Teoria – Tema 1 y 2:

1
Enumere 4 oraciones que definan qué es un proceso.
Programa en ejecución.
Instancia de un programa ejecutado en un computador.
"Espíritu animado" de un programa.
Entidad asignada a una CPU y ejecutada por ésta.
Unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual y un conjunto de recursos del sistema asociados.

1. ¿Cómo pueden tratarse múltiples interrupciones?

Pueden ocurrir n interrupciones al mismo tiempo.
Solución1: Inhabilitar interrupciones mientras se trata una, las que ocurran quedan pendientes y se tratarán luego de que termine la interrupcion actual. Solución elegante, pero no tiene en cuenta las prioridades o criticidad de las interrupciones (por ejemplo, las comunicaciones, que deben ser atendidas rápidamente para hacer lugar a la recepción de nuevos datos y evitar pérdida.
Solución 2: Manejar prioridades; una interrupción de prioridad más alta puede interrumpir a una de más baja prioridad.

2. Describa brevemente las tres técnicas para llevar a cabo operaciones de I/O.

-E-S programada: el módulo de E-S no lanza interrupciones sobre la CPU, sino que la CPU chequea periódicamente si se ha completado la operación de E-S para transferir los datos hacia/desde la memoria principal. Consume mucho tiempo (espera activa), ocupa la CPU en forma innecesaria.
-E-S dirigida por interrupciones: la CPU envía señal de E-S al módulo de E-S correspondiente y continúa con otra tarea. El módulo de E-S (en paralelo) realiza la operación de E-S y cuando ésta esta lista, envía la señal de interrupción a la CPU; ésta interrumpe -temporariamente- la ejecución del proceso actual, realiza la transferencia de datos y reanuda la ejecución del proceso interrumpido. Elimina espera innecesaria de la CPU, cada palabra a transferir del módulo E-S a la memoria principal debe pasar por la CPU.
-DMA: DMA puede ser un módulo separado, implementado sobre el bus del sistema o bien puede estar incorporado dentro de un módulo de E-S. Cuando la CPU necesita realizar una operación de E-S, señala al DMA indicando:

-Tipo de operación (read / write)
-Dirección del dispositivo de E-S
-Dirección inicial de memoria principal
-Número de palabras a escribir / leer

la CPU continúa con otra tarea. A partir de allí (en paralelo) el módulo DMA hace la operación de E-S en forma independiente, transfiere los datos y cuando la operación ha finalizado, envía señal de interrupción a la CPU. De esta forma, los datos no pasan a través de la CPU para ser transferidos desde/hacia la memoria principal.

1. Teniendo en cuenta el grado de concurrencia entre los procesos que permiten las estrategias para el tratamiento del interbloqueo ¿Cuál es la estrategia más conservadora, la menos conservadora y la más moderada?

La más conservadora es la Prevención, la más moderada es la Predicción y la menos conservadora es la Detección.

2. ¿Cuáles son las estrategias utilizadas para el tratamiento del problema del interbloqueo? ¿Qué tratan de lograr estas estrategias?

Las estrategias tratan de encontrar un camino seguro en la asignación de recursos a procesos, evitando, de esta forma, el interbloqueo.

Las estrategias son: Prevención, Predicción, Detección.

