TEST1 2015-2016

EJERCICIO 1

El siguiente código:

```
Node *p;
p=malloc(sizeof(Node));
for( p = list ; p!= nil ; p = p-> next){
        printf("Nodo %d\n",p->id);
}
```

Seleccione una:

a) Está mal, hay que hacer malloc en cada iteración del bucle

b) Atravesará un puntero a nil si la lista está vacia

c) Está perfecto d) Está mal, tiene un leak innecesario

e) Debería reservar sizeof (Node *) en lugar de sizeof(Node)

EJERCICIO 2

El valor de retorno de fork:

a) Devuelve cuantos hijos ha tenido este proceso

b) Devuelve a un proceso su propio pid

c)Es igual para el padre y para el hijo, puesto que son indistinguibles.

d)Es la forma de distinguir el proceso padre del hijo e) Solo es importante si es menor que cero, que es un error

EJERCICIO 3

```
int
main (int argc, char *argv[]){
        int i;
        for(i=0;i<5;i++){
                if ( i == 2)
```

```
                continue;
                if ( i > 3)
                        break;
                printf("%d",i);
        }
    }
```

a) imprime 013

b) imprime 0134

c) imprime 01234

d) imprime 01134

e) imprime 0123

EJERCICIO 4

```
    write(1,"hola",4);
```

Seleccione una:

a) Deberías tener un 5 en lugar del 4 para poder escribir una línea en la salida estándar

b) Escribe cuatro caracteres en la pantalla

c) Escribe cuatro caracteres en la salida estándar del proceso

d) Debería usar 0 en lugar de 1

e) No compila, el segundo parámetro es incorrecto

EJERCICIO 5

```
    nw=write(fd,buf, Nbytes);
    if (nw < 0){
            err(1,"write ha fallado\n");
    }
```

a) No es necesario comprobar el valor de retorno de write, siempre es Nbytes.

b) La llamado a write puede fallar y no lo detectaría el if.

c) Podría devolver un número menor que NBytes y no sería un error.

d) Solo estaría bien si la condición comprobase si nw es menor o igual que 0.

e) Nunca va a entrar en el if, write siempre devuelve Nbytes

EJERCICIO 6

```
int
main(int argc, char *argv[ ]){
        int i;
        argc--;
        argv++;
        for(i=0;i<argc;i++){
        printf("%s\n",argv[i]);
        }
        return 0;
}
```

Si ejecutamos

```
$ programa uno dos tres
```

Seleccione una:

a) No puede decrementar argc

b) Imprime dos líneas: dos y tres

c) Hace exactamente lo mismo que el comando echo

d) Imprime tres líneas: uno, dos, tres

e) main nunca puede retornar

EJERCICIO 7

```
int
main( ){
printf ("A");
```

```
        fork();
                fork();
                printf("B");
        }
```

Selecciona una:

a) Escribe A una vez y B una veces

b) Escribe A dos veces y B dos veces

c) Escribe A una vez y B dos veces

d) No puede llamar a printf tras llamar a fork

e) Escribe A una vez y B cuatro veces

EJERCICIO 8

```
        res=execv(cmd,argv);
        if (res < 0)
                err(1,"exec %s failed",cmd);
        printf("exec worked\n");
```

Seleccione una:

a) Nunca imprimirá exec worked

b) Siempre imprimirá exec worked.

c) Está mal: no se puede llamar a exec sin haber llamado a fork

d) Está mal: El printf debería estasr en un else

e) Devuelve en res el pid del hijo o -1 si es un error.

EJERCICIO 9

La siguiente función:

```
        char*
        readfile(int fd){
                char *p;
```

```
int nr;
nr=0;
do{
        nr=read(fd,p+nr,200);
        if ( nr < 0)
                err(1,”read”);
        }while(nr>0)
        return p;
}
```

a) Está mal, no se puede hacer return de un puntero.

b) Está mal, read necesitaría usar un puntero a FILE

c) Está mal, tiene problemas de memoria

d) Está bien, lee un fichero entero a memoria y devuelve el puntero a su inicio

e) Está mal, no se controla bien el error de read

EJERCICIO 10

Si uso fread(de stdlib) para leer de un fichero de 32 en 32 bytes....

Selecciona una:

a) Nunca se realiza una llamada al sistema, fread accede al disco directamente, por eso es más rápido

b) Se comprimen los datos leídos, por eso es más rápido

c) No funciona, fread solo sirve para leer líneas del fichero.

d) Se usa recolección de basura para la memoria dinámica, y por eso es más rápido

e) Se realizan menos llamadas al sistema de lectura que si uso directamente read, por eso es más rapido

TEST1 2017-2018

EJERCICIO 1

El siguiente código:

```
while(1){
        res=execv(cmd,argv);
        if (res<0){
                err(1,”exec %s failed”,cmd);
        }
}
```

Seleccione una:

a) No tiene sentido, va a ejecutar muchas veces el comando.

b) Está bien porque considera el posible error del exec

c) Está mal, no va a entrar nunca en el bucle.

d) No tiene sentido, sobra el bucle y el if

e) Está mal, después del exec hay que llamar a fork

EJERCICIO 2

El siguiente código:

```
for(i=0;i<3;i++){
        fork();
}
fprintf(stderr,”hola\n”);
exit(0);
```

Seleccione una:

a) Escribe 8 lineas en la salida de error

b) Escribe 8 lineas en la salida de estandar

c) Escribe 3 lineas en la salida de error

d) Escribe 3 lineas en la salida de estandar

e) Escribe 4 lineas en la salida de error

EJERCICIO 3

El siguiente código:

```
f=fopen(“/tmp/x”,”w”);
fprintf(f,”hi\n”);
for(;;){


}
```

Seleccione una:

a) /tmp/x tiene 3 bytes

b) /tmp/x tiene 4 bytes

c) No se puede escribir en el directorio /tmp

d) /tmp/x tiene 2 bytes

e) Es muy posible que no se escriba nada en el fichero

EJERCICIO 4

El siguiente código:

```
int i;
int *p;
p=malloc(100*sizeof(int));
if (p==NULL)
        err(1,”malloc falla”);
memset(p,0,sizeof(p));
for(i=0;i<100;i++)
        if (p[i]!=0)
                errx(1,”array mal inicializado”);
printf(“array inicializado ok”);
```

Seleccione una:

a) No sabemos a priori que mensaje va mostrar

b) Muestra array inicializado ok

c) Muestra array mal inicializado

d) Está mal inicialiazado el array

EJERCICIO 5

Cuando un proceso llama a fork

a) Tiene que llamar a wait siempre

b) Siempre va a hacer un exec

c) El hijo comparte variables con el padre, tiene que tener cuidado con no sobreescribir los valores.

d) No puede retornar de la llamada, si retorna es un error.

e) El hijo puede dar valor a sus variables de forma independiente.

TEST1 – NOVIEMBRE 2019-2020

PREGUNTA 1

El siguiente código, suponiendo que el resto del programa es correcto y omitiendo la comprobación de errores:

```
int sts;
if (fork()==0){
        printf("hola\n");
        exit(EXIT_SUCCESS);
}
else{
        wait(&sts);
        printf("adios\n");
}
```

a) Escribe dos líneas y no podemos estar seguros del orden en el que se escribirán hola y adiós.

b) Escribe dos líneas y estamos seguros de que siempre escribe antes hola que adiós.

c) Sólo escribe adiós.

d) Sólo escribe hola.

e) Escribe dos líneas y estamos seguros de que siempre escribe adiós antes que hola.

PREGUNTA 2

```
int
main(int argc, char *argv[])
{
        int i;
        for(i=0;i<5;i++){
                if (!i)
                        break;
        }
        printf("%d\n",i);
        exit(EXIT_SUCCESS);
}
```

a) Imprime 11 valores.

b) Se queda en un bucle infinito.

c) No imprime nada.

d) Imprime 0.

e) No compila.

PREGUNTA 3

```
int
main(int argc, char *argv[])
{
        char p[5]="hola";
        int len;
```

```
        *p='z';
        p[2]=0;
        len=strlen(p);
        printf("%d\n",len);
        exit(EXIT_SUCCESS);
    }
```

a) Imprime 3

b) Imprime 1

c) Imprime 2

d) Compila, pero puede dar un error de compilación, P no es una string.

e) No imprime nada: no compila.

PREGUNTA 4

En un sistema de ficheros tipo Unix:

a) Una entrada del directorio relaciona un nombre del fichero con el i-nodo correspondiente, sólo puede haber un nombre para un i-nodo.

b) Un i-nodo contiene los metadatos de un fichero, como su nombre, permisos y las referencias a sus bloques de datos.

c) Usa en una tabla FAT para indexar los i-nodos con una lista enlazada con tabla

d) Un i-nodo contiene los datos de un fichero, una entrada de directorio contiene sus metadatos: su nombre, permisos, dueño, etc.

e) Una entrada del directorio relaciona un nombre del fichero con el i-nodo correspondiente, puede haber varios nombres para un i-nodo.

PREGUNTA 5

En Linux, el comando modprobe:

a) No existe tal comando

b) Sirve para compilar un módulo del Kernel.

c) Sirve para descargar un módulo del Kernel.

d) Sirve para enlazar con una biblioteca del sistema.

e) Sirve para cargar un módulo del Kernel.

PREGUNTA 6

```
int*
statusSon(int pid){
        int x;
        waitpid(pid,&x,0);
        x=WEXITSTATUS(x);
        return &x;
}
```

a) No compila: una función no puede devolver el tipo puntero a entero.

b) Es errónea: no existe la función waitpid para esperar a un hijo concreto.

c) Es correcta: aunque innecesaria por ser un recubrimiento de waitpid.

d) Es errónea: la función waitpid no devuelve el estatus del hijo.

e) Es errónea: Devuelve la dirección de una variable local.

PREGUNTA 7

```
cat /proc/$$/maps
```

a) Muestra las regiones de memoria del shell, y ese fichero se escribe en el disco cada cierto tiempo.

b) El comando cat da un error: No such file or directory.

c) Muestra las regiones de memoria del proceso que ejecuta cat. Y ese fichero no está en el disco.

d) Muestra las regiones de memoria del proceso que ejecuta cat. Y ese fichero es un fichero que se escribe en el disco cada cierto tiempo.

e) Muestra las regiones de memoria del Shell. Y ese fichero no esta en el disco.

PREGUNTA 8

Si tenemos un planificador Round-Robin y aumentamos mucho el tiempo de cuanto:

a) Se desperdicia mucho tiempo en cambios de contexto.

b) Degenera en una política SJF y provoca hambruna en los procesos dominados por CPU.

c) Degenera en la política FCFS y provoca un efecto convoy en los procesos dominados por entrada/salida.

d) No tiene sentido. Round-Robin no usa cuantos.

e) Se beneficia mucho a las aplicaciones interactivas.

PREGUNTA 9

Si el sistema está usando enlazado dinámico con lazy binding tengo un programa con una función fnt() definida...

a) La dirección de memoria de la función fnt se resuelve cuando gcc compila y enlaza el programa.

b) La dirección de memoria de la función fnt se resuelve justo antes de que comience la ejecución del programa.

c) La dirección de memoria de la función fnt no se resuelve hasta la primera vez que la se la llama.

d) La dirección de memoria de la función fnt se resuelve cuando el programa termina la ejecución.

e) No se pueden usar funciones en ese caso.

