Problema 5

Dada la siguiente jerarquía de procesos:

Indica para cada proceso qué pids puede conocer del resto de la jerarquía y qué código implementarías para que cada uno muestre esos pids en pantalla.
¿Es posible garantizar que el orden de los mensajes sea siempre el mismo?

Problema 2

La Figura 1 contiene el código del programa *Mtarea* (se omite el control de errores para simplificar y podemos asumir que ninguna llamada a sistema devuelve error).

1. *(T2)* Dibuja la jerarquía de procesos que se genera si ejecutamos este programa de las dos formas siguientes (asigna identificadores a los procesos para poder referirte después a ellos):
    a. ./Mtarea 5 0
    b. ./Mtarea 5 1
2. Indica, para cada una de las dos ejecuciones del apartado anterior:
    a. *(T2)* ¿Será suficiente el tamaño del vector de pids que hemos reservado para gestionar los procesos hijos del proceso inicial? (Justifícalo y en caso negativo indica que habría que hacer).
    b. *(T2)* Qué procesos ejecutarán las líneas 36+37
    c. *(T3)* Qué procesos ejecutarán la función realizatarea
3. *(T3)* Sabemos que una vez cargado en memoria, el proceso inicial de este programa, tal y como está ahora, ocupa: 2Kb de código, 4bytes de datos, 4kb de pila y el tamaño del heap depende de argv[1] (asumid que como mucho será 4kb). Si ejecutamos este programa en un sistema Linux con una gestión de memoria basada en paginación, sabiendo que una página son 4kb, que las páginas no se comparten entre diferentes regiones y que ofrece la optimización COW a nivel de página. Sin tener en cuenta en las posibles librerías compartidas, CALCULA (desglosa la respuesta en función de cada región de memoria):
    a. El espacio lógico que ocupará cada instancia del programa Mtarea (número de páginas)
    b. El espacio físico que necesitaremos para ejecutar las dos ejecuciones descritas en el apartado 1

```
int *pids;
void usage()
{
  char  b[128];
  sprintf(b,"./Mtarea procesosnivel1(cuantos) procesosnivel2(0=no/1=si)\n");
  write(1,b,strlen(b));
  exit(0);
}
void realizatarea(int i){
   // Omitimos su código por simplicidad pero no hay ninguna llamada a sistema relevante
   // para el ejercicio
}

void procesardatos(int i, int multiproceso)
{
    int it;
    if (multiproceso>0){ it=0;  while((fork()>0) && (it<2)) it++;}
    realizatarea(i);
    exit(1);
}
void main(int argc,char *argv[])
{
  int i,ret,procesos;
  char buff[128];
  if (argc!=3) usage();
  procesos=atoi(argv[1]);
  pids=sbrk(procesos*sizeof(int));
  for(i=0;i<procesos;i++){
     ret=fork();
    if (ret==0) procesardatos(i,atoi(argv[2]));
    pids[i]=ret;
  }
  while((ret=waitpid(-1,NULL,0))>0){
    for(i=0;i<procesos;i++){
      if (pids[i]==ret){
        sprintf(buff,"acaba el proceso num %d con pid %d \n" ,i,ret);
        write(1,buff,strlen(buff));
      }
    }
  }
}
```

Figura 1 Código de Mtarea

Problema 3

La Figura 2 contiene el código del programa *ejercicio_exec* (se omite el control de errores para simplificar y podemos asumir que ninguna llamada a sistema provoca un error). Contesta a las siguientes preguntas **justificando todas tus respuestas**, suponiendo que los únicos SIGALRM que recibirán los procesos serán consecuencia del uso de la llamada a sistema *alarm* y que ejecutamos el programa de la siguiente manera: %./ejercicio_exec 4

```
1. int sigchld_recibido = 0;
2. int pid_h;
3. void main(int argc,char *argv[])
4. {
5.   int ret,n;
6.   int nhijos = 0;
7.   char buff[128];
8.
9.   n=atoi(argv[1]);
10.   if (n>0) {
11.         pid_h = fork();
12.         if (pid_h == 0){
13.           n--;
14.           sprintf(buff, "%d", n);
15.           execlp("./ejercicio_exec", "ejercicio_exec", buff, (char *)0);
16.         }
17.         strcpy(buff,"Voy a trabajar \n");
18.         write(1,buff,strlen(buff));
19.         hago_algo_de_trabajo();/*no ejecuta nada relevante para el problema */
20.         while((ret=waitpid(-1,NULL,0))>0) {
21.             nhijos++;
22.         }
23.         sprintf(buff,"Fin de ejecución. Hijos esperados: %d\n",nhijos);
24.         write(1,buff,strlen(buff));
25.   } else {
26.         strcpy(buff,"Voy a trabajar \n");
27.         write(1,buff,strlen(buff));
28.         hago_algo_de_trabajo();/*no ejecuta nada relevante para el problema */
29.         strcpy(buff, “Fin de ejecución\n”);
30.         write(1,buff, strlen(buff));
31.   }
32. }
```

Figura 2 Código del programa ejercicio_exec

1. Dibuja la jerarquía de procesos que se crea y asigna a cada proceso un identificador para poder referirte a ellos en las siguientes preguntas.
2. Para cada proceso, indica qué mensajes mostrará en pantalla

Problema 4

La **Error! Reference source not found.** muestra el código del programa “prog” (por simplicidad, se omite el código de tratamiento de errores). Contesta a las siguientes preguntas, suponiendo que se ejecuta en el Shell de la siguiente manera y que ninguna llamada a sistema provoca error:

%./prog 3

1. (T2) Representa la jerarquía de procesos que creará este programa en ejecución. Asigna a cada proceso un identificador para poder referirte a ellos en el resto de preguntas

2. (T2) ¿Qué proceso(s) ejecutarán las líneas de código 18 y 19?

3. (T2) ¿En qué orden escribirán los procesos en pantalla? ¿Podemos garantizar que el orden será siempre el mismo? Indica el orden en que escribirán los procesos.

```
1.
2.      main(int argc, char *argv[]) {
3.      int nhijos, mipid;
4.      int ret, i;
5.      char buf[80];
6.
7.      nhijos = atoi(argv[1]);
8.      mipid = getpid();
9.      for (i=0; i<nhijos; i++) {
10.         ret = fork();
11.         if (ret > 0){
12.             waitpid(-1, NULL, 0);
13.             sprintf(buf, "Soy el proceso %d y acabo la ejecución\n",getpid());
14.             write(1,buf,strlen(buf));
15.             exit(0);
16.         }
17.     }
18.     sprintf(buf, "Soy el proceso %d y acabo la ejecución\n",getpid());
19.     write(1,buf,strlen(buf));
20.     }
21.
```

Figura 3 Código de prog

Problema 5

Queremos implementar un código que reciba como parámetros una lista de palabras y un fichero y que indique para cada palabra si aparece o no en el fichero. Para ello crearemos tantos procesos como palabras recibamos y cada uno de ellos mutará para ejecutar el comando grep recibiendo como parámetros una de las palabras y el fichero (nota: el comando grep busca el patrón que recibe como parámetro en el fichero que también recibe como parámetro y retorna como parámetro del exit un 0 si no la ha encontrado, un 1 si la ha encontrado y un 2 si ha habido algún error). El proceso padre será el encargado de mostrar un mensaje indicando para cada palabra si existe o no.

1. Implementa una versión de este código en el que los hijos se ejecuten secuencialmente.
2. Haz los cambios mínimos para que el esquema de creación de los hijos sea paralelo.