Observaciones en cuanto al uso de señales

1. Es muy importante determinar si su sistema *"resetea"* o no los manipuladores de señales, acorde con lo realizado en el punto 3. Si un sistema *"resetea"* el manipulador a su valor por defecto y nosotros *"recapturamos"* la señal inmediatamente que entramos en la función de manipulacion (esto es lo habitual), en general, suele ser la opción correcta, pero no siempre es así. Por ejemplo, la señal `SIGCLD` funciona distinto que `SIGCHLD` y su *"recapturación"* podría provocar un loop infinito: cada vez que usamos la función `signal()` para capturar la señal `SIGCLD`, el kernel chequea si hay procesos hijos por los cuales esperar y en tal caso, ejecuta el manipulador; si dentro del manipulador volvemos a ejecutar la función `signal()` capturándola otra vez, entraremos en loop. Para evitar esto, debemos capturar la señal luego de una llamada a la función `wait()`. El programa 10.3 [Stevens, pag. 282] muestra esta situación.
2. Un programa puede funcionar correctamente en un estado de carga de trabajo normal para el sistema operativo (SO), mientras que podría entrar en una condición de carrera (race condition) cuando el SO tenga una mayor carga de trabajo. Esto podría ocurrir con señales que se supone ocurrirán una única vez: por ejemplo, supongamos el uso de la función `alarm()` para ejecutar cierto código dentro de `n` segundos y luego llamamos a la función `pause()` para detener el programa hasta la ocurrencia de la señal `SIGALRM`, ¿Qué sucede si el programa es interrumpido por el SO después de la ejecución de `alarm()` y antes de la ejecución de `pause()`? La lentitud del sistema es tal, que para cuando el SO le vuelve a asginar tiempo de CPU al proceso, resulta que el timer ya ha expirado, con lo cual debe ejecutar el manipulador de `SIGALRM` (antes de entrar en `pause()`), una vez ejecutado el manipulador, el cual hace su trabajo y retorna; se ejecuta `pause()` esperando la ocurrencia de una señal, si no hay ninguna otra señal capturada en el programa, esta función `pause()` nunca retornará, por lo tanto, el programa quedará bloqueado indefinidamente.
3. Se debe ser muy cuidadoso con el código que se escribe dentro de un manipulador de una señal, por lo general es un código muy pequeño y que activa/desactiva ciertas banderas (flags) que luego son tratados por el programa principal. Tener en cuenta que se trata de una ejecución asincrónica y que existen funciones no re-entrantes, algunas recomendaciones pueden consultarse en la Página 611 del Capitulo 24 "Signal Handling" [LooseSandra, Pág. 611] del manual de referencia de la libreria GNU C/C++.
4. Hay una cuestión de permisos vinculados con las señales, no cualquier proceso puede enviar una señal a otro, esto está en relación con quien es el dueño (owner) del proceso y del grupo de proceso que envía la señal y del proceso que la recibe.
5. Las señales son mensajes asincrónicos que se encolan para cada proceso, otra particularidad es que, cuando se encola `n` veces una misma señal en un proceso determinado en un lapso de tiempo muy pequeño, dicha señal se suele enviar (puesto que este comportamiento puede variar según el tipo de unix utilizado) sólo una vez y no tantas veces como se generó.
6. Un mismo programa puede manejar `n` señales, con lo cual, cuando se está ejecutando un manipulador de una señal, éste puede ser interrumpido para pasarle el control a otro manipulador. Supongamos que tenemos que realizar la ejecución de una sección crítica de código, se pueden suspender determinadas señales durante la ejecución de dicha sección y restaurarlas luego de haber salido de la sección crítica evitando que el manipulador sea interrumpido (ver ejemplo en Capítulo 10 "Signals" [Stevens]).

7. Tener en cuenta la modificación de variables globales desde `main()` y desde los manipuladores de señales, es conveniente declarar las variables globales con el cualificador `volatile` esto impide que se interrumpa el programa mientras se está actualizando el valor de la variable. También ANSI C define el tipo de dato `sig_atomic_t` para lo mismo, ambos pueden combinarse:

```
static volatile sig_atomic_t bandera;
```

Si desea profundizar el tema, recomendamos la lectura del capitulo 10 "Signals" del libro de Stevens en donde se tratan estos problemas y sus propuestas de solución que exceden el alcance de esta publicación. Existe muchísimo más acerca de señales: un proceso puede bloquear señales, trabajar con un conjunto de señales (signal set), tener asociado una máscara de señales, ver si hay señales pendientes, etc. Muchas de las funciónes que Ud. ya conoce hacen uso de señales, tales como: `abort()`, `system()`, `sleep()`, etc.