PREGUNTA 10

Cuando usamos un contenedor (p. ej. de Docker) en lugar de una máquina virtual de sistema para ejecutar un Red Hat Linux sobre un Ubuntu Linux...

a) Hay dos Kernels de Linux ejecutando: el del sistema Ubuntu y el del Red Hat.

b) Los contenedores sólo sirven para emular otro tipo de procesador (ISA).

c) No se necesita ejecutar ningún Kernel de Linux, todos los programas ejecutan sin sistema operativo.

d) Sólo hay un Kernel de Linux ejecutando.

e) Hay tres Kernels de Linux ejecutando: el Hypervisor (VMM), el del sistema Ubuntu y el del Red Hat.

PREGUNTA 11

```
int
main(int argc, char *argv[])
{
        fork();
        fork();
        printf("hola\n");
}
```

a) Falla en ejecución por no pasar un parámetro a fork().

b) Escribe una vez hola por su salida.

c) Escribe dos veces hola por su salida.

d) Escribe cuatro veces hola por su salida.

e) Escribe tres veces hola por su salida.

PREGUNTA 12

```
int
main(int argc, char *argv[])
{
        int *p;
        int i;
        int arr[10];
        for(i=0;i<10;i++)
                arr[i]=i;
        p=arr+2;
        p=&(p[1]);
```

```
        printf("%d\n",p[2]);
    }
```

a) Imprime un número cualquiera (accede a memoria sin inicializar).

b) No compila.

c) Imprime 0.

d) Imprime 5.

e) Imprime 3.

PREGUNTA 13

En general, un driver del sistema operativo ejecuta en...

a) Ring 2.

b) Ring -1.

c) Los drivers no necesitan ejecutar instrucciones.

d) Ring 3.

e) Ring 4.

PREGUNTA 14

```
    gcc -o x y.o
```

a) Compilar el fichero fuente del programa y crear un fichero objeto.

b) Compilar el fichero objeto y crear un fichero ejecutable.

c) Enlazar el fichero objeto con la librería de C y crear un fichero ejecutable.

d) Enlazar el código fuente del programa y crear un fichero objeto.

e) Enlazar el fichero fuente de la librería de C y crear un fichero objeto.

PREGUNTA 15

```
    int
    isXXX(unsigned int val){
        return val&1;
```

```
}
```

a) Devuelve true sí y solo sí recibe true (interpretando un booleano como un entero en C).

b) Devuelve siempre true (interpretando un booleano como un entero en C).

c) Devuelve si un número sin signo es par.

d) Devuelve siempre false (interpretando un booleano como un entero en C).

e) Devuelve si un número sin signo es impar.

PREGUNTA 16

```
int
main (int argc, char *argv[]){
        int p[5]={0,1,2,3,4};
        int i;
        for(i=0;i<sizeof(p); i++) {
        printf("%d\n",p[i]);
        }
        exit(EXIT_SUCCESS);
}
```

a) Está mal: sizeof da eñ tamaño en bytes, posiblemente imprima basura

b) Funciona y funcionará exactemente igual si p es un puntero en lugar de un array

c) Imprime valores de 0 a 5.

d) No compila, no se puede inicializar un array en la pila.

e) No funciona, le falta por inicializar un valor.

PREGUNTA 17

```
struct Coord{
        int x;
        int y;
```

```
};
typedef struct Coord Coord;
int main(int argc, char *argv[])
{
        int i;
        Coord *coords;
        coords=malloc(10*2*sizeof(int));
        for(i=0;i<10;i++){
                coords[i].x=i;
                coords[i].y=i;
                printf("(%d,%d)\n",coords[i].x,coords[i].y);
        }
}
```

a) Es incorrecto: puede desbordar el array.

b) Es correcto, imprime pares de números consecutivos.

c) No compila, la declaración debería ser struct Coord *.

d) No compila, lo que devuelve malloc no se puede asignar a coords.

e) Es correcto, imprime pares de números iguales.

PREGUNTA 18

```
enum{
        Origin=5
};
int main(int argc, char *argv[])
{
        int v;
        v=2;
        printf("%d\n",scale(v,7));
```

```
        exit(EXIT_SUCCESS);
    }
    static int scale (int val, int sc)
    {
        return (val-Origin)*sc;
    }
```

a) Compila, ejecuta e imprime -21.

b) No compila, se está usando scale fuera de ámbito.

c) Compila y funciona completamente.

d) Compila, ejecuta e imprime 21.

e) Compila, ejecuta e imprime 0.

PREGUNTA 19

Al ejecutar estos comandos, suponiendo que no hay ningún tipo de error en los 4 primeros comandos (pista: touch crea los ficheros):

```
cd /tmp/
mkdir x
cd x
touch fichero.txt fichero.TXT fichero.tXt
ls /tmp/x/*.[tT][xX][tT]
```

a) El comando ls tendrá un error: ` No such file or directory ´.

b) El comando ls listará tres ficheros.

c) El comando ls listará un fichero.

e) El comando ls no listará nada.

d) El comando ls listará dos ficheros.

PREGUNTA 20

Las siguientes sentencias imprimen:

```
int i;
i=0;
i+=4;
i--;
printf("%d\n",++i);
```

a) 4. b) 5. c) 0 d) 2 e)3

TEST1 - NOVIEMBRE 2020-2021

PREGUNTA 1

Dado el siguiente fragmento de código

```
struct Patata{
        int x;
        int y;
};
typedef struct Patata Patata;
Patata*
new_patata(int x,int y)
{
        Patata p;
        p.x=x;
        p.y=y;
        return &p;
}
int
main(int argc, char *argv[])
{
        Patata *s;
        s=new_patata(3,4);
```

```
        printf("s.x:%d\n",s->x);
    }
```

a) Dará un warning, pero funcionará la ejecución

b) Está bien, pero falta un malloc después de la llamada a new_patata (la función está bien)

c) Es totalmente correcto, porque no hay ningún error de tipos

d) Hay un error grave, y posiblemente dé un segmentation fault en ejecución

e) Está bien, pero falta un free después del printf

PREGUNTA 2

El programa, suponiendo que los includes están bien

```
    int
    main(int argc, char *argv[]){
        char *p="hola";
        char z[5]={'h','o','l','a','\0'};
        if (strcmp(p,z)){
            printf("son iguales");
        }
        exit(EXIT_SUCCESS);
    }
```

a) No imprime son iguales: las cadenas son diferentes

b) Imprime son iguales a pesar de que las cadenas son diferentes

c) No compila, no se puede comparar así un array y un puntero a char.

d) No imprime son iguales, a pesar de que las dos cadenas son iguales.

e) Imprime son iguales, las cadenas son iguales.

PREGUNTA 3

El siguiente código

execl(“/bin/ls”,”ls”,”/tmp”,NULL);

a) Está mal, sobra el ultimo parámetro, NULL

b) Está perfecto si queremos ejecutar un ls /tmp

c) Está mal, debería ser execv

d) Está bien si y solo si se están creando hilos con pthreads

e) Está mal, antes de un exec siempre tiene que haber un fork.

PREGUNTA 4

La variable de entorno $PATH contiene:

a) La ruta actual del Shell

b) Normalmente no existe esa variable de entorno en un sistema UNIX

c) Las rutas de los directorios donde se buscarán los ejecutables, separados por “:”

d) Las rutas de los HOME de los usuarios del sistema, separadas por “:”

e) Las rutas de los directorios donde se buscarán las bibliotecas, separadas por “:”

PREGUNTA 5

Sin una modificación en el núcleo hace que los cambios de contexto tarden más, en cuál de las siguientes políticas tendrá mas impacto

a) FCFS

b) SJF

c) Afecta a todas estas políticas por igual

d) Round-Robin con cuanto pequeño

e) Round-Robin con cuanto grande

PREGUNTA 6

Un programa lee una variable de entorno así:

x=getenv(“$HOME”);

a) Debería haber utilizado “HOME” como parámetro.

b) Debería utilizar el procedimiento getmyhome()

c) Debería haber utilizado “home” como parámetro.

d) Debería haber utilizado “$HOME” como parámetro.

e) Debería haber utilizado $HOME, sin comidas, como parámetro.

PREGUNTA 7

El siguiente código:

```
s=strdup(“bien”);
s=strcat(s,”bien”);
```

a) Reserva memoria correctamente gracias a strdup

b) Si se sustituye strcat por strncat sería correcto.

c) No concatena correctamente las dos cadenas, ”bien” y “bien” formando “bienbien”

d) Concatena correctamente las dos cadenas, ”bien” y “bien” formando “bienbien”

e) Si tuviese un free al final estaría perfecto

PREGUNTA 8

La función

```
int*
statusSon(int pid)
{
        int x;
        waitpid(pid,&x,0);
        x=WEXITSTATUS(x);
        return &x;
}
```

a) Es correcta, aunque innecesaria por ser un recubrimiento de waitpid

b) Es errónea: devuelve la dirección de una variable local

c) Es errónea: la función waitpid no devuelve el status del hijo.

d) No compila: una función no puede devolver el tipo puntero a entero.

e) Es errónea: no existe la función waitpid para esperar aun hijo concreto

PREGUNTA 9

```
struct Patata{
        int x;
        int y;
};
typedef struct Patata Patata;
Patata*
new_patata(int x,int y)
{
        Patata *p;
        p=malloc(sizeof(Patata));
        p->x=x;
        p->y=y;
        return p;
}
int
main(int argc, char *argv[])
{
        Patata *s;
        s=malloc(sizeof(Patata));
        s=new_patata(3,4);
        printf("%d\n",s->x);
        free(s);
}
```

a) Nunca imprimirá 3.

b) Está bien programado y compilará y ejecutará perfectamente

c) Tiene un error porque el campo está sin inicializar

d) Tiene un fallo que se arregla poniendo otro free extra tras la llamada a new_patata

e) Tiene un leak, reserva memoria innecesaria.

PREGUNTA 10

El siguiente comando de Shell

```
test -f /tmp/a || echo x
```

a) Nunca escribe X en la salida

b) Escribe X en la salida si el fichero /tmp/a existe

c) Escribe X en la salida siempre

d) Escribe X en la salida si existe la variable de entorno /tmp/a

e) Escribe X en la salida si el fichero /tmp/a no existe

PREGUNTA 11

La siguiente función:

```
void
x(void){
        fork();
        fork();
        printf("%d\n",getppid());
        exit(EXIT_SUCCESS);
}
```

a) Escribe cuatro números por su salida en el mismo orden, siempre todos los números serán distintos.

b) Escribe cuatro números por su salida, siempre todos los números serán distintos

c) Escribe tres números por su salida, siempre todos los números serán distintos.

d) Escribe cuatro numero por salida en el mismo orden, siempre dos números serán iguales y otros dos no

e) Escribe cuatro números por su salida, siempre dos números serán iguales y otros dos no

PREGUNTA 12

```
int
main(void) {
        char p[]={};
        int i;
        for(i=0;i<4;i++){
                p[i]='a';
        }
        for(i=0;i<4;i++){
                printf("%c",p[i]);
        }
        exit(EXIT_SUCCESS);
}
```

a) Es totalmente correcto, no imprimirá nada

b) Es incorrecto porque sobra el inicializar del array

c) Es incorrecto, tiene problemas de memoria

d) Es totalmente correcto, imprimará valores a cero

e) Es totalmente correcto e imprimirá cuatro aes

PREGUNTA 13

El siguiente código

```
execv(“/bin/ls”,”ls”,”/tmp”,NULL);
```

a) Está perfecto

b) Está mal, no hay ninguna función/llamada al sistema llamada execv.

