)—intento2(31828)—sh(31829)—pstree(31830)
alejandro@abMint ~/Documentos $ init(1)—intento2(31828)—sh(31831)—pstree(31832)
```

Parece que funciona, sin embargo si esta vez el proceso hijo se ejecuta primero, tendríamos que en la salida de *pstree* el proceso padre siempre aparece. Colocar *sleeps* no soluciona el problema de fondo, pues aún este persistiría. Es decir esta solución aún cuando muestra lo que se esperaba, en otras circunstancias, no mostraría lo deseado. Por lo que esta solución no es completamente correcta.

## La solución

Una solución independiente del planificador es la siguiente: creamos un *pipe* y hacemos que el proceso padre lea de él, con el fin de que se bloquee al leer un *pipe* vacío. El proceso hijo ejecuta *pstree*, luego mata a su padre y vuelve a ejecutar *pstree*. Esto puede sonar un poco chocante pero con seguridad siempre funcionará, sin asumir premisa alguna. A continuación el programa:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <signal.h>
6  #define OK 0
7
8  int main(void) {
9      int fd[2], id1;
10
11     id1 = getpid();
12     pipe(fd);
13     if(!fork()) { /* Hijo */
14         int id2;
15         char cad[20];
16
17         id2 = getpid();
18         sprintf(cad, "pstree -spl %d", id2);
19         system(cad);
20         kill(id1, SIGTERM);
21         system(cad);
22         exit(OK);
23     }
24     /* Padre */
25     read(fd[0], &id1, sizeof(id1));
26     exit(OK);
27 }
28
```

A continuación su ejecución

```
alejandro@abMint ~/Documentos $ ./huerfano
init(1)—gnome-terminal(28436)—bash(28444)—huerfano(32027)—huerfano(32028)—sh(32029)—pstree(32030)
Terminado
alejandro@abMint ~/Documentos $ init(1)—huerfano(32028)—sh(32031)—pstree(32032)
```

**2.- (4 puntos) Zombie process:** On Unix and Unix-like computer operating systems, a **zombie process** or **defunct process** is a process that has completed execution (via the `exit` system call) but still has an entry in the process table: it is a process in the "Terminated state". This occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status: once the exit status is read via the `wait` system call, the zombie's entry is removed from the process table and it is said to be "reaped". A child process always first becomes a zombie before being removed from the resource table. In most cases, under normal system operation zombies are immediately waited on by their parent and then reaped by the system – processes that stay zombies for a long time are generally an error and cause a resource leak. (*From Wikipedia*)



Escriba un programa con nombre **zombie.c** que cree un proceso hijo que luego se convertirá en proceso **zombie**. El proceso hijo en su código deberá ejecutar **ps -l** antes de convertirse en **zombie**, y después cuando ya es proceso **zombie**. La salida deberá ser semejante a la siguiente:

```
Terminal
alejandro@abMint ~/Documentos $ ./zombie
F S  UID  PID  PPID  C  PRI  NI ADDR SZ WCHAN  TTY  TIME CMD
0 S  1000 11618 11611 0  80   0 - 6334 wait  pts/2  00:00:00 bash
0 S  1000 21780 11618 0  80   0 - 1049 wait  pts/2  00:00:00 zombie
1 S  1000 21781 21780 0  80   0 - 1049 pipe_w pts/2  00:00:00 zombie
0 S  1000 21782 21780 0  80   0 - 1112 wait  pts/2  00:00:00 sh
0 R  1000 21783 21782 0  80   0 - 3149 -      pts/2  00:00:00 ps
F S  UID  PID  PPID  C  PRI  NI ADDR SZ WCHAN  TTY  TIME CMD
0 S  1000 11618 11611 0  80   0 - 6334 wait  pts/2  00:00:00 bash
0 S  1000 21780 11618 0  80   0 - 1049 wait  pts/2  00:00:00 zombie
1 Z  1000 21781 21780 0  80   0 - 0 exit  pts/2  00:00:00 zombie <defunct>
0 S  1000 21784 21780 0  80   0 - 1112 wait  pts/2  00:00:00 sh
0 R  1000 21785 21784 0  80   0 - 3149 -      pts/2  00:00:00 ps
alejandro@abMint ~/Documentos $
```

## Solución

En un sistema UNIX-like los procesos existentes corresponden a un árbol jerárquico, donde cada padre tiene la "responsabilidad" de limpiar la entrada en la tabla de procesos correspondiente a su hijo. Esto normalmente lo lleva a cabo mediante la llamada al sistema `wait()`. Si un proceso hijo ha terminado y el padre no lo ha esperado con `wait`, mientras siga vivo el proceso padre, su hijo (que ha terminado) aparecerá como **zombie**. Una vez que el proceso padre muere, el proceso que lo generó (¿proceso abuelo?) limpiará las entradas tanto del padre como la del hijo. O debemos decir procesos hijo y nieto.

Teniendo en cuenta lo discutido en la pregunta anterior, la estrategia será semejante. Esta vez el proceso hijo se quedará bloqueado leyendo un pipe vacío, en caso de que llegue primero. El proceso padre ejecuta `ps`, mata al hijo y vuelve a ejecutar `ps`. Bajo las condiciones descritas el proceso hijo deberá aparecer como proceso **zombie**.