8. Existen operaciones bloqueantes, por ejemplo, una operacion `read()` puede dejar el proceso bloqueado indefinidamente esperando una recepción de datos que nunca llegará. En este caso, podemos utilizar la función `select()` o `poll()` para indicar allí un período de time-out para una operación de lectura, a modo de evitar esta situación. Pero, ¿qué sucede con el resto de funciones bloqueantes que no tienen esta posibilidad? podemos usar la función `alarm()` antes de la función bloqueante, de la siguiente forma:

```
alarm(5); // por ejemplo, establezco time-out de 5 segundos
llamada_a_funcion_bloqueante();
alarm(0); // desactivo alarma
```

entonces, el manipulador de la señal `SIGALRM` será ejecutado en caso de que la función bloqueante no retorne; desde el manipulador de la señal se puede tomar alguna acción. Pero, ¿qué sucede si pretendo volver de esta función de manipulación de la señal a otra parte del código? para ello debo utilizar las funciones `sigsetjmp()` y `siglongjmp()`; estas funciones derivan de las funciones `setjmp()` y `longjmp()`, pero estan adaptadas para trabajar con señales. Veamos una nueva versión de `tp45.c` utilizando esta técnica:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <setjmp.h>

void usage();
void sig_usr(int signo);
void sig_alrm(int signo);
int salir=0;
int idproc=0;
sigjmp_buf jmpbuf;

int main(int argc,char *argv[]) { // inicio main()
   if ( argc != 2 ) {
      usage();
      exit(1);
   }
   idproc=atoi(argv[1]);
   if ( idproc <= 1 ) {
      fprintf(stderr,"main():%d no es un id de proceso valido\n",idproc);
      usage();
      exit(3);
   }
   if ( kill((pid_t) idproc,0) == -1 ) {
      fprintf(stderr,"main():%d no es un id de proceso valido\n",idproc);
      usage();
      exit(2);
   }
   if (signal(SIGALRM, sig_alrm) == SIG_ERR) {
      fprintf(stderr,"main():imposible capturar SIGALRM\n");
```

```
      exit(4);
   }
   if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
      fprintf(stderr,"main():imposible capturar SIGUSR2\n");
      exit(5);
   }
   if (sigsetjmp(jmpbuf,1)) {
      printf("main():dentro de 60 segundos vuelvo a pedir totales!\n");
   }
   alarm(60);
   for ( ;!salir; ) {
      pause();
   }
   return 0;
} // fin main()

void sig_usr(int signo) {     /* signo=nro de señal */
   if (signo == SIGUSR2)  salir=1;
   else printf("sig_usr():recibi señal %d\n", signo);
   return;
}
void sig_alrm(int signo)      {     /* signo=nro de señal */
   if (kill((pid_t) idproc,SIGUSR1) != 0) {
      fprintf(stderr,"sig_alrm():Error enviando señal SIGUSR1 a %d\n",idproc);
      raise(SIGUSR2); // envio señal de salida
   }
   siglongjmp(jmpbuf,1);
}
void usage() {
   printf("tp44 <tp43 proc.id>\n");
   printf("Donde:\n");
   printf("<tp43 proc.id>=nro. de proceso de tp43 que debe estar previamente en
ejecucion\n");
}
```

la salida a obtener por este programa sería (suponiendo que el programa que emite los totales de ventas de las sucursales se esta ejecutando previamente con el id de proceso 3079):

```
grchere@debian:~/gccwork/src/signals$ ./tp46 3079
main():dentro de 60 segundos vuelvo a pedir totales!
suc[0]=1498.857343
suc[1]=1204.173758
suc[2]=1590.140845
suc[3]=2070.052891
suc[4]=1668.643526
suc[5]=1735.123461
suc[6]=1237.069390
suc[7]=1876.513227
suc[8]=1316.073510
suc[9]=1736.023419
main():dentro de 60 segundos vuelvo a pedir totales!
suc[0]=1514.862496
suc[1]=1408.207045
suc[2]=1601.421418
suc[3]=2182.057773
suc[4]=1706.484476
suc[5]=1786.676463
suc[6]=1408.395217
suc[7]=2033.849564
suc[8]=1457.069481
suc[9]=1943.772091
....
```

analizando la salida, podrá deducir el funcionamiento de `sigsetjmp()`, `siglongjmp()`.

Parte practica:
tema 1
Dado el siguiente código: (p1.2013.c)