c) Está mal, sobra el último parámetro NULL

d) Está mal, antes de un exec siempre tiene que haber un fork.

e) Está mal, debería ser un execl

PREGUNTA 14

Si tengo la siguiente sentencia en C, que es correcta:

```
x=pal->fin->x;
```

a) La variable pal tiene que contener un puntero a una struct

b) La variable pal no tiene por qué ser un puntero

c) la variable x tiene que ser un puntero

d) El campo fin no puede ser un puntero

e) El campo fin no puede ser un puntero a struct

PREGUNTA 15

El siguiente código:

```
fork();
fork();
printf("hey\n");
```

a) Escribe 4 líneas con el texto "hey" en su salida

b) Escribe 1 línea con el texto "hey" en su salida

c) Escribe 2 líneas con el texto "hey" en su salida

d) Escribe 5 líneas con el texto "hey" en su salida

e) Escribe 3 líneas con el texto "hey" en su salida

PREGUNTA 16

```
int
main(void)
```

```
{
        int a;
        int b;
        a=3;
        b=++a;
        b-=2;
        a=b++;
        printf("%d",a);
}
```

a) No compila

b) Imprime 1

c) Imprime 3

d) Imprime 2

e) Imprime 0

PREGUNTA 17

Al final del siguiente código

```
int
main(void) {
        unsigned char c;
        c=1;
        c=(c<<2)+1;
        c=c&0;
}
```

a) Al final c está sin inicializar

b) Al final en c hay un 7

c) Al final en c hay un 4

d) Al final en c hay un 6

e) No compila, no se puede mezclar unsigned char e int

PREGUNTA 18

Si tenemos un planificador Round-Robin y aumentamos mucho el tiempo de cuanto:

a) Degenera una política SJF y provoca hambruna en los procesos dominados por la CPU

b) No tiene sentido, Round-Robin no usa cuantos

c) Degenera en la política FCFS y provoca un efecto convoy en los procesos dominados por entrada/salida

d) Se desperdicia mucho tiempo en cambios de contexto

e) Se beneficia mucho a las aplicaciones interactivas

PREGUNTA 19

Si el sistema está usando enlazado dinámico con lazy binding y tengo un programa con una función fnt() definida

a) La dirección de memoria de la función fnt no se resuelve hasta la primera llamada

b) La dirección de memoria de la función fnt se resuelve cuando gcc compila y enlaza el programa

c) La dirección de memoria de la función fnt se resuelve cuando el programa termina la ejecución.

d) No se puede usar funciones en ese caso

e) La dirección de memoria de la función fnt se resuelve justo antes de que comience la ejecución del programa

TEST1 - NOVIEMBRE 2021-2022

PREGUNTA 1

El siguiente comando:

```
$ ldd /bin/cat
        linux-voso.so.1[0x00007ff082f9c0001
        libc.so.6 => /lib/x86 64-linux-anu/libc.so.6 (0x00007f2f1eec0000)
```

/lib64/ld-linux-x86-64.so.2 (0x00007f2f1f0dd000)

a) Muestra las bibliotecas dinámicas que necesita el script /bin/cat

b) Te elimina la GOT, la PLT y la tabla de símbolos del binario ELF, que no se necesitan para ejecutar

c)Muestras las bibliotecas estáticas para las que se hará lazy binding

d) Muestra las bibliotecas dinámicas que necesita el binario ELF /bin/cat

e) Muestra las bibliotecas estáticas que necesita el binario ELF /bin/cat

PREGUNTA 2

Las llamadas al sistema sirven para ...

a) Que el kernel pueda ser microkernel

b) Que el kernel pida servicio a los programas de usuario

c) Que el kernel pueda ser monolítico

d) Que el kernel pueda comunicarse con el gestor de máquinas virtuales

e) Que los programas de usuario pidan servicios al kernel

PREGUNTA 3

En un planificador de tipo Round-Robin:

a) Cuando un proceso agota su cuanto se le notifica y más tarde realiza una llamada a yield

b) Un proceso solo puede agotar su cuanto cuando está ejecutando una llamada al sistema

c) Cuando un proceso agota su cuanto se le expulsa

d) Cuando un proceso agota su cuanto hace una llamada al sistema para que se le expulse

e) Un proceso tiene que llamar a yield para ver si hay que expulsarle

PREGUNTA 4

Al ejecutar el siguiente código en un terminal

```
if (fork() == 0) {
        fprintf(stderr,”A”);
}
fprintf(stderr,”B”);
if (fork() == 0) {
        fprintf(stderr,”C”);
}
fprintf(stderr, "D");
```

a) Se imprime una A

b) Se imprimen: dos A, tres B, seis C y siete D, no sabemos el orden

c) Se imprimen: una A, dos B, dos C y cuatro D, en ese orden

d) Se imprimen: una A, cuatro B, cuatro C y ocho D, en ese orden

e) Se imprimen: una A, dos B, dos C y cuatro D, no sabemos el orden

PREGUNTA 5

Si ejecutamos los siguientes comandos en un terminal (el $ al inicio de línea es el prompt):

```
$ mipid=20
$ bash
$ echo “$mipid”
```

a) No escribe nada por su salida

b) Da un error, ya que las variables de entorno siempre tienen que estar en mayúsculas

c) Escribe 20 por su salida

d) Escribe por su salida un dólar seguido de una letra n

e) Escribe por su salida el PID (identificador de proceso) del último proceso ejecutado en la shell

PREGUNTA 6

Cuando usamos un contenedor (p. ej: de Docker) en lugar de una máquina virtual de sistema para ejecutar un Red Hat sobre un Ubuntu Linux ...

a) Hay dos kernels de Linux ejecutando: el del sistema Ubuntu y el del Red Hat

b) No se necesita ejecutar ningún kernel de Linux, todos los programas ejecutan sin sistema operativo

c) Los contenedores sólo sirven para emular otro tipo de procesador (ISA)

d) Sólo hay un kernel de Linux ejecutando

e) Hay tres kernels de Linux ejecutando: el hipervisor (VMM), el del sistema Ubuntu y el del Red Hat

PREGUNTA 7

Suponiendo que /bin/Is existe y tenemos permisos para ejecutarlo, el siguiente código:

```
execl("/bin/ls", "ejecuto ls", “SHOME", NULL);
printf ("hey");
```

a) Veremos hey en la salida, porque exec fallará: el NULL del final es erróneo

b) No veremos hey en la salida, pero no se listarán las entradas de nuestro directorio casa

c) Se creará un proceso hijo que listará las entradas de nuestro directorio casa y el proceso padre escribirá hey por la salida

d) Veremos hev en la salida, porque exec fallará: el primer argumento que se le pasa provocará un error

e) No veremos hev en la salida. veremos un listado de las entradas de nuestro directorio casa

PREGUNTA 8

Cuál de las siguientes afirmaciones es falsa:

a) Todos los hijos de un proceso heredan en su creación una variable de entorno modificada por el padre

b) El entorno de un proceso comienza estando en su pila.

c) Execv deja una copia idéntica del entorno que había antes de ejecutarlo.

d) Se puede programar un comando que no es builtin que cambie una variable de entorno de la shell.

e) Se puede programar un builtin de la shell que cambie una variable de entorno de la shell.

PREGUNTA 9

Un módulo del kernel...

a) Es una parte del núcleo que se puede cargar dinámicamente en tiempo de ejecución. Es código que ejecuta en área de usuario.

b) Es una parte del núcleo que se puede añadir o quitar estáticamente cuando se compila el kernel, pero no después. Es código que ejecuta en área de kernel.

c) Es una parte del núcleo que se puede cargar y descargar dinámicamente en tiempo de ejecución. Es código que ejecuta en área de kernel.

d) Es un tipo de contenedor, una forma de virtualización, que ejecuta en área de hipervisor

e) Es una parte del núcleo que se puede añadir o quitar estáticamente cuando se compila el kernel, pero no después. Es código que ejecuta en área de usuario.

PREGUNTA 10

El siguiente programa principal:

```
int
main(int argo, char *argv[]) {
        fork();
        fork();
        printf ("hola\n");
        exit(EXIT_SUCCESS);
}
```

a) Escribe una vez hola por su salida

b) Escribe tres veces hola por su salida

c) Falla en ejecución por no pasar un parámetro a fork()

d) Escribe dos veces hola por su salida

e) Escribe cuatro veces hola por su salida

PREGUNTA 11

EI comando renice de Linux..

a) Permite cambiar el cuanto (niceness) de la política Round-Robin

b) Permite cambiar la prioridad de los procesos de dinámica a estática y activar el aging.

c) Permite cambiar la arquitectura de multiprocesador a NUMA

d) Permite cambiar la prioridad (niceness) de un proceso, con valores entre -20 y 19

e) Permite cambiar la prioridad (niceness) de un proceso, con valores entre 0 y 100

PREGUNTA 12

Si una modificación en el núcleo hace que los cambios de contexto tarden más, en cuál de las siguientes políticas tendrá más impacto:

a) Afecta a todas estas políticas por igual

b) SIF

c) FCFS

d) Round-Robin con cuanto grande

e) Round-Robin con cuanto pequeño

PREGUNTA 13

Marca la respuesta incorrecta. Un fichero ELF tiene dentro información sobre..

a) El enlazado T

b) El punto de entrada del programa (por donde comienza a ejecutar)

c) El tipo de planificador que debe usar el kernel para ejecutar el programa

d) El tipo de arquitectura necesario para ejecutar las instrucciones del programa

e) Los datos inicializados del programa (variables globales inicializadas)

PREGUNTA 14

Para llamar a exec,

a) Siempre hay que hacer fork, si no da un error la llamada: command not found

b) Lo normal es hacer fork antes, para ejecutar en un proceso hijo y poder controlar lo que sucede

c) Antes hay que llamar a wait

d) Hay que concatenar todos los argumentos en una sola string con el último carácter a '&'

e) Hay que pasarle al programa al menos dos argumentos

PREGUNTA 15

El siguiente código:

```
if ((wait &s) != -1){
        if (s == 1) {
                printf("el hijo murio con fallo");
        }else{
                printf ("el hijo murio con exito");
        }
}
```

a) Está mal, debería usar siempre waitpid para esperar por un hijo

b) Esta mal, si no hay hijo por el que esperar, se queda bloqueado de por vida

c) Imprime si el hijo ha salido con éxito o no bien

d) Está mal, el hijo ha podido salir con fallo y que se imprima lo contrario

e) Está mal, en la variable s tendremos el PID del hijo, no su status de salida

PREGUNTA 16

Cuando una shell ejecuta la línea de comando:

```
echo $USER
```

a) El proceso de echo sustituye el valor de la cadena $USER por el valor de la variable de entorno USER.

b) El proceso de la shell sustituye la cadena $USER por el valor de la variable de entorno USER.

c) El valor de USER lo sustituye echo en el procesado de argumentos.

d) El comando echo sólo escribe el nombre del usuario a veces, porque el dólar no está escapado.

e) La variable USER forma parte del globbing de la shell, por tanto, no es una sustitución.

