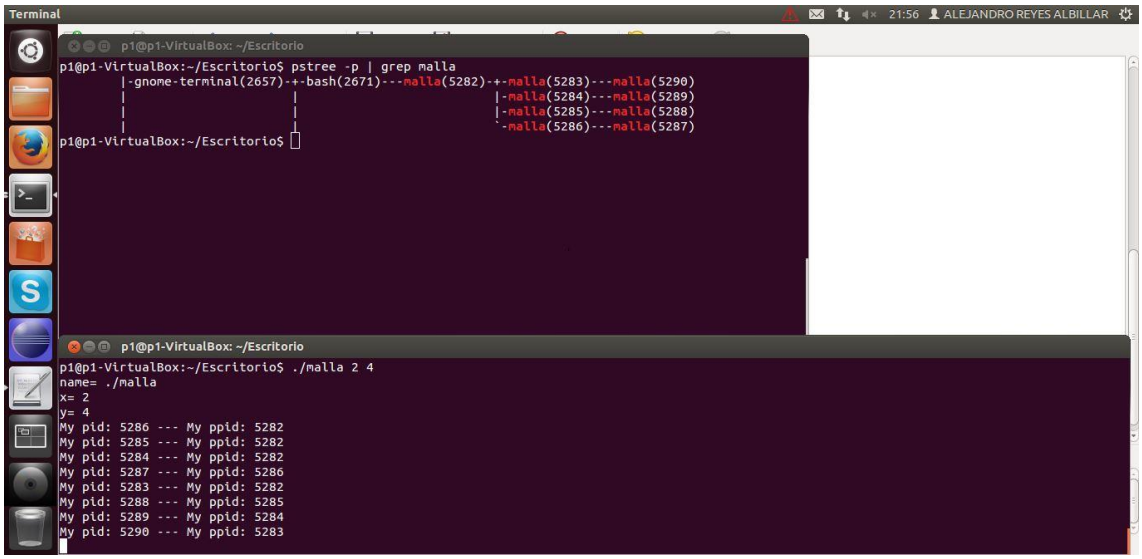
1º) a) malla.c

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

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

  if(argc==3){
    x=atoi(argv[1]);
    y=atoi(argv[2]);
    printf("name= %s\n", argv[0]);
    printf("x= %d\n",x);
    printf("y= %d\n",y);