```
...
int main(int argc, char **argv) {
      pid_t pid;
      printf("\nH\n");
      pid = fork();
      if ( pid == 0 ) {
            printf("Z\n");
            pid_t pid2 = fork();
            if ( pid2 == 0 ) {
                  printf("M\n");
            } else {
                  wait(NULL);
                  pid_t pid3 = fork();
                  if ( pid3 == 0 ) {
                        printf("X\n");
                  } else {
                        wait(NULL);
                  }
            }
      } else {
            wait(NULL);
            pid_t pid4 = fork();
            if ( pid4 == 0 ) {
                  printf("Y\n");
            } else {
                  wait(NULL);
                  printf("\n");
            }
      }
      exit(0);
}
```

1.a) se generan procesos zombies?.
1.b) se generan procesos huerfanos?
1.c) cual es la salida por consola o bien la misma es indeterminada?

ej1) a) no b) no
salida del programa:
H
Z
M
X
Y

2)

Dado el siguiente código. Especifique cual es el resultado de su ejecución. (p1_2.2013.c)

```
...
void handler(void);
int fd = 0;
int main () {
```

```
      fd = open("abecedario",O_CREAT|O_TRUNC|O_WRONLY, 0700);
      signal (SIGUSR1,(void *) handler);
      alarm(7);
      while (1) raise (SIGUSR1);
      close(fd);
      exit(0);
}

void handler (void) {
      static char letra = 'A';
      write(fd,&letra,1);
      letra ++ ;
      pause();
}
```

solo graba la "A" en archivo "abecedario"

3) **¿Escriba la función main() para que un hilo escriba el abecedario en el archivo?(p1_3.2013.c)**

```
...
void *funcion(void *);

int main(void) {
      ???
}
void *funcion(void *p) {
      int fd = open("abecedario",O_CREAT|O_TRUNC|O_WRONLY, 0700);
      int letra = 'A';
      while (letra <= 'Z') {
            write(fd,&letra,1);
            letra++;
            sleep(2);
      }
      close(fd);
      printf("funcion():fin!\n");
      pthread_exit(0);
}
```