PREGUNTA 17

En un sistema de tipo FAT:

a) Los nombres de ficheros están en los clusters de datos de los ficheros.

b) Las entradas de directorio están en la tabla FAT.

c) Los nombres de ficheros y directorios están en entradas de directorio.

d) Los nombres de directorios están en los clusters de datos de los ficheros

e) Los nombres de ficheros y directorios no están en los bloques de datos de los directorios

PREGUNTA 18

El siguiente comando ($ es el prompt):

```
$ echo [0-9]*.q
```

a) Escribe por su salida, separados por un espacio, los nombres de todos los ficheros/directorios del directorio actual que comienzan por un número y terminan con ".q". Si no hay ninguno con un nombre así, no escribe nada

b) Escribe por su salida, separados por un espacio, los nombres de todos los ficheros/directorios del directorio actual que comienzan por algo que no sea un número y terminan con ".q”. Si no hay ninguno con un nombre así, no escribe nada

c) Escribe por su salida, uno por línea, los nombres de todos los ficheros/directorios del directorio actual que comienzan por un número y terminan con ".q". Si no hay ninguno con un nombre así, escribe [0-9]*.q

d) Escribe por su salida, separados por un espacio, los nombres de todos los ficheros/directorios del directorio actual que comienzan por un número y terminan con ".q". Si no hay ninguno con un nombre así, escribe [0-9]*.q

e) Escribe por su salida, uno por línea, los nombres de todos los ficheros/directorios del directorio actual que comienzan por un número y terminan con ".q". Si no hay ninguno con un nombre así, no escribe nada

PREGUNTA 19

Queremos reservar espacio para guardar en un array 200 strings en C cuya memoria (la memoria de las strings) ya está reservada. La sentencia es:

a) malloc(200);

b) malloc(200*sizeof(char));

c) malloc(200*MaxStr*sizeof(char*));

d) malloc(200*sizeof(char *));

e) malloc(200*sizeof(char **);

PREGUNTA 20

En una máquina virtual con paravirtualización..

a) El sistema operativo huésped necesita tener un microkernel

b) El sistema operativo huésped necesita modificaciones

c) No puede virtualizar un sistema operativo, únicamente puede virtualizar un proceso

d) Necesita ejecutar sobre una CPU con instrucciones especiales de virtualización

e) El sistema operativo huésped no necesita modificaciones

TEST1 - NOVIEMBRE 2022-2023

PREGUNTA 1

```
struct Patata{
```

```
        int x;
        int y;
};
typedef struct Patata Patata;
Patata*
new_patata(int x,int y)
{
        Patata *p;
        p=malloc(sizeof(Patata));
        p->x=x;
        p->y=y;
        return p;
}
int
main(int argc, char *argv[])
{
        Patata *s;
        s=new_patata(0,0);
        s=malloc(sizeof(Patata));
        printf("%d\n",s->x);
        free(s);
}
```

a) Está bien, pero falta un malloc después de la llamada a new_patata (la función está bien).

b) Funcionará en ejecución, pero tiene un leak de memoria.

c) Está bien, pero falta un free después del printf

d) Es totalmente correcto, porque no hay ningún error de tipos

e) Seguramente dará un segmentation fault en ejecución

PREGUNTA 2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int
main(int argc, char *argv[])
{
        char p[5]="hola";
        int len;
        *p='z';
        p[2]=0;
        len=strlen(p);
        printf("%d\n",len);
        exit(EXIT_SUCCESS);
}
```

a) Imprime 1

b) Imprime 2

c) Compila, pero puede dar un error de ejecución, p no es un string

d) No imprime nada, no compila

e) Imprime 3

PREGUNTA 3

Cuando escribo en la Shell un comando acabado por ampersand &

a) El comando se ejecuta en primer plano

b) Solo se escribe el prompt si el comando terminó con exito

c) Imprime el prompt inmediatamente

d) La shell hace wait por el comando

e) Solo se escribe el prompt si el comando terminó con fallo.

PREGUNTA 4

El Path

/tmp/../../../../etc/./.home/./paurea/fichero/.

a) Equivale a /home/paurea/

b) Equivale a /etc/home/paurea/fichero

c) Equivale a /

d) No tiene sentido

e) Equivale a /etc/fichero

PREGUNTA 5

Si escribo ls *.[abc]ef?

a) Puede imprimir a.cefg a.cefh otro.aefy

b) Puede imprimir a.cefg ce.cfh otro.tefy

c) Puede imprimir yyy.cefg ce.cefh otro.adfy

d) Puede imprimir a.cefg mas.cefh otro.tefy

e) Puede imprimir a.cefgr ce.cefh otro.aefy

PREGUNTA 6

La variable de entorno PATH contiene:

a) Directorios donde se buscarán lo comandos escritos como paths absolutos y que empiecen por punto.

b) Directorios donde se buscarán lo comandos escritos como paths relativos y que no empiecen por punto.

c) Directorios donde se buscarán lo comandos escritos como paths relativos únicamente.

d) Directorios donde se buscarán lo comandos escritos como paths relativos y que empiecen por punto.

e) Directorios donde se buscarán lo comandos escritos como paths absolutos.

PREGUNTA 7

¿Qué afirmación es cierta sobre un fichero en texto plano?

a) Cada byte representa un carácter

b) Codificará las fonts para pintar los caracteres en su interior

c) No puede contener caracteres que no se puede ver o imprimir (entonces sería binario)

d) Hay que interpretarlo dependiendo de la codificación.

e) Puede ser de tipo ELF

PREGUNTA 8

Para llamar a exec

a) Hay que pasarle programa al menos dos argumentos

b) Lo normal es hacer fork antes, para ejecutar en un proceso hijo y poder controlar lo que sucede

c) Antes hay que llamar a wait

d) Siempre hay que hacer fork, sino da un error la llamada: command not found.

e) Hay que concatenar todos los argumentos en una sola string con el ultimo carácter a &

PREGUNTA 9

Las variables de entorno.

a) Son lo mismo que las variables de Shell

b) Las heredan todos los procesos de su proceso padre

c) Se usan solo en la Shell para hacer sustituciones

d) Son las mismas entre todos los procesos, si uno las modifica, los otros ven el resultado

e) Se pasan como parámetros a los procesos en el argv

PREGUNTA 10

En linux, ejecuta en modo privilegiado (ring 0):

a) Todos los procesos de root

b) Ninguno de los drivers.

c) El Kernel

d) Los procesos que se ejecutan con sudo.

e) Algunas llamadas al sistema y otras no.

PREGUNTA 11

En la pila hay:

a) Parte del BSS (la que no está en el heap).

b) Todas las variables sin inicializar y registros de activación.

c) Registros de activación que contienen, entre otras cosas, variables locales.

d) Todas las variables que no están en el heap.

e) Variables locales que siempre están inicializadas a cero.

PREGUNTA 12

```
int
main(int argc, char *argv[]) {
        int i;
        argc--;
        for(i=0;i<argc;i++){
                printf("%s\n", argv[i]);
        }
        return 0;
}
```

Si ejecutamos

```
$> programa uno dos tres
```

a) Imprime dos líneas: dos, tres

b) Imprime tres líneas: programa, uno, dos

c) No puede decrementar argc.

d) main nunca puede retornar.

e) Hace exactamente lo mismo que el comando echo.

PREGUNTA 13

Supongamos que hago lo siguiente en un programa en C:

```
void *p;
char *s;
int i;
p=malloc(100*sizeof(char*));
s=(char *)p;
for(i=0;i<100;i++){
        s[i]='c';
}
for(i=0;i<100;i++){
        printf("el char vale %c\n",s[i]);
}
```

a) El cast es incorrecto y va a dar un error de compilación.

b) Es posible que compile y ejecute, pero es incorrecto y reserva más de la memoria necesaria.

c) Va a dar un error de tipos, intenta usar un puntero a void como puntero a char sin cast

d) Va a dar un error de tipos, reserva un puntero a char y lo mete en un puntero a void

e) Es totalmente correcto y reserva la cantidad de memoria necesaria exacta.

PREGUNTA 14

El operador ->

a) No funciona con punteros a struct

b) Atraviesa un puntero y accede al indice de un array

c) Es azúcar sintáctico, le suma 1 al puntero

d) Atraviesa un puntero y accede al campo de un record.

e) Solo funciona sobre punteros a entero.

PREGUNTA 15

Si un usuario presiona en una shell la combinación de teclas Ctrl-z (control y z)

a) Mata al proceso que esta ejecutando en foreground.

b) Busca un comando que se ha ejecutado recientemente.

c) Limpia el terminal.

d) Completa el nombre de un fichero.

e) Se detiene el proceso que está ejecutando en foreground

PREGUNTA 16

```
int a;
int b;
a=3;
b=++a;
b-=2;
a=b++;
printf("%d",a);
```

a) Imprime 2 b) Imprime 1 c) Imprime 3 d) Imprime 0 e) No compila

PREGUNTA 17

```
p=(Patata *)malloc(sizeof(Patata));
p=m;
p->lavada=0;
```

a) Esta mal porque debería ser sizeof(Patata *) para que el resto tuviese sentido

b) Podría tener sentido si se hace free de la memoria que contiene a p

c) Está perfecto si m es de tipo puntero a Patata.

d) Nunca tendrá sentido, es un leak

e) Sólo estará bien si hay un free más adelante.

PREGUNTA 18

Un programa lee una variable de entorno así (para obtener el directorio casa de un usuario):

```
x=getenv("$PATH");
```

a) Debería haber utilizado "$HOME" como parámetro

b) Debería haber utilizado "HOME" como parámetro.

c) Debería haber utilizado $HOME sin comillas, como parámetro.

d) Debería utilizar el procedimiento getmyhome();

e) Debería haber utilizado "home" como parámetro.

PREGUNTA 19

La siguiente función

```
void
x(void) {
        fork();
       fork();
       printf("%d\n",getppid());
       exit(EXIT_SUCCESS);
}
```

a) Escribe cuatro números por su salida en el mismo orden, siempre todos los números serán distintos

b) Escribe cuatro números por su salida, siempre dos números serán iguales y los otros dos no.

c) Escribe tres números por su salida, siempre todos los números serán distintos.

d) Escribe cuatro números por su salida, siempre todos los números serán distintos

e) Escribe cuatro números por su salida en el mismo orden, siempre dos números serán iguales y otros dos no.

PREGUNTA 20

La siguiente función

```
enum{
        MAXUSER=128
};
char *
setuser(char *readuser)
{
        char user[MAXUSER];
        strcpy(user,readuser);
        fprintf(stderr,"El Usuario es %s\n",user);
        return user;
}
```

a) Incorrecta: no inicializa user, pero el resto es correcto

b) No compila: readuser es de un tipo Incorrecto para strcpy (no es una string).

c) Es incorrecta: retorna un puntero a la pila.

d) No compila: user es de un tipo incorrecto para strcpy (no es una string)

e) Es correcta

TEST1 - ENERO 2018-2019

PREGUNTA 1

La mascara de señales de un proceso:

Seleccione una:

a) Indica que señales ignora el proceso, que nunca se podrán entregar

b) Indica qué manejadores de señales tiene implementado el programa

c) Es otra forma de llamar a la tabla de descriptores de fichero del proceso.

d) Indica que señales tiene bloqueadas el proceso, que pueden estar pendientes de entrega

e) Indica si los ficheros son ficheros convencionales, directorios o fifo

PREGUNTA 2

La TLB (Translation Look-aside buffer):

Seleccione una:

a) Es una memoria cache que acelera la traducción de dirección virtual a dirección física

b) Es una memoria cache donde se guardan los datos de los procesos

c) Es una memoria cache que acelera la traducción de dirección física a dirección virtual

d) Es una memoria cache donde se guardan las instrucciones de los procesos

e) Es una memoria cache que acelera la traducción de número de página a ruta de un fichero

PREGUNTA 3

Es un sistema de paginación en demanda y overcommitment, suponiendo que las páginas son de 4KB, se ejecuta el siguiente programa, ¿Cuántos marcos de página usará el proceso para sus datos(variables globales)