    for(j=0; j<y; j++){
          //Aqui se crean los hijos del proceso padre
     if(fork()==0){
      printf("My pid: %d --- My ppid: %d\n", getpid(), getppid());
      for(i=0;i<x-1; i++){ //El primer hijo ya se creÃ³ asÃ- que serÃ¡ x-1
    if(fork()==0){
      printf("My pid: %d --- My ppid: %d\n", getpid(), getppid());
    }else{break;}
      }break;
     }
    }sleep(15);
  }
  else{
    printf("Se deben pasar 2 argumentos mediante la sintaxis malla numcolumnas(x)
numfilas(y)");
  }
}
```

ALEJANDRO REYES ALBILLAR GRUPO 1
ara65@alu.ua.es

Éste código se ejecuta mediante la llamada ./malla <numerofilas> <numerocolumnas> y se compilla mediante la orden gcc –o <nombreejecutable> malla.c, utilizando el compilador de C.

Una ejecución de este código se puede ver en la siguiente imagen.

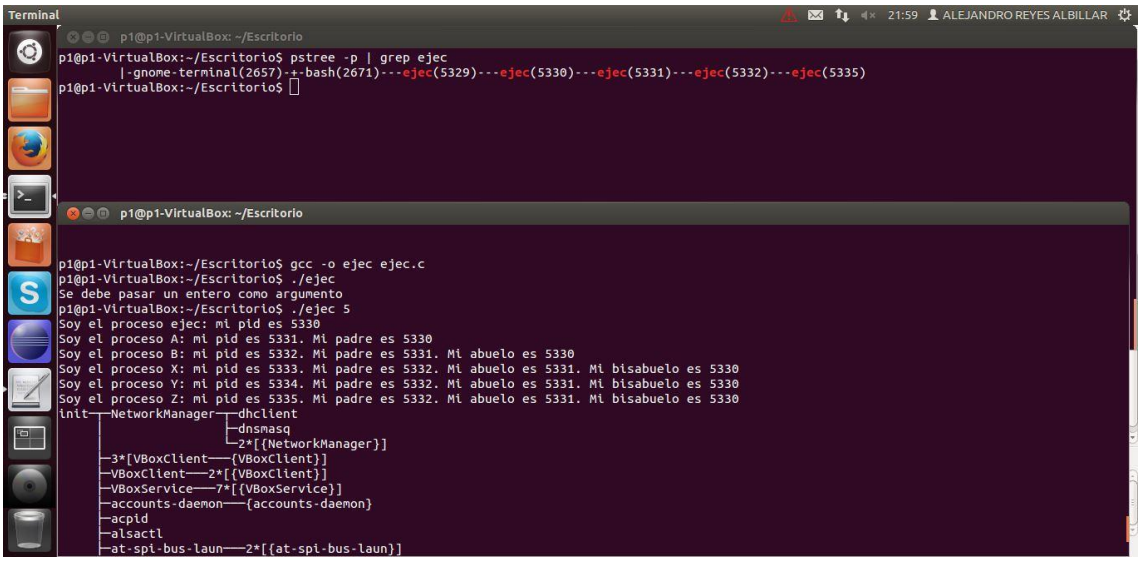


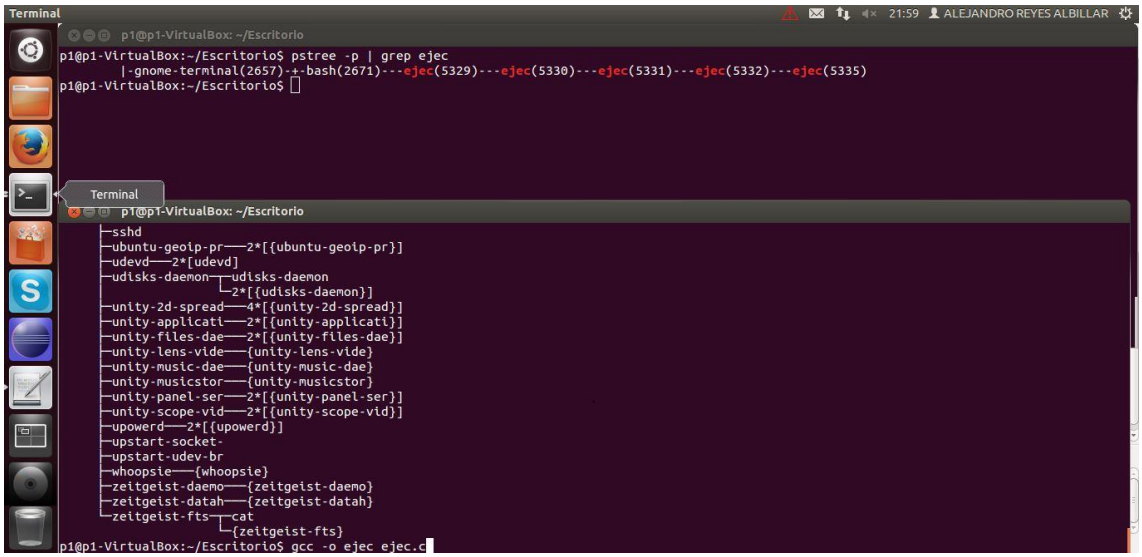
Gracias al comando pstree -p |grep <nombreejecutable> podemos ver el árbol de procesos generado por el código.

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

int main(int argc, char* argv[]){
  int i, j, seguir;
  int bispid, abupid;
  if(argc==2){
    for(i=0; i<=2;i++){
      if(fork()==0){
    if(i==0){printf("Soy el proceso ejec: mi pid es %d\n", getpid());}
    else if(i==1){
      printf("Soy el proceso A: mi pid es %d. Mi padre es %d\n",getpid(),getppid());
      abupid=getppid();
    }
    else if(i==2){
      printf("Soy el proceso B: mi pid es %d. Mi padre es %d. Mi abuelo es %d\n",
getpid(),getppid(),abupid);
```

```
      bispid=abupid;
      abupid=getppid();
      for(j=0; j<=2; j++){
        if(fork()==0){
          switch(j){
          case 0:
            printf("Soy el proceso X: mi pid es %d. Mi padre es %d. Mi abuelo es %d. Mi
bisabuelo es %d\n", getpid(),getppid(),abupid,bispid);
            break;
          case 1:
            printf("Soy el proceso Y: mi pid es %d. Mi padre es %d. Mi abuelo es %d. Mi
bisabuelo es %d\n", getpid(),getppid(),abupid,bispid);
            break;
          case 2:
            printf("Soy el proceso Z: mi pid es %d. Mi padre es %d. Mi abuelo es %d. Mi
bisabuelo es %d\n", getpid(),getppid(),abupid,bispid);
            void pstree(){
               execlp("pstree", "pstree", NULL);
               perror("Error al ejecutar el comando pstree");
               exit(0);
            }
            signal(SIGALRM,pstree);
            alarm(atoi(argv[1]));
            pause();
            break;
          default:
            break;
          }
         exit(0);
        } else{wait();}
      }
    }else{exit(0);}
      }else{wait();exit(0);}
    }
  }
  else{
```

```
    printf("Se debe pasar un entero como argumento\n");
  }
}
```