```
pthread_t thr;
pthread_create(&thr,NULL,funcion,NULL);
pthread_join(thr,NULL);
printf("main():fin!\n");
exit(0);
```

4

Pipes: Proceso Padre - Hijo - Nieto **(2)**

Padre conectado con Hijo, Hijo conectado con Nieto y Nieto conectado con Padre, comienza el Padre escribiendo un carácter por ejemplo el carácter 'a' el Hijo genera el próximo carácter (queda ab) el Nieto genera el próximo carácter (queda abc) el Padre genera el próximo carácter (queda abad) así continúan mientras no se genere el carácter 'Z' entonces el Padre guarda en un archivo toda la cadena y todo termina.

(p1_4_2.2013.c)

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

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

void handler_hijo(void);
void handler_nieto(void);

int main(int argc, char **argv) {
	pid_t pid;
	int p[2],ph[2],pn[2];
	// creo pipes
	pipe(p);
	pipe(ph);
	pipe(pn);
	char nulo = 0;
	pid = fork();
	if ( pid > 0 ) {
		// padre (escritor,lector)
		close(p[1]);   // cierro escritor padre
		close(ph[0]);  // cierro lector hijo
		close(pn[0]);  // cierro lector nieto
		close(pn[1]);  // cierro escritor nieto
		char letra = 'A',uletra=letra;
		printf("padre graba %c\n",letra);
		write(ph[1],&letra,1);
		write(ph[1],&nulo,1);
		sleep(1);          // para asegurarme que ya hijo y nieto estan en pause()
		kill(pid,SIGUSR1);  // envio señal para despertar hijo
		while( letra < 'Z' && read(p[0],&letra,1) == 1 ) {
			if ( letra != nulo ) {
				printf("padre graba %c\n",letra);
				write(ph[1],&letra,1);
				uletra=letra;
			} else {
				// agrego letra
				letra=uletra+1;
				printf("padre graba %c\n",letra);
				write(ph[1],&letra,1);
				write(ph[1],&nulo,1);
			}
		}
		wait(NULL); // espero finalizacion hijo
		close(p[0]);
		close(ph[1]);
		printf("fin padre\n");
	} else {
		// hijo (escritor,lector)
		pid_t pid2 = fork();
		if ( pid2 > 0 ) { // hijo (escritor,lector)
```

```
        int seniale=0;
        signal(SIGUSR1,(void *) handler_hijo);
        pause();  // quedo a la espera de la señal del padre
        close(p[0]);  // cierro lectura pipe padre
        close(p[1]);  // cierro escritura pipe padre
        close(ph[1]); // cierro escritura pipe hijo
        close(pn[0]); // cierro lectura pipe nieto
        char letra=' ',uletra=letra;
        while( letra < 'Z' && read(ph[0],&letra,1) == 1 ) {
                if ( letra != nulo ) {
                        printf("hijo graba %c\n",letra);
                        write(pn[1],&letra,1);
                        uletra=letra;
                } else {
                        letra=uletra+1;
                        printf("hijo graba %c\n",letra);
                        write(pn[1],&letra,1);
                        write(pn[1],&nulo,1);
                        // agrego letra
                        if ( !seniale ) {  // señalo nieto
                                seniale=1;
                                kill(pid2,SIGUSR1);
                        }
                }
        }
        // cierro los manipuladores que use
        close(pn[1]);
        close(ph[0]);
        wait(NULL); // espero finalizacion nieto
        printf("fin hijo\n");
} else { // nieto (escritor,lector)
        signal(SIGUSR1,(void *) handler_nieto);
        pause();  // quedo a la espera de señal del hijo
        close(p[0]);   // cierro lectura pipe padre
        close(ph[1]);  // cierro escritura pipe hijo
        close(ph[0]);  // cierro lectura pipe hijo
        close(pn[1]);  // cierro escritura pipe nieto
        char letra = ' ',uletra=letra;
        while( letra < 'Z' && read(pn[0],&letra,1) == 1 ) {
                if ( letra != nulo ) {
                        printf("nieto graba %c\n",letra);
                        write(p[1],&letra,1);
                        uletra=letra;
                } else {
                        // agrego letra
                        letra=uletra+1;
                        printf("nieto graba %c\n",letra);
                        write(p[1],&letra,1);
                        write(p[1],&nulo,1);
                }
        }
        // cierro los manipuladores que use
```

```
                close(p[1]);
                close(pn[0]);
                printf("fin nieto\n");
            }
    }
    return 0;
}

void handler_hijo(void)  { printf("recibi señal hijo\n"); }
void handler_nieto(void) { printf("recibi señal nieto\n"); }
```

5

Existen dos procesos independientes cuyos identificadores de procesos están almacenados en dos archivos uno llamado pidUser1 y el otro en pidUser2, también existe una función que retorna un pid_t de un archivo, cuyo prototipo es: **(2)**

pid_t LeePidDeArchivo(char * nombreArchivo)

Realizar el código de un proceso User1 y User2 que cooperan para generar el abecedario en un archivo usando señales (SIGUSR1 y SIGUSR2) como herramienta de sincronización.La comunicación es la siguiente: Ambos procesos son adoradores del abecedario. Entonces deciden generar el abecedario cooperativamente en un archivo llamado "abecedario", puede comenzar cualquiera de los dos, escribiendo la letra 'A' en el archivo, de la siguiente manera: si read(fd,&letra,1) == 0 significa que el archivo esta vacío y se debe escribir la 'A', de lo contrario se le suma 1 a la letra y se genera la siguiente letra. Existe una función que retorna la última letra grabada en el archivo "abecedario":

char DameUltimaLetraGrabada(int fd)

Existe una función que escribe la letra en el archivo "abecedario"

void EscribeLetra(char letra,int fd)

Proceso independiente 1: (p1_5_1.2013.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <string.h>

pid_t LeePidDeArchivo(char *archivo);
char DameUltimaLetraGrabada(int fd);
void EscribeLetra(char letra,int fd);
void grabopid(char *archivo,pid_t p);

void handler_p1_5_1(void);

int main(int argc, char **argv) {
    signal (SIGUSR1,(void *) handler_p1_5_1);
    grabopid("pidUser1",getpid());
```