```
char a[512*4*1024];

int
```

```
main(int argc, char *argv[])
{
        a[0]='a';
}
```

Seleccione una:

a) 2048 marcos de página

b) 512 marcos de página

c) Ninguno, no tiene datos

d) 1 marco de página

e) 2097152 marcos de página

PREGUNTA 4

El siguiente comando de Shell

```
test -f /tmp/a || echo x
```

a) Escribe X en la salida si existe la variable de entorno /tmp/a

b) Nunca escribe X en la salida

c) Escribe X en la salida si el fichero /tmp/a no existe

d) Escribe X en la salida siempre

e) Escribe X en la salida si el fichero /tmp/a existe

PREGUNTA 5

¿Cuál de las siguientes expresiones regulares no encaja con esta línea?

El perro de San Roque no tiene rabo porque Ramon Ramirez se lo ha cortado.

a) ^El.*\.$

b).*

c) .*S.*

d) ^[A-Za-z]+\.$

e) ^[a-z]+o.$

PREGUNTA 6

La memoria virtual:

a) Es inútil hoy en día, ya no se usa

b) Complica la depuración de los programas, pero facilita la compresión

c) Solo es interesante porque permite usar memoria secundaría como swap

d) Permite que el planificador implemente una política Round Robin

e) Permite proteger la memoria de los distintos procesos.

PREGUNTA 7

Distintos procesos que comparten memoría llamaran a la siguiente función en distintas ocasiones. La variable contador es una variable global compartida por todos los procesos. Dicho contador nunca debería llegar a ser negativo. La variable mutex es un cierre compartido por todos los procesos, usado para proteger la variable contador

```
        void
        decrementarhastacero(void)
        {
                if (contador > 0){
                        pthread_mutex_lock(&mutex);
5:                      contador--;
                        pthread_mutex_unlock(&mutex);
                }
        }
```

a) Está mal, primero se debe de llamar a unlock, y después a lock

b) Está mal, desde dentro de una función no se puede usar un cierre

c) Parece que está bien

d) Hay una condición de carrera, dos procesos pueden ejecutar la línea 5 concurrentemente

e) Hay una condición de carrera, el contador puede llegar a ser negativo.

PREGUNTA 8

La llamada al sistema mlock

a) Permite evitar condiciones de carrera cuando queremos usar una variable compartida

b) Permite evitar que la memoria de un proceso vaya almacenamiento secundario (swap)

c) Permite evitar condiciones de carrera cuando varios procesos usan un fichero a la vez

d) No existe

e) Permite crear un log para almacenar los avisos del programa

PREGUNTA 9

El siguiente código:

```
d=opendir("/tmp");
if (d==NULL)
        err(EXIT_FAILURE,"opendir failed");
while((ent=readdir(d))!=NULL){
        printf("%s\n",ent->d_name);
}
Closedir(d);
```

a) Escribe por la salía el nombre de todas las entradas del directorio /tmp, una por línea

b) Falla siempre porque hay que abrir el directorio con open, no con opendir

c) Escribe por la salida el nombre de todos los ficheros convecionales del directorio /tmp, uno por línea

d) Escribe por la salida el nombre de todos los directorios del directorio /tmp, uno por linea

PREGUNTA 10

La diferencia entre las funciones read y fread es:

a) read hace buffering, fread no hace buffering

b) Las dos hacen buffering, pero fread no sirve para leer de un pipe

c) read no hace buffering, fread si hace buffering

d) Ninguna hace buffering, pero fread solo sirve para leer structs

e) Las dos hacen buffering, pero fread solo sirve para leer structs

PREGUNTA 11

Tenemos un asignador dinámico de memoria Quick Fit que usa trozos de dos tamaños, 128 y 256 bytes, para reservas pequeñas. Para reservas mayores de 256 bytes, usa una estrategia Best Fit. Dado el siguiente código que usa dicho asignador, en el que la función enlaza mete el nodo en una lista enlazada

```
typedef struct Node Node;
struct Node{
        Node *next;
        int num;
}
for(i=0;i<1024*1024;i++){
        Node *n=malloc(sizeof(Node));
        n->num=i;
        enlaza(n);
}
```

a) Probablemente estaremos perdiendo mucha memoria por fragmentación externa

b) Parece buena elección, probablemente no malgastemos mucha memoría

c) Probablemente estaremos perdiendo mucha memoría por fragmentación interna

d) No tiene sentido, no existe ninguna política llamada Quick Fit

e) No perderemos memoría, pero debería usar la política NRU en lugar de Best Fit

PREGUNTA 12

La llamada mmap

a) Permite crear nuevas regiones de memoria de un proceso y proyectar ficheros en memoria

b) Permite trazar con mapas de bits de forma rápida y sencilla

c) Reemplaza totalmente a las llamadas open, read, write y close

d) Permite crear un mapa de traducción entre direcciones virtuales y direcciones físicas

e) Funciona especialmente bien para proyectar ficheros sintéticos y ficheros en read

PREGUNTA 13

La señal SIGKILL(9)

a) Para un proceso, y puede ser ignorada. Es la que envía al proceso al hacer Ctrl+z en el terminal

b) Mata un proceso y no puede ser ignorada. Es la que envía al proceso al hacer Ctrl+z en el terminal

c) Mata un proceso, pero puede ser ignorada. Es la que se envía al proceso al hacer Crtl+c en el terminal

d) Mata un proceso, pero puede ser ignorada. No es la que envía al proceso al hacer Crtl+c en el terminal

e) Mata un proceso y no puede ser ignorada. No es la que se envía al proceso al hacer Crtl+c en el terminal

PREGUNTA 14

El siguiente script

```
#!/bin/bash
for i in $*
do
        echo $1
        shift
done
```

a) Da un error de sintaxis en el for

b) Da un error, no hay ningún comando que se llame shift

c) Imprime una línea con cada uno de los argumentos que se le pasan al script

d) Imprime tantas líneas como argumentos se le pasan al script, pero siempre imprime el mismo argumento

e) Imprime líneas todo el rato, es un bucle infinito

PREGUNTA 15

Cuando se ejecuta en Shell el siguiente comando

```
ls -l / | grep -v usr | wc -l
```

a) Crea dos pipes y tres procesos hijos. El Shell lee y escribe los pipes para comunicar procesos

b) Crea dos pipes y tres procesos: un hijo, un nieto y un bisnieto. El Shell no lee ni escribe en ningún pipe

c) Crea dos pipes y tres procesos hijos. El Shell lee del ultimo pipe para escribir en el terminal de salida

d) Crea dos pipes y tres procesos hijos. El Shell no lee ni escribe en ningún pipe

e) Ese comando es incorrecto, el Shell no crea nada.

PREGUNTA 16

Si el directorio de trabajo de un proceso que ejecuta este código es /tmp, suponiendo que no hay ningún problema de permisos y la llamada open no falla

```
fd=open("./afile",O_WRONLY | O_CREAT | O_TRUNC,0640);
dup2(fd,1);
```

a) Creará o truncará el fichero /afile, y será su entrada estándar

b) Creará o truncará el fichero /tmp/afile, y será su salida estándar

c) Creará o truncará el fichero /afile, y será su salida estándar

d) Creará o truncará el fichero /tmp/afile, y será su entrada estándar

e) Fallará el dup2 porque el modo de apertura no es compatible con la entrada estándar

PREGUNTA 17

El siguiente comando:

```
echo uno dos tres | awk '{print $3 $2 $1}' | sed -E 's/u.o/xxx/g' 2> /dev/null
```

a) Escribe por su salida tresdosuno

b) Escribe por su salida tresdosxxx

c) Escribe por su salida unodostres

d) Escribe por su salida unodosxxx

e) No escribe nada

PREGUNTA 18

```
int fd;
char buf[Bufsize];
int nr;
while (1){
        nr=read(fd,buf,Bufsize);
        if (nr==0)
                err(EXIT_FAILURE,"error en la lectura");
        if (write(1,buf,nr)!=nr)
                err(EXIT_FAILURE,"error en la escritura");
}
```

a) Está mal porque no se puede hacer un write en el descriptor 1

b) Está perfecto, escribe el fichero completo en la salida y termina con exito.

c) Está mal, porque no se comprueba correctamente el error del read

d) Está mal porque es un bucle infinito

e) Está mal porque no se comprueba correctamente el error del write

TEST2 - FEBRERO 2020-2021

PREGUNTA 1

```
#include <stdio.h>
int
newval(int z)
{
        return ++z;
}
int
main(int argc, char *argv[]){
        int x;
        x=3;
        printf("%d",newval(x--));
}
```

a) Imprime 3

b) Imprime -1 porque hay un error

c) Imprime 4

d) No compila porque modifica un parámetro

e) Imprime 5

PREGUNTA 2

En un programa C en Linux, el equivalente de esto en el shell es:

```
2>&3
```

a) dup(2,3);

b) No hay forma de hacer algo parecido en C

c) dup(2);

d) dup2(3,2);

e) dup(3);

PREGUNTA 3

En general, cuando hablamos de planificación, un proceso puede transitar entre estos dos estados

a) De listo para ejecutar a bloqueado

b) De bloqueado a ejecutando

c) De Listo para ejecutar a ejecutando

d) Un proceso no puede tener ningún estado

e) De perezoso a virtual.

PREGUNTA 4

Los siguientes dos comandos ejecutados en un terminal (el $ es el prompt del Shell)

```
$sh > /dev/null
ls -l /proc/$$/fd | awk '{print $11}'
```

a) Me dicen donde tiene sus entradas y salidas la shell que acabo de ejecutar

b) Mostrará todos los ficheros abiertos por el proceso menos /dev/null

c) Van a comportarse de forma errática

d) Es posible que den un fallo por no tener permisos

e) No se escribirá nada por el terminal

PREGUNTA 5

Suponiendo que múltiples hilos pueden llamar a esa función para incrementar la misma variable entera, y que mutex es una variable global correctamente inicializada

```
void incrementar(int *x)
{
        int aux=*x;
        aux=aux+1;
        pthred_mutex_lock(&mutex);
        *x=aux; pthread_mutex_unlock(&mutex);
```

```
}
```

a) Donde pone pthread_mutex_lock debería poner pthread_mutex_unlock y viceversa.

b) Se debería quitar los asteriscos de ese código para que al menos compile

c) Una llamada a la función incrementar si garantiza que el entero apuntado por x se incremente

d) Una llamada a la función incrementar nunca hace que el entero apuntado por x se incremente

e) Una llamada a la función incrementar no garantiza que el entero apuntado por x se incremente.

PREGUNTA 6

Cuando se llama a malloc

a) Lo normal es que no intervenga el kernel salvo cuando se usa mucha memoria

b) El puntero que devuelve apunta a una dirección de la pila (del registro de activación del malloc)

c) Siempre se reserva la cantidad de memoria pedida, nunca una cantidad mayor

d) Siempre interviene el kernel para hacer mas grande el segmento

e) Se entra al kernel, que llamara a kmalloc para servir la llamada al sistema.

PREGUNTA 7

Si se ejecuta este código

```
char *p=0;

*p=53;
```

a) Probablemente el proceso morirá porque recibirá una señal SIGINT, pero podría manejarla/ignorarla

b) Probablemente el proceso morirá porque recibirá una señal SIGSEGV, pero podría manejarla/ignorarla

c) Probablemente el proceso morirá porque recibirá una señal SIGKILL, pero podría manejarla/ignorarla

d) Está perfecto, se asigna un byte con valor 53 en la dirección apuntada por p

e) El proceso morirá con total seguridad porque recibirá una señal SIGKILL

PREGUNTA 8

El comando (-r significa reverse)