Este código crea un árbol de 3 hijos en vertical y 3 en horizontal a partir del último en vertical que, transcurrido un tiempo pasado por parámetro, activa una alarma que hace que el último hijo creado ejecute el comando pstree . Debido a que para poder crear los hijos en horizontal, es decir en paralelo, debemos matar con la llamada exit() al proceso en cuestión, que sería hermano, no podemos realizar la segunda parte del ejercicio.

El ejercicio 1, tanto el apartado a) como el b) están ejecutados tras compilarlos en c, con el comando gcc –o. En el resto de ejercicios, debido a requerimientos del mismo, se compilaran los ejercicios en c++ con la orden g++ -o.

2º)

```
#include <iostream>

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>
```

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

using namespace std;

const char* toString(int i,char* argv){
  int use;
  int cont=0;
  string s="";
  string aux="";
  string aux2="";
  string ss=argv;
  ss+=".h";
  int k=i;
  int m=i;
  while(k>-1){
    use=k%10;
    if(i<10){
      s+='0';
      switch(i){
    case 0:
      s+='0';
      break;
    case 1:
      s+='1';
      break;
    case 2:
      s+='2';
      break;
    case 3:
      s+='3';
      break;
    case 4:
      s+='4';
      break;
```

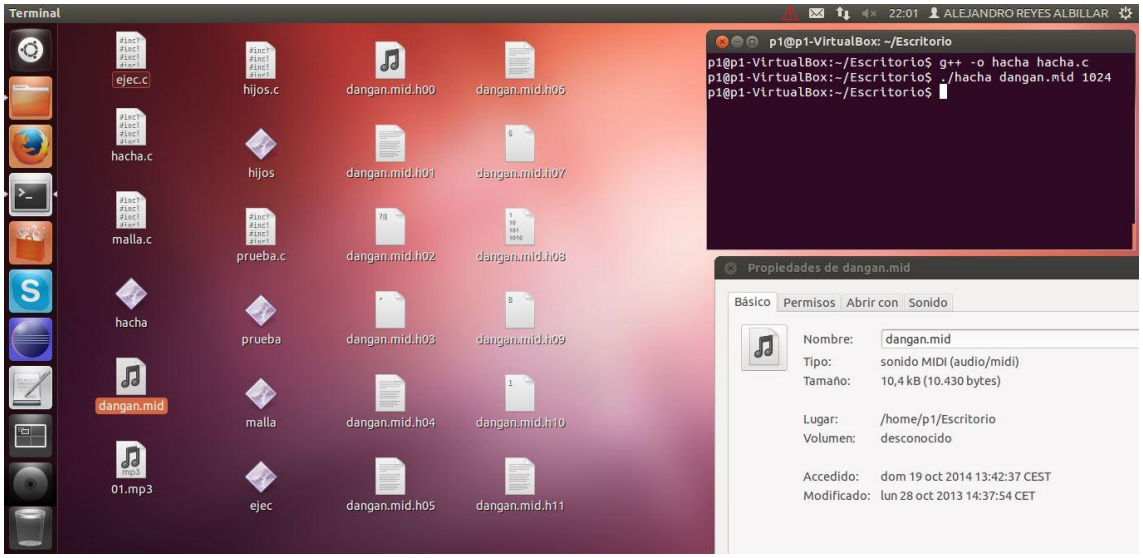
```
case 5:
  s+='5';
  break;
case 6:
  s+='6';
  break;
case 7:
  s+='7';
  break;
case 8:
  s+='8';
  break;
case 9:
  s+='9';
  break;
default:
  break;
  }
}
else{
  switch(use){
case 0:
  s+='0';
  break;
case 1:
  s+='1';
  break;
case 2:
  s+='2';
  break;
case 3:
  s+='3';
  break;
case 4:
  s+='4';
```