```
    printf("proceso p1_5_1 pid=%d grabado,espero por pidUser2\n",getpid());
    while(access("pidUser2",F_OK)!=0) sleep(1);
    pid_t pid = LeePidDeArchivo("pidUser2");
    printf("proceso p1_5_1 pid=%d colaborando con p1_5_2 pid=%d\n",getpid(),pid);
    int fd = open("abc",O_CREAT | O_RDWR,0660);
    char letra;
    if ( read(fd,&letra,1) == 0 ) letra='A';
    else letra=DameUltimaLetraGrabada(fd)+1;
    while(letra<='Z') {
            printf("p1_5_1 escribo letra %c\n",letra);
            EscribeLetra(letra,fd);
            kill(pid,SIGUSR2);
            pause();
            letra=DameUltimaLetraGrabada(fd)+1;
    }
    close(fd);
    remove("pidUser1");
    printf("fin p1_5_1\n");
    return 0;
}

void handler_p1_5_1(void) {
    printf("p1_5_1 llego señal SIGUSR1\n");
}


// lee un pid
pid_t LeePidDeArchivo(char *archivo) {
    int fd;
    char pids[5];
    memset(pids,0,5);
    while( (fd=open(archivo,O_RDONLY)) == -1);
    read(fd,pids,5);
    close(fd);
    int n = atoi(pids);
    return (pid_t) n;
}

// grabo un pid
void grabopid(char *archivo,pid_t p) {
    // grabo mi pid
    char pids[255];
    int fd;
    sprintf(pids,"%d",p);
    fd = open(archivo,O_CREAT | O_WRONLY,0660);
    write(fd,pids,strlen(pids)+1);
    close(fd);
}

char DameUltimaLetraGrabada(int fd) {
    lseek(fd,-1,SEEK_END);
    char letra;
```

```
    read(fd,&letra,1);
    return letra;
}

void EscribeLetra(char letra,int fd) {
    lseek(fd,0,SEEK_END);
    write(fd,&letra,1);
}
```

Proceso Independiente 2: (p1_5_2.2013.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <string.h>

pid_t LeePidDeArchivo(char *archivo);
char DameUltimaLetraGrabada(int fd);
void EscribeLetra(char letra,int fd);
void grabopid(char *archivo,pid_t p);

void handler_p1_5_2(void);

int main(int argc, char **argv) {
    signal (SIGUSR2,(void *) handler_p1_5_2);
    grabopid("pidUser2",getpid());
    printf("proceso p1_5_2 pid=%d grabado,espero por pidUser1\n",getpid());
    while(access("pidUser1",F_OK)!=0) sleep(1);
    pid_t pid = LeePidDeArchivo("pidUser1");
    printf("proceso p1_5_2 pid=%d colaborando con p1_5_1 pid=%d\n",getpid(),pid);
    int fd = open("abc",O_CREAT | O_RDWR,0660);
    char letra;
    if ( read(fd,&letra,1) == 0 ) letra='A';
    else letra=DameUltimaLetraGrabada(fd)+1;
    while(letra<='Z') {
            printf("p1_5_2 escribo letra %c\n",letra);
            EscribeLetra(letra,fd);
            kill(pid,SIGUSR1);
            pause();
            letra=DameUltimaLetraGrabada(fd)+1;
    }
    close(fd);
    remove("pidUser2");
    printf("fin p1_5_2\n");
    return 0;
}
```

```
void handler_p1_5_2(void) {
    printf("p1_5_2 llego señal SIGUSR2\n");
}


// lee un pid
pid_t LeePidDeArchivo(char *archivo) {
    int fd;
    char pids[5];
    memset(pids,0,5);
    while( (fd=open(archivo,O_RDONLY)) == -1);
    read(fd,pids,5);
    close(fd);
    int n = atoi(pids);
    return (pid_t) n;
}

// grabo un pid
void grabopid(char *archivo,pid_t p) {
    // grabo mi pid
    char pids[255];
    int fd;
    sprintf(pids,"%d",p);
    fd = open(archivo,O_CREAT | O_WRONLY,0660);
    write(fd,pids,strlen(pids)+1);
    close(fd);
}

char DameUltimaLetraGrabada(int fd) {
    lseek(fd,-1,SEEK_END);
    char letra;
    read(fd,&letra,1);
    return letra;
}

void EscribeLetra(char letra,int fd) {
    lseek(fd,0,SEEK_END);
    write(fd,&letra,1);
}
```