```
seq 1 100 | sort -r
```

a) No va a ordenar bien los números al revés, es incorrecto

b) Ordena los números del revés correctamente

c) Ejecuta 3 procesos

d) Está mal porque no llama a dup

e) Está mal porque sort una un algoritmo de ordenación inestable

PREGUNTA 9

En general los spin locks

a) No se debe usar nunca porque los mutex son mejores

b) No se deben usar cuando la contienda es alta

c) No se deben usar para proteger una región crítica en la que se escriba una estructura

d) No se deben usar cuando la contienda es baja

e) No se deben usar si la región crítica es pequeña

PREGUNTA 10

Si ejecutamos este programa:

```
char b[32*1024*1024];
int ç
main(int argc, char *argv[]){
        int fd[2];
        if (pipe(fd)<0)
                err(1,"pipe failed");
        write(fd[1],b,sizeof(b));
```

```
        read(fd[0],b,sizeof(b));
        exit(EXIT_SUCCESS);
    }
```

a) Si read y write no fallan, seguramente se quede bloqueado de por vida

b) Falla siempre en la creación del pipe porque su argumento está mal

c) Si read y write no fallan, el programa recibe una señal de SIGSEGV

d) Si read y write no fallan, seguramente termina con éxito y no escribe nada por su salida

e) Si read y write no fallan, el programa recibe una señal SIGPIPE

PREGUNTA 11

```
    int
    main(int argc, char *argv[]){
        int z;
        int *p;
        z=1;
        p=(int *)z;
        printf("%d",*p);
    }
```

a) Es correcto, imprime 1

b) Atravesará el puntero a NULL (la dirección está en la primera página) por eso eso dará un segmentation fault.

c) Es incorrecto por los tamaños de los punteros por eso da un segmentation fault

d) Imprime una dirección de la memoria de la pila

e) No compila no se puede convertir un entero en un puntero

PREGUNTA 12

```
int x;
x=open("/tmp/afile",O_RDONLY);
if (x<0)
        err(1,"error");
dup2(x,0);
close(x);
execl("/bin/wc","wc","-l",NULL);
exit(0);
```

a) Se queda leyendo del terminal y, cuando termina, escribe en /tmp/afile el numero de líneas leídas

b) Si no se puede abrir el fichero /tmp/afile, hay un segmentation fault.

c) Se queda leyendo del terminal y, cuando termina, escribe en el terminal el número de líneas leídas

d) Si se puede abrir el fichero /tmp/afile, escribe por la salida estándar el número de líneas de ese fichero

e) Siempre da un fallo en ejecución en la llamada execl, ya que debería estar usando execv

PREGUNTA 13

Cuando se escribe con la llamada al sistema de write en un pipe y ningún proceso tiene abierto el otro extremo.

a) La llamada al sistema write devuelve un 0

b) El proceso recibe una señal SIGPIPE, que por omisión provoca que termine el proceso

c) Se escriben los datos en el pipe normalmente y se quedan allí de por vida

d) El proceso se queda bloqueado hasta que alguien lee del pipe

e) La llamada al sistema write falla y devuelve -1.

PREGUNTA 14

La sección de un fichero binario ELF que describe las instrucciones del programa es:

a) La sección. got

b) La sección .data

c) La sección .text

d) La sección .got.plt

e) La sección .bss

PREGUNTA 15

Vamos a procesar varios ficheros como los del ejemplo ($ es el prompt del sistema). Los ficheros contienen saludos y nombres en minúsculas y los saludos son como mucho de dos palabras y en todas las líneas hay un numero al final. El comando sed, un ejemplo de cuya de ejecución correcta se ve a continuación.

```
$ cat fich
hola pepe 123
adios juan 456
hasta luego alberto 345
$ sed -E 's/^[a-z]+([a-z]+)? ([a-z]+)[0-9]+$/hecho \2/' fich
hecho pepe
hecho juan
hecho alberto
```

a) No funciona igual en algunos casos sin el circunflejo en la expresión regular

b) No funciona igual en algunos casos sin el dólar en la expresión regular

c) Habría que escribirlo con grep, que es mejor para editar flujos de texto

d) Funciona igual para todos los ficheros, aunque se quite el principio y el final de línea (circunflejo y dólar en la expresión regular)

e) Es incorrecto, está mal el escapado del comando de sed.

PREGUNTA 16

En un sistema donde se aplica RELRO (relocation read-only), y se resuelven todos los símbolos que provienen de las bibliotecas

a) Cuando se enlace el programa con gcc

b) Cuando el proceso realiza una llamada al sistema exit

c) Cuando se compila el programa con gcc

d) Cuando el programa comienza a ejecutar

e) En tiempo de ejecución, a medida que se van llamando a las funciones.

PREGUNTA 17

```
int
main (int argc, char *argv[])
{
        int i;
        argc=2;
        argv++;
        for(i=0;i<argc;i++) {
        printf("%s\n",argv[i]);
}
return 0;
}
```

Si ejecutamos (con $ el prompt)

$ programa uno dos tres

a) Hace exactamente lo mismo que el comando echo

b) Imprime tres líneas: programa, uno dos

c) main nunca puede rertornar

d) Imprime dos lineas: uno, dos

e) No puede decrementar argc

PREGUNTA 18

Suponiendo que el resto del programa es correcto, este código:

```
x=fork();
if (x==0)
        fork();
printf("x");
```

a) Imprime una equis

b) Imprime tres equis

c) No imprime nada

d) Imprime cuatro equis

e) Imprime dos equis.

PREGUNTA 19

```
int main()
{
        int x;
        x=fork();
        if (x==0)
                execl("/bin/ls","ls","/",NULL);
        execl("/bin/wc","wc","-l",NULL);
}
```

a) Escribe las líneas del directorio raíz y se queda leyendo de la entrada estándar

b) Da un segmentation fault por el NULL que se pasa como último argumento a execl

c) No escribe nada por su salida y termina

d) Provoca un interbloque (deadlock)

e) Escribe un número de entradas del directorio raíz

PREGUNTA 20

El siguiente programa (suponiendo que tiene los includes adecuados, child() lee de la entrada estándar y gen_lines() es correcto):

```
void child(void);
void gen_lines(int);
int main(int argc, char *argv[])
{
	int fd[2];
	int status;
	if (pipe(fd)<0)
		err(EXIT_FAILURE,"cannot make a pipe");
	switch(fork()){
		case -1:
			err(EXIT_FAILURE,"cannot fork");
		case 0:
			close(fd[1]);
			dup2(fd[0],0);
			close(fd[0]);
			child();
			err(EXIT_FAILURE,"exec failed");
	}
	wait(&status);
	gen_lines(fd[1]);
	close(fd[0]);
	exit(WEXITSTATUS(status));
}
void child(void)
{
	//aqui lee de stdin (no llama a exec)
}
void gen_lines(int fd)
```

```
{
        //esto es correcto
}
```

a) Se quedará bloqueado indefinidamente

b) Tiene un error, cierra fd[1] dos veces

c) Funciona correctamente

d) No funciona bien porque llama a fork pero no a exec

e) El proceso padre se quedará zombie ya que el hijo llamará a exec

TEST.FINAL - ENERO 2022-2023

PREGUNTA 1

El siguiente código forma parte de un programa con múltiples threads concurrentes que comparten una variable global entera llamada x. El código del resto del programa es correcto. La siguiente función:

```
pthread_spin_lock_t lk:
void incrementar()
{
        if(x < Maxcont){
                pthread_spin_lock (&lk);
                x++;
                pthread _spin_unlock(&lk);
        }
}
```

a) Está bien, pero estaría mejor implementado con un Test-and-Set

b) La variable x podrá terminar siendo mayor que Maxcont

c) Está mal, pero si usara un mutex en lugar de un spin lock estaría bien

d) Esto no compila, el spin lock se tiene que inicializar en la función incrementar

e) La variable x siempre será menor o igual que Maxcont

PREGUNTA 2

El siguiente código forma parte de un programa con múltiples threads concurrentes que comparten una lista enlazada de estructuras

Cliente llamada listaclientes.

Siendo lk una variable compartida de tipo pthread_spin_lock_t correctamente inicializada, la siguiente función:

```
int eliminarcliente(Cliente *c)
{
    if(existe_cliente(c, listaclientes)){
        pthread_spin_lock(&lk);
        borrar_de lista(c, listaclientes);
        pthread_spin_unlock(&lk);
        return 0;
    }
    return -1;
}
```

a) Está mal, pero si usara un mutex estaría bien

b) Está bien

c) Está mal porque seguramente va a provocar un deadlock

d) Está mal, hay una condición de carrera a la hora de acceder a la lista enlazada

e) Esto mal, la primera sentencia de la función debería ser una llamada a pthread_spin_init

PREGUNTA 3

Dado el siguiente código (suponiendo que el resto del programa es correcto):

```
p = &a;
a = *(p++);
```

a) La variable a no puede ser un puntero.

b) Deja en a el mismo valor que tenía.

c) No puede ser correcto, hay un error de tipos.

d) La variable a no puede ser un entero.

e) Cambia siempre el valor de la variable.

PREGUNTA 4

El siguiente programa:

```
#include <stdio.h>
#include <stdlib.h>
enum {
        NItems = 32768,
};
int
main(int argc, char *argv[])
{
        char *array;
        array = (char *)malloc(NItems);
        array[16] = 'r';
        printf("%ld\n", sizeof(array));
        free (array);
        exit (EXIT_SUCCESS);
}
```

a) Imprime el tamaño del array, que se sabrá en tiempo de ejecución.

b) Imprime 32768.

c) Imprime 1 (el tamaño de un char).

d) Da un error en ejecución: está accediendo a memoria no reservada (justo después del malloc).

e) Imprime un número pequeño (un número menor que 128). Debería imprimir 8 bytes (64 bits) o 4 bytes (32bits) ,depende de la arquitecutra

PREGUNTA 5

El siguiente programa principal:

```
int
main(int arge, char *argv[])
{
        fork();
        fork();
        printf("hola\n");
        exit (EXIT_SUCCESS);
}
```

a) Escribe unas veces hola por su salida

b) Falla en ejecución por no pasar un parámetro a fork().

c) Escribe cuatro veces hola por su salida.

d) Escribe tres veces hola por su salida

e) Escribe dos veces hola por su salida

PREGUNTA 6

Un proceso muere por un segmentation fault porque ha intentado:

a) Escribir en un pipe del que no puede leer ningún otro proceso

b) Acceder a una dirección de memoria virtual cuya página estaba en área de intercambio (swap)

c) Hacer un malloc y ya no quedaba memoria

d) Acceder a una dirección de memoria virtual que no puede tener traducción a memoria física

e) Leer una variable local de una función que ya ha retornado

PREGUNTA 7

Suponiendo que no hay ningún tipo de error relacionado con permisos, si ejecutamos esto en una shell (el dólar es el prompt, la opción -i de Is muestra el número de i-nodo, y el comando In sin argumentos crea un enlace duro):

```
$ rm /tmp/a

$ echo hola > /tmp/a

$ ln /tmp/a /tmp/b

$ ls -i /tmp/a

7746226 /tmp/a
```

Si ahora ejecutamos un programa que tiene la siguiente sentencia realizando una llamada al sistema:

```
unlink ("/tmp/a");
```

a) El fichero con i-nodo 7746226 ya no existe, se ha borrado, y no hay ningún enlace simbólico roto

b) No se puede hacer eso, no hay ninguna llamada al sistema llamada unlink

c) El fichero con i-nodo 7746226 sigue existiendo y tiene su cuenta de referencias (número de enlaces, st_nlinks) a 1

d) El fichero con i-nodo 7746226 ya no existe, se ha borrado, y ahora tendremos un enlace simbólico roto (/tmp/b)

e) El fichero con i-nodo 7746226 sigue existiendo y tiene su cuenta de referencias (número de enlaces, st_nlinks) a 2

PREGUNTA 8

```
#!/bin/sh
if test S# -eq 0
        then
        echo debes pasar al menos un argumento 1>&2
        exit 1
fi
while test $# -ne 0
```

```
do
        echo $1
        shift
        done
exit 0
```

Si en una shell ejecutamos esto (el dólar es el prompt):

```
$ ls -1 x.sh
-rwxrwxr-x 1 esoriano esoriano 176 Jan 8 14:20 x.sh
$./x.sh uno dos tres
```

a) Escribirá un error por su salida de errores y acabará con fallo

b) No se podrá ejecutar porque no tiene permisos de ejecución

c) Escribirá sus tres argumentos, cada uno en una línea, y terminará con éxito

d) Escribirá una única línea con el primer argumento y terminará con éxito

e) Se mete en un bucle infinito, escribiendo siempre el primer argumento

PREGUNTA 9

Dos procesos que se comunican mediante un pipe creado con la llamada al sistema pipe(2):

a) Deben hacer wait tras cada write o read del pipe.

b) Deben tener una relación de parentesco (uno tiene que heredar el pipe del padre)

c) No tienen que tener una relación de parentesco (para eso están los fifos).

d) Pueden ser hijos del mismo padre, pero no padre e hijo.

e) Deben tener ambos procesos ambos extremos del pipe abiertos y sólo cerrarlos cuando acaben de usar el pipe.

PREGUNTA 10

Tenemos un sistema con paginación en demanda. A lo largo de la vida de un proceso, únicamente las páginas que han tenido traducción en la tabla de páginas:

a) Se corresponden con memoria que devuelve malloc (se haya accedido a ellas o no).

b) No tienen nunca permiso para escribir en ellas (es lo que significa copy on write).

c) Tienen siempre permiso para escribir en ellas, para eso se les asigna un marco.

d) Han tenido memoria física real reservada en algún momento: un marco por página.

e) Han sido escritas o leídas: no se heredan nunca del proceso padre.

PREGUNTA 11

La función flock()...

a) No existe tal función

b) Sirve para usar un lock de lectores/escritores para evitar condiciones de carrera al acceder a una variable

c) Sirve para usar un lock de lectores/escritores para evitar condiciones de carrera al acceder a un fichero

d) Sirve para hacer escrituras append en un fichero (escribir al final para añadir al fichero)

e) Sirve para descargar el buffer cuando tenemos entrada/salida con buffering.

PREGUNTA 12

El objetivo de la siguiente función es escribir en la salida estándar el contenido del fichero que se le pasa como argumento:

```
void
dump (char *path)
{
        char buf[8*1024];
        int nr;
        int fd;
        fd = open(path, O_RDONLY);
        do{
                nr = read(fd, buf, 8*1024);
```

```
            write(1, buf, nr);
        }while(nr != 0);
        close(fd);
    }
```

a) Está mal porque no se puede declarar un array de chars de esa forma

b) Está mal porque no controla los errores correctamente y puede provocar un bucle infinito

c) Está bien

d) Está mal porque debería abrir el fichero con la opción O_RDWR | O_TRUNC

e) Está mal porque intenta escribir en la entrada estándar

PREGUNTA 13

Cuando se mata un proceso con el comando kill:

a) Se le comunica al proceso cerrando un pipe.

b) No se puede matar un proceso con ese comando.

c) Nunca se mata un proceso con señales: es peligroso.

d) No hay un comando kill, es una llamada al sistema.

e) Se le manda una señal y el kernel acaba con el proceso.

PREGUNTA 14

En un sistema de ficheros estilo Unix con i-nodos...

a) Se usa asignación de espacio contigua

b) Se usa asignación de espacio indexada de 4 niveles

c) Se usa asignación de espacio con lista enlazada con tabla

d) Se usa asignación de espacio con lista enlazada

e) Se usa asignación de espacio indexada con esquema combinado

PREGUNTA 15

La función free:

a) Puede recibir puntero que no ha devuelto malloc (por ejemplo que apunten a la pila)

b) Es una llamada al sistema: libera un segmento de memoria del proceso

c) Es una función de espacio de usuario

d) Libera entradas en la tabla de páginas de un proceso

e) Se puede hacer dos veces seguidas sobre el mismo puntero: no es un error

PREGUNTA 16

Tenemos un sistema operativo con paginación en demanda y overcommiting ejecutando en un sistema con páginas de 4KB. Al ejecutar el siguiente programa en C:

```
#include
int
man(int argc, char argv[])
{
        void *p = malloc(1000 * 4 * 1024);
        if (p == NULL){
                exit(EXIT_FAILURE) ;
        }
        exit (EXIT_SUCCESS);
}
```

a) El proceso no llegará a reclamar ninguno de los 1000 marcos de página, nunca acabará con fallo

b) El proceso siempre necesitará reservar al menos 1000 marcos de página durante su ejecución, puede acabar con fallo

c) Esto no compila porque en C no podemos declarar un puntero a void

d) El proceso necesitará conseguir más de 1000 bloques de datos del sistema de ficheros para ejecutar, puede acabar con fallo

e) El proceso necesitará conseguir más de 1000 marcos de página antes de comenzar a ejecutar, puede acabar con fallo

PREGUNTA 17

Si queremos leer línea a línea un fichero en un programa escrito en C:

a) Lo mejor es usar la llamada al sistema system para ejecutar cat

b) Lo mejor es usar la función de entrada/salida sin buffering llamada getline

c) Lo mejor es usar la función de entrada/salida con buffering llamada fgets

d) Lo mejor es usar la llamada al sistema read, que es entrada/salida con buffering

e) Lo mejor es usar la llamada al sistema execv para ejecutar /bin/cat

PREGUNTA 18

Tenemos una función en C llamada TAS que es un test-and-set, y su implementación es correcta. Tres procesos concurrentes comparten la variable v y la variable x, que están inicializadas a cero y no se usa en ninguna otra parte del código. La variable i es local y no se comparte. Los tres procesos ejecutan este código concurrentemente:

```
for （i=0; 1<10; i++)
{
        If (!TAS(v) ){
                x++;
        }
}
```

a) El valor final de x será 10

b) El valor final de x será 30

c) El valor final de x será 0

d) No sabemos cuál será el valor final de x, hay una condición de carrera.

e) El valor final de x será 1

TEST1 - JUNIO 2018-2019

PREGUNTA 1

Dado el siguiente fragmento de código (suponemos que el resto es correcto)

```
void init3(char *buf)
{
        int i;
        for(i=0;i<3;i++){
                buf[i]=0;
        }
}
int main(int argc, char *argv[])
{
        char buf[12];
        init3(&buf[0]);
....
}
```

a) El programa dará un fallo ejecución por acceder a direcciones no reservadas.

b) El programa no inicializa los tres primeros valores del array, inicializa sólo parcialmente el primero (3 bytes).

c) La función inicializa los tres primeros valores del array.

d) El programa no compilará por un error de tipos, la función recibe un puntero y se le pasa un array.

e) Sería correcto si la llamada fuse init3(buf)

PREGUNTA 2

Dado el siguiente código:

```
struct MsgPos{
        int x;
```

```
        int y;
        char msg[32];
};
typedef struct MsgPos MsgPos;
void setmsg(MsgPos *m, char *msg)
{
        strncpy(m.msg,msg,32);
}
int main(int argc, char *argv[])
 {
        MsgPos m;
        setmsg(&m,"hola");
        printf("%s",m.msg);
}
```

a) No compilará porque no se puede poner un array de tamaño fijo en un record en C.

b) Imprimirá hola

c) No compilará porque en la declaración de la variable debería poner struct MsgPos;

d) No compilará por un error de tipos (punteros vs struct)

e) Compilará, pero dará un error de ejecución al atravesar un puntero a NULL.

PREGUNTA 3

Cuando se escribe con la llamada al sistema write en un pipe y ningún proceso tiene abierto el otro extremo.

a) El proceso recibe una señal SIGPIPE, que por omisión provoca que termine el proceso.

b) Se queda bloqueado hasta que alguien lea del pipe

c) La llamada al sistema write devuelve un 0

d) La llamada al sistema write falla y devuelve -1

e) Se escriben los datos en el pipe normalmente y se quedan allí de por vida.

PREGUNTA 4

Si el procesador de nuestro ordenador usa una tabla de páginas multinivel de N niveles, en el peor de los casos

a) Se necesitan N accesos a memoria reales para acceder a una dirección

b) Solo se necesita un acceso a memoria real para acceder a una dirección, porque tenemos la TLB

c) Se necesitan N+1 accesos a memoria para acceder a una dirección

d) No existen máquinas reales con tablas multinivel, solo máquinas teóricas

e) Nunca necesitamos acceder a la memoria, ya que tenemos las cache L1, L2 y L3

PREGUNTA 5

El siguiente script:

```
#!/bin/sh
for i in $(ls |egrep '.z$'); do
        echo $i
done
```

a) Hace exactamente lo mismo que ejecutar echo *.z

b) Puede escribir nombres de ficheros y directorios en el directorio local que no acaban en z (dos caracteres, el carácter punto seguido del carácter z)

c) Hace exactamente lo mismo que ejecutar ls -d *.z

d) Escribe nombres de ficheros y directorios en el directorio local que acaban en .z (dos caracteres, el carácter punto seguido del carácter z)

e) Hace exactamente lo mismo que ejecutar ls *z

PREGUNTA 6

En enlazador dinámico se encarga de:

a) No existe el concepto de cargador dinámico, son siempre estáticos

b) Proporcionar las funciones de gestión de memoria dinámica: malloc y free

c) Ejecutar instrucciones especiales del procesador para memoria dinámica

d) Compilar partes del programa según se necesiten usar o no

e) Resolver los símbolos que usa el programa según se vayan necesitando

PREGUNTA 7

Suponiendo que múltiples hilos pueden llamar a esta función para incrementar la misma variable entera, y que mutex es una variable global correctamente inicializada

```
void incrementar(int *x)
{
        int aux=*x;
        aux=aux+1;
        pthread_mutex_lock(&mutex);
        *x=aux; pthread_mutex_unlock(&mutex);
}
```

a) Una llamada a la función incrementar sí garantiza que el entero apuntado por x se incremente

b) Una llamada a la función incrementar no garantiza que el entero apuntado por x se incremente

c) Una llamada a la función incrementar nunca hace que el entero apuntado por x se incremente

d) Donde pone pthread_mutex_lock debería poner pthread_mutex_unlock y viceversa

e) Se deberían quitar los asteriscos de ese código para que al menos compile

PREGUNTA 8

Suponiendo que el fichero /tmp/a existe y se puede abrir sin problemas