```
    break;
  case 5:
    s+='5';
    break;
  case 6:
    s+='6';
    break;
  case 7:
    s+='7';
    break;
  case 8:
    s+='8';
    break;
  case 9:
    s+='9';
    break;
  default:
    break;
    }
    cont++;
  }
  k=k/10;
  if(k==0){
    k=-1;
  }
}
if(m>=10){
  for(int i=s.size()-1;i>=0;i--){
    aux+=s[i];
  }
  s=aux;
}

ss+=s;
return ss.c_str();
```

```
}



int main(int argc, char * argv[]){
  int nuevo, tam, cont;
  char buffer[atoi(argv[2])+1];
  if(argc==3){
    int abierto=open(argv[1], O_RDONLY);
    if(abierto>=0){ //Si abre el archivo
      //aqui tenemos que poner un while para que lea varias veces el archivo hasta el
final
      cont=0;
      while(tam>0){//necesito condicion para que finalice el bucle
    tam=read(abierto, buffer, atoi(argv[2]));//Devuelve el nÃºmero de bytes leidos con
Ã©xito
    if(tam==-1){ //Error al leer el archivo
      perror("Error de lectura\n");
    }
    else{
      buffer[tam]=0;//Pone el final de la cadena
      //hacemos algo con lo contenido en buffer
      nuevo=creat(toString(cont, argv[1]),0777);
      if(nuevo<0){ //Errror al crear
        perror("Error al crear \n");
      }
      else{
        //Aqui  ponemos los datos dentro del archivo
        if(write(nuevo, buffer, strlen(buffer)) != strlen(buffer)){
          perror("Error al escribir");//Error al escribir
          break;
        }
        close(nuevo);
      }
      cont++;
    }
      }
```