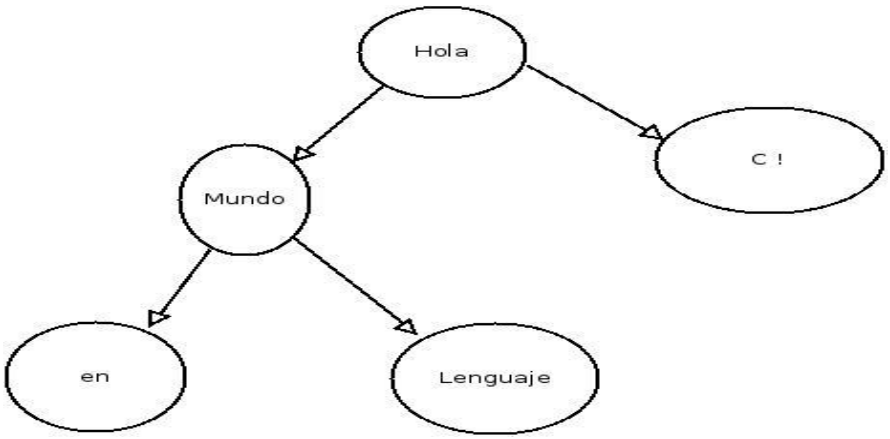
ambos programas tienen riesgo de interbloqueo

Parte practica:

tema 2

1

Dado el siguiente diagrama que indica cada proceso y el mensaje que emite cada proceso en consola:



(p1.2013.c solo cambiar los printf por el mensaje)

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

int main(int argc, char **argv) {
    pid_t pid;
    printf("\nHola\n");
    pid = fork();
    if ( pid == 0 ) { // hijo z
            printf("Mundo\n");
            pid_t pid2 = fork();
            if ( pid2 == 0 ) { // hijo m
                    printf("en\n");
            } else { // hijo z
                    wait(NULL); // espero m
                    pid_t pid3 = fork();
                    if ( pid3 == 0 ) { // hijo x
                            printf("Lenguaje\n");
                    } else { // hijo z
                            wait(NULL); // espera x
                    }
            }
    } else { // padre
            wait(NULL); // espero z
            pid_t pid4 = fork();
```

```
        if ( pid4 == 0 ) { // hijo y
                printf("C!\n");
        } else {        // padre h
                wait(NULL); // espera y
                printf("\n");
        }
    }
    exit(0);
}
```

2.

Dado el siguiente código

```
...
int main(int argc, char * argv[]){
        pid_t pid;
        int i ;
        for(i = 0; i < atoi(argv[1]); i++) {
                pid = fork();
                if ( pid > 0 ) break;
        }
        printf("yo %d Mi Padre %d Mi hijo %d",……,……,……);
        exit(0);
}
```

a) Dibujar la estructura de procesos que genera el código anterior si se llama a ejecución con la siguiente línea de comando $ ./a.out 4
b) Complete las líneas punteadas del printf
c) ¿Se generan procesos huérfanos? Justifique.
d) ¿Se generan procesos zombies? Justifique.

a) todos procesos encadenamos padre – hijo – nieto – bisnieto
b) printf("yo %d Mi Padre %d Mi hijo %d",getpid(),getppid(),pid);
c,d) si, es posible xq no hay uso de wait()

3.

Dado el siguiente código incompleto (productor – consumidor utilizando threads), modifique los hilos de productor y consumidor par utilizar el mutex mu de la forma apropiada.