A continuación el programa y su la salida de su ejecución:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <signal.h>
6
7  #define OK 0
8  int main(void) {
9      int fd[2],id;
10     pipe(fd);
11     if((id=fork())) { /* Padre */
12         system("ps -l");
13         kill(id,SIGTERM);
14         system("ps -l");
15         exit(OK);
16     }
17     /* Hijo */
18     read(fd[0],&id,sizeof(id));
19     exit(OK);
20 }
21
```

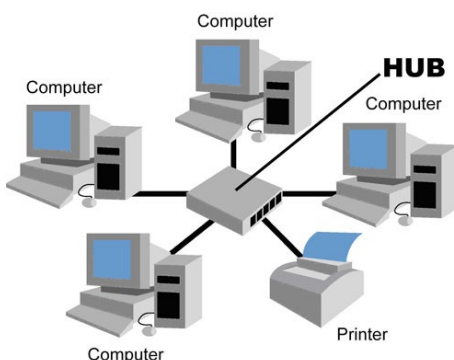
```
alejandro@abMint ~/Documentos $ ./zombie
F S  UID  PID  PPID  C  PRI  NI ADDR SZ WCHAN  TTY  TIME CMD
0 S  1000 8743 28444 0  80   0 - 1049 wait  pts/2  00:00:00 zombie
1 S  1000 8744 8743 0  80   0 - 1049 pipe_w pts/2  00:00:00 zombie
0 S  1000 8745 8743 0  80   0 - 1112 wait  pts/2  00:00:00 sh
0 R  1000 8746 8745 0  80   0 - 3149 -      pts/2  00:00:00 ps
0 S  1000 28444 28436 0  80   0 - 6376 wait  pts/2  00:00:00 bash
F S  UID  PID  PPID  C  PRI  NI ADDR SZ WCHAN  TTY  TIME CMD
0 S  1000 8743 28444 0  80   0 - 1049 wait  pts/2  00:00:00 zombie
1 Z  1000 8744 8743 0  80   0 - 0 exit  pts/2  00:00:00 zombie <defunct>
0 S  1000 8747 8743 0  80   0 - 1112 wait  pts/2  00:00:00 sh
0 R  1000 8748 8747 0  80   0 - 3149 -      pts/2  00:00:00 ps
0 S  1000 28444 28436 0  80   0 - 6376 wait  pts/2  00:00:00 bash
alejandro@abMint ~/Documentos $
```

The diagram shows a search tree. The root node is at the top. It has six children. The first child has one child. The second child has one child. The third child has one child. The fourth child has one child. The fifth child has one child. The sixth child has one child. This structure represents a search process where a single path is followed from the root to a leaf node.

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  #define N 40
8  void hijo(int,int *);
9  void error(char **);
10
11 int main(int n, char *argv[]) {
12     int x,z,np,fd[2],pids[N],total;
13     char cmd[15];
14     if(n!=2) error(argv);
15     pipe(fd);
16     np = atoi(argv[1]);
17     sprintf(cmd,"pstree -lc %d",getpid());
18     for(x=0;x< np;x++)
19         if(fork()==0) hijo(x,fd);
20     total = (1+np)*np/2;
21     for(z=0;z<total;z++) read(fd[0],&pids[z],sizeof(int));
22     system(cmd);
23     for(z=0;z<total;z++) kill(pids[z],SIGTERM);
24     exit(0);
25 }
26
27 void hijo(int x,int *fd) {
28     int y,mypid;
29     for(y=0;y<x;y++)
30         if(fork(>0) break;
31     mypid = getpid();
32     write(fd[1],&mypid,sizeof(int));
33     for(;;);
34 }
35
36 void error(char *argv[]) {
37     printf("Usage: %s <nrprocesses>",argv[0]);
38     exit(0);
39 }
40

```



4.- (4 puntos) **Star networks** are one of the most common computer network topologies. In its simplest form, a star network consists of one central switch, hub or computer, which acts as a conduit to transmit messages. This consists of a central node, to which all other nodes are connected; this central node provides a common connection point for all nodes through a hub. In star topology, every node (computer workstation or any other peripheral) is connected to a central node called a hub or switch. The switch is the server and the peripherals are the clients. Thus, the hub and leaf nodes, and the transmission lines between them, form a graph with the topology of a star. If the central node is *passive*, the originating node must be able to tolerate the reception of an echo of its own transmission, delayed by the two-way transmission time (i.e. to and

from the central node) plus any delay generated in the central node. An *active* star network has an active central node that usually has the means to prevent echo-related problems.

Escriba un programa que simule una topología estrella. Recuerde que cada nodo sólo puede comunicarse con el nodo central. Ningún nodo puede comunicarse con otro que no se el central. Esto es una característica importante en la topología. Los nodos serán representados por procesos mientras que las comunicaciones serán representadas por los *pipes*.

### Solución

Si se observa con detenimiento la forma que tienen esta topología es la de un abanico de procesos. Sin embargo si se crean los *pipes*, justo antes de crear cada procesos. Resultará que el proceso 1, tendrá acceso al *pipe* 1, el proceso 2 tendrá acceso a los *pipes* 1 y 2. el proceso 3, tendrá acceso a los *pipes* 1, 2 y 3. Y así sucesivamente. Para solucionar este problema, pues según el esquema cada proceso debe comunicarse con el proceso padre es a través de un único *pipe*. Lo único que se debe hacer es que cada proceso hijo debe eliminar los descriptores de los *pipes* anteriores a su número de orden. Esto significa que hay que crear un arreglo para los descriptores de los *pipes*. A continuación el programa:

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #define N 5
5
6  int fd[N][2];
7  void hijo(int);
8
9  int main(void) {
10     int x;
11     for(x=0; x<N; x++) {
12         pipe(fd[x]);
13         if(!fork()) hijo(x);
14     }
15     exit(0);
16 }
17
18 void hijo(int x) {
19     int y;
20     for(y=0; y<x; y++) {
21         close(fd[y][0]);
22         close(fd[y][1]);
23     }
24     exit(0);
25 }
26

```

Pando, 4 de setiembre de 2015.

*Prof: Alejandro T. Bello Ruiz*