```
      close(abierto);
    }
    else{ //No se ha podido abrir el archivo
      perror("No se ha podido abrir el archivo");
      exit(0);
    }
  }
  else{ //Error de argumentos
    printf("Error de argumentos, debes pasar los argumentos de este modo:\n ./hacha
<nombrearchivo> <tamañoarchivodividido>\n");
    exit(0);
  }
  exit(0);
}
```

Este ejercicio utiliza strings para poner las extensiones a los archivos por lo que es necesario que se compile con c++, ya que el compilador de c tiene multitud de restricciones y no reconoce correctamente muchas de las funciones que en c++ están implementadas.



Este programa, dado un archivo y un tamaño en bytes pasados por parámetro divide el archivo en cuantos archivos sean necesarios añadiéndole al nombre del archivo pasado por parámetro la extensión .hXX, donde XX será un número que inicia en 00 y que continúa indefinidamente según se necesiten más o menos archivos de partición.

Debido a que los archivos .mp3 y .mid, es decir de sonido, suelen contener caracteres especiales propios de ellos mismos y de los métodos Unicode, algunos archivos suelen parecer vacíos o incluso no llegarse a realizar la partición por lo que recomendamos tener cuidado y paciencia. Se recomienda utilizar un fichero de texto para comprobar que se realiza correctamente la operación. Es también cierto que debido al bucle introducido, el ultimo archivo creado siempre será un archivo vacío de 0 bytes de tamaño pero no nos importa ya que no era lo que se buscaba en esta práctica, sino el hecho de saber utilizar los ficheros y los procesos como es debido.

¡¡ATENCIÓN!!

Para archivos de más de 1 MB se recomienda aumentar considerablemente la cantidad de bytes que tendrá cada archivo fragmentado debido a la cantidad de archivos creados, un archivo de 8.17 MB se convierten en más de 8000 archivos fragmentados, por lo que el sistema puede tener problemas para manejar tal cantidad de archivos.

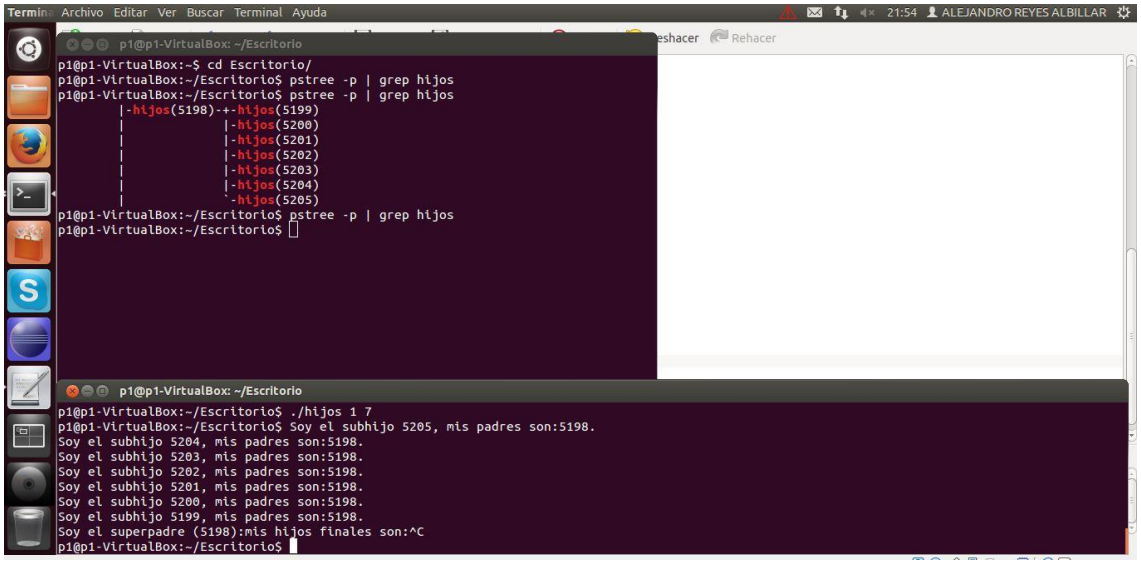
3º)

Debido a múltiples errores y a imprevistos este ejercicio ha sido compilado en c++, dejando de lado todas las opciones de memoria compartida que queríamos utilizar de C.

Este ejercicio crea una serie de hijos y padres almacenando sus pid's en distintas variables, las cuales son vectores ya que no he encontrado mejor manera de que realizaran su trabajo de una forma más eficiente. Dado que tampoco suele funcionar todo como se desea cuando se trabaja con funciones nuevas, los pid's de los hijos no se han almacenado correctamente en el vector sons creado en el siguiente código por lo que la ejecución ha sido parada manualmente para evitar la entrada a un bucle mediante el uso de Ctrl+C, más comúnmente visto como ^C en el terminal.

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <vector>
using namespace std;
int main(int argc, char * argv[]){
  if (argc==3){
    vector<int> sons;
    vector<int> padres;
    for(int i = 0;i < atoi(argv[1]);i++){ //Crea los hijos verticales
      if(fork()==0){
        padres.push_back(getpid());
        if(i==atoi(argv[1])-1){
          for(int j = 0;j < atoi(argv[2]);j++){ // crea los hijos horizontales
           if(fork()==0){
             sons.push_back(getpid());//No lo realiza correctamente, no entiendo el
porqué
             cout <<"Soy el subhijo "<<getpid()<<", mis padres son:";
             for(int k=0;k<padres.size();k++){//imprime los padres
               if(k<padres.size()-1){
                 cout << padres[k] << ", ";
               }
               else{
                 cout << padres[k] << "."<<endl;
```

```
            }
          }
          exit(0);
        }
        else{
          wait();
        }
        }
      }
       //Se dicen quien es el superpadre y cuales son sus hijos
      if(i==atoi(argv[1])-1){
        cout <<"Soy el superpadre ("<<getpid()<<"):mis hijos finales son:";
        for(int m=0;m<sons.size();m++){//imprime los hijos
         if(m<sons.size()-1){
          cout << sons[m] << ", ";
         }
         else{
          cout << sons[m] << "."<<endl;
         }
        }
      }
    }
    else{
      wait();
      exit(0);
    }
  }
  sleep(5);
 }
 else{// Error de argumentos
  printf("Errror en los argumentos. Se deben pasar los argumentos siguiendo el
siguiente esquema: \n./hijos <numHijosVerticales> <numHijosHorizontales>\n");
 }
 return (0);
}
```



Aunque la impresión de los mensajes no se realizan de la manera que deberían debido al problema anteriormente explicado, el árbol de procesos se crea correctamente tal y como se desea. En este caso, debido a un descuido, la ejecución fue tomada con el comando ./hijos 1 7 que crea un único padre, hijo del proceso inicial, que crea a 7 hijos hermanos entre sí. En otras pruebas realizadas sobre el mismo código, se puede observar que la frase que cada hijo emite:

“Soy el subhijo <pid>, mis padres son:<padre1><padre2>…<padre x>”

Incluye los mismos pids, cada uno de un proceso diferente, que han ido añadiéndose al vector que se ha creado.

Dado que los pids de los hijos no se han podido introducir correctamente en el vector sons, debido a un problema desconocido, la frase del superpadre:

“Soy el superpadre (<pid>): mis hijos finales son: <hijox1><hijox2>…<hijo xy>”

No se realiza correctamente y se debe interrumpir con ^C como se ve en la imagen para evitar el colapso del sistema.