```
...
char elemento();
void *productor(void *);
void *consumidor(void *);
#define BUFFER_SIZE 5
char buffer[BUFFER_SIZE];
int count=0;
pthread_mutex_t mu;

int main(int argc, char **argv) {
      ...
}
char elemento() {
      static char letra = 'A' - 1;
      if ( letra == 'Z') letra = 'A';
```

```
        else letra++;
        return letra;
}
void *productor(void *t) {
        int in=0;
        while(1) {
                while(count == BUFFER_SIZE);
                buffer[in] = elemento();
                count++;
                in = (in + 1) % BUFFER_SIZE;
        }
        pthread_exit("fin productor");
}
void *consumidor(void *t) {
        int out=0;
        char mielemento;
        while(1) {
                while(count == 0);
                mielemento = buffer[out];
                count--;
                printf("%c %d\n",mielemento,out);
                out = (out + 1) % BUFFER_SIZE;
        }
        pthread_exit("fin consumidor");
}
```

(thrpcs.c pero cambiando semaforo posix por mutex en seccion critica)

```
void *productor(void *t) {
    int in=0;
    while(1) {
            while(count == BUFFER_SIZE);
            pthread_mutex_lock(&mu);
                    buffer[in] = elemento();
                    count++;
            pthread_mutex_unlock(&mu);
            in = (in + 1) % BUFFER_SIZE;
    }
    pthread_exit("fin productor");
}

void *consumidor(void *t) {
    int out=0;
    char mielemento;
    while(1) {
            while(count == 0);
            pthread_mutex_lock(&mu);
                    mielemento = buffer[out];
                    count--;
            pthread_mutex_unlock(&mu);
            printf("%c %d\n",mielemento,out);
            sleep(1);
            out = (out + 1) % BUFFER_SIZE;
    }
    pthread_exit("fin consumidor");
}
```

4

idem tema 1

5

Existen dos threads que dependen de su proceso creador main.
Realizar el código para que los dos threads cooperen para generar el abecedario en un archivo.La comunicación es la siguiente: Ambos threads son adoradores del abecedario.
Entonces deciden generar el abecedario cooperativamente en un archivo llamado "abecedario", puede comenzar cualquiera de los dos, escribiendo la letra 'A' en el archivo, de la siguiente manera: si read(fd,&letra,1) == 0 significa que el archivo esta vacío y se debe escribir la 'A', de lo contrario se le suma 1 a la letra y se genera la siguiente letra. Existe una función que retorna la última letra grabada en el archivo "abecedario":
char DameUltimaLetraGrabada(int fd)
Existe una función que escribe la letra en el archivo "abecedario"
void EscribeLetra(char letra,int fd)

(p1_5.2013.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <fcntl.h>

// prototipo de funciones
void *productor(void *);
char DameUltimaLetraGrabada(int fd);
void EscribeLetra(char letra,int fd);

// variables globales compartidas
char letra='A';
int fd;
volatile int salir=0;

// mutex para implementar exclusión mutua entre productores
pthread_mutex_t mu;

int main(int argc, char **argv)
{

    fd = open("abcd",O_CREAT | O_RDWR,0660);
    if ( read(fd,&letra,1) == 0 ) letra='A';
    else letra=DameUltimaLetraGrabada(fd)+1;

    pthread_t tprod1,tprod2;
   pthread_mutex_init(&mu, NULL);
    pthread_create(&tprod1, NULL, productor,  NULL);
    pthread_create(&tprod2, NULL, productor,  NULL);

    pthread_join(tprod1,NULL);
```

```
	pthread_join(tprod2,NULL);

	close(fd);
	printf("main():fin!\n");
	exit(0);
}

void *productor(void *t) {
	while(!salir) {
		pthread_mutex_lock(&mu);
			if (!salir ) {
				printf("escribo %c\n",letra);
				EscribeLetra(letra,fd);
			}
			if ( letra == 'Z' ) salir=1;
			letra+=1;
		pthread_mutex_unlock(&mu);
	}
	printf("fin productor\n");
	pthread_exit("fin productor");
}

char DameUltimaLetraGrabada(int fd) {
	lseek(fd,-1,SEEK_END);
	char letra;
	read(fd,&letra,1);
	return letra;
}

void EscribeLetra(char letra,int fd) {
	lseek(fd,0,SEEK_END);
	write(fd,&letra,1);
}
```