```
fd=open("/tmp/a",O_RDWR|O_CREAT|O_TRUNC,0640);
dup2(fd,2);
```

close(fd);

a) El proceso se quedará sin salida estándar porque se cierra fd al final

b) El fichero /tmp/a pasará a ser la salida de errores del proceso.

c) Dará un fallo en la llamada dup2 porque el fichero no está abierto en el modo correcto.

d) El proceso se quedará sin salida de errores porque se cierra fd al final.

e) El fichero /tmp/a pasará a ser la salida estándar el proceso.

PREGUNTA 9

La paginación en demanda consiste

a) Cargar de golpe todos los marcos de página del proceso antes de comenzar a ejecutarlo

b) Ir cargando en marcos de página las partes del proceso poco a poco, según se necesiten.

c) Ir liberando los marcos de página usados en demanda según el proceso los deja de usar.

d) No cargar ningúna página de memoria del proceso hasta que no acaba de ejecutar

e) Ir reservando marcos de página para la pila del proceso(stack) según se va haciendo mas grande.

PREGUNTA 10

Un segmentation fault (SIGSEGV) se produce cuando:

a) Se produce un error de entrada/salida cuando se está leyendo/escribiendo un fichero

b) Se produce un fallo de página y la dirección virtual si pertenece al espacio de direcciones del proceso

c) Se produce un fallo de página y la dirección virtual no pertenece al espacio de direcciones del proceso

d) Se bloquea el programa en un lock durante mucho tiempo

e) Se produce un fallo de página y no quedan marcos de página disponibles para este proceso.

PREGUNTA 11

El siguiente código suponiendo que fd tiene un fichero abierto correctamente para lectura:

```
char c;
int nr;
int count=0;
while ((nr=read(fd,&c,1))!=0){
        if (nr<0)
                err(EXIT_FAILURE,"can't read");
        count++;
}
printf("count:%d\n",count);
```

a) Da un error en ejecución porque no se puede pasar un char* como segundo parámetro de read

b) Cuenta correctamente los bytes de un fichero, pero es muy ineficiente. c) Está mal, es un bucle infinito

d) Cuenta correctamente los bytes del fichero y es muy eficiente

e) Cuenta incorrectamente los bytes del fichero, se salta bytes

PREGUNTA 12

En general, cuando hablamos de planificación, un proceso puede transitar entre estos dos estados

a) De bloqueado a ejecutando.

b) Un proceso no puede tener ningún estado

c) De listo para ejecutar a bloqueado

d) De perezoso a virtual

e) De listo para ejecutar a ejecutando

PREGUNTA 13

```
enum{
        BUFSZ=10 };
};
int main(int argc, char *argv[])
{
        char *buf;
        buf=(char *)malloc(BUFSZ*sizeof(char*));
        ...
}
```

a) Compilará, pero reserva menos de 10 bytes de memoria.

b) Reserva correctamente justo 10 bytes de memoria.

c) No va a necesitar llamar a free para liberar buf(está en la pila)

d) No reserva 10 bytes de memoria.

e) Va a dar un error de tipos, no compilará.

PREGUNTA 14

Si queremos leer línea a línea un fichero en un programa escrito en C

a) Lo mejor es usar la llamada al sistema read, que es entrada/salida con buffering

b) Lo mejor es usar la función de entrada/salida sin buffering llamada getline

c) Lo mejor es usar la llamada al sistema system para ejecutar cat

d) Lo mejor es usar la llamada al sistema execv para ejecutar /bin/cat

e) Lo mejor es usar la función de entrada/salida con buffering llamada fgets

PREGUNTA 15

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

```
        for(i=1;i<argc; i++) {
        printf("%c", argv[i] [0]);
        }
}
```

a) Imprimirá el primer caracter de cada argumento

b) No compila, argv debería ser un string para que compilase

c) Escribirá el nombre del programa

d) Dará un error de tipos, no compilará

e) Atraviesa memoria sin reservar, dará un error de ejecución.

PREGUNTA 16

Suponiendo que el i-nodo 33 está libre antes de ejecutar nada, ejecutamos los siguientes comandos

```
$> echo -n hola > afile
$> ls -i afile          #mostramos el número de inodo de afile
    33 afile
$> ln afile link        #creamos un enlace duro
$> rm afile
```

a) El i-nodo 33 está libre porque se ha borrado el fichero.

b) El i-nodo 33 sigue existiendo, su cuenta de referencias está a 2 y sus datos son "hola"

c) El i-nodo 33 sigue existiendo, su cuenta de referencias está a 1 y el fichero está vacio.

d) El i-nodo 33 sigue existiendo, su cuenta de referencias está a 2 y el fichero está vacio.

e) El i-nodo 33 sigue existiendo, su cuenta de referencias está a 1 y sus datos son "hola" mksir

PREGUNTA 17

```
$>ls -l
total 0
-rw-rw-r--  1 paurea paurea  0  jun 6  11:54  a
-rw-rw-r--  1 paurea paurea  0  jun 6  11:54  aa
-rw-rw-r--  1 paurea paurea  0  jun 6  11:54  b
-rw-rw-r--  1 paurea paurea  0  jun 6  11:54  bb
-rw-rw-r--  1 paurea paurea  0  jun 6  11:54  c
-rw-rw-r--  1 paurea paurea  0  jun 6  11:54   cc
$>ls -l | grep -v '^total' | awk '{print $8}' | sed -E s/://g | sort -u | wc -l
```

a) Escribe 6

b) Da un error, hay que escapar los parámetros de sed para la Shell

c) Escribe 42

d) Escribe 1

e) No escriba nada

PREGUNTA 18

El siguiente fragmento de código (suponiendo que el resto del código es correcto)

```
Calc *c;
Calc arithm;
c=(Calc *)malloc(sizeof(Calc));
c=&arithm;
res=plot(c);
```

a) No es correcto porque necesita reservar memoria para arithm

b) No es correcto, tiene un leak de memoria.

c) No compilará por un error de tipos.

d) Compilará, pero dará un error de ejecución.

e) Es correcto, no tiene ningún problema.

PREGUNTA 19

```
int main(int argc, char *argv[])
{
        int fd[2];
        char c='x';
        if (pipe(fd)<0)
                err(1,"pipe failed");
        write(fd[1],&c,1);
        read(fd[0],&c,1);
        printf("%c",c);
        exit(EXIT_SUCCESS);
}
```

a) Si read y write no fallan, se queda bloqueado.

b) Si read y write no fallan, escribe una x

c) Falla siempre en la creación del pipe porque su argumento está mal.

d) Si read y write no fallan, el programa recibe una señal SIGPIPE

e) Si read y wirte no fallan, el programa recibe una señal SIGSEGV

PREGUNTA 20

Dado el comando:

```
echo Soy una prueba | sed -E 's/.*oy(.*)$/Eres\1/g'
```

a) Escribe Eres Soy

b) Escribe SEres una prueba

c) Escribe Eresoy

d) No escribe nada

e) Escribe Eres una prueba

TEST1 - JUNIO 2019-2020

PREGUNTA 1

La memoria virtual

Seleccione una:

a) Es inútil hoy en día, ya no se usa

b) Solo es interesante porque permite usar memoria secundaria como swap

c) Complica la depuración de los programas, pero facilita su compresión.

d) Permite que el planificador implemente una política Round Robin.

e) Permite proteger la memoria de los distintos procesos.

PREGUNTA 2

El siguiente programa

```
struct Coord{
        int x;
        int y;
};
typedef struct Coord Coord;
int main(int argc, char *argv[])
{
        int i;
        Coord *coords;
        coords=malloc(10*2*sizeof(int));
         for(i=0;i<10;i++){
                coords[i].x=i;
                coords[i].y=i;
                printf("(%d,%d)\n",coords[i].x,coords[i].y);
        }
}
```

a) Es incorrecto: puede desbordar el array.

b) Es correcto, imprime pares de números consecutivos.

c) No compila, la declaración debería ser struct Coord *.

d) No compila, lo que devuelve malloc no se puede asignar a coords.

e) Es correcto, imprime pares de números iguales.

PREGUNTA 3

El siguiente código forma parte de un programa con múltiples threads concurrentes que comparten una variable global entera llamada x.

```
pthread_spin_lock lk;
    void incrementar(){
        if (x<Maxcount) {
        pthread_spin_lock(&lk);
            x++;
            pthread_spin_unlock(&lk);
        }
    }
```

a) Está mal, la variable lk debería ser local y no lo es

b) Está mal, pero si usara un mutex estaría bien.

c) Está mal, la primera sentención de la función debería ser una llamada a pthread_spin_init

d) Está mal, tiene una condición de carrera

e) Está bien.

PREGUNTA 4

Si un proceso lee de un pipe vacío y ningún proceso tiene abierto el otro extremo de escritura.

a) Se queda bloqueado hasta que otro proceso abra el pipe para leer

b) Se queda bloqueado para toda la vida

c) Recibe una señal SIGPIPE. Esto ocurre cuando se escribe en un pipe y no hay nadie leyendo en el otro extremo

d) Lee 0 bytes

e) La Lectura retorna error (-1)

PREGUNTA 5

Si el directorio de trabajado de un proceso que ejecuta este código es /tmp, suponiendo que no hay ningún problema de permisos y la llamada a open no falla.

```
fd=open("./afile",O_WRONLY|O_CREAT|O_TRUNC,0040);
dup2(fd,1);
```

a) Creará o truncará el fichero /tmp/afile y será su salida estándar

b) Creará o truncará el fichero /tmp/afile y será su entrada estándar

c) Creará o truncará el fichero /afile , y será su salida estándar

d) Fallará el dup2 porque el modo de apertura no es compatible con la entrada estándar.

e) Creará o truncará el fichero /afile y será su entrada estándar.

PREGUNTA 6

```
void
x(void){
        fork();
        fork();
        printf("%d\n",getppid());
        exit(EXIT_SUCCESS);
}
```

a) Escribe cuatro números por su salida en el mismo orden, siempre dos números serán iguales y otros dos no.

b) Escribe cuatro números por su salida, siempre todos los números serán distintos.

c) Escribe cuatro números por su salida, siempre dos números serán iguales, y otros dos no.

d) Escribe tres números por su salida, siempre todos los números serán distintos.

e) Escribe cuatro números por su salida en el mismo orden, siempre todos los números son distintos.

PREGUNTA 7

```
int
main(int argc, char *argv[]){
        int i;
        for(i=0;i<5;i++){
                if (!i)
                        break;
        }
        printf("%d\n",i);
        exit(EXIT_SUCCESS);
}
```

a) Imprime 0

b) Se queda en bucle infinito

c) No compila

d) Imprime 11 valores

e) No imprime nada

PREGUNTA 8

Si el sistema está usando enlazado dinámico con lazy binding y tengo un programa con un fnt() definida.

a) La dirección de memoria de la función fnt se resuelve cuando gcc compila y enlaza el programa

b) La dirección de memoria de la función fnt no se resuelve hasta la primera vez que se le llama

c) No se pueden usar funciones en ese caso.

d) La dirección de memoria de la función fnt se resuelve cuando el programa termina la ejecución.

e) La dirección de memoria de la función fnt se resuelve junto antes de que comience la ejecución del programa.

PREGUNTA 9

Los siguientes comandos ejecutados en un terminal (ten en cuenta que dos dólares segundos se sustituyen por el PID del Shell actual y que el programa de Shell se escribe en su salida de errores.

```
$> sh 2> /dev/null


ls -l /proc/$$/fd
```

a) Me dicen donde tienen sus entradas y salidas el terminal.

b) Nunca mostrarán /dev/null

c) Es posible que den un fallo por no tener permisos

d) Me dicen donde tiene sus entradas y salidas que acabo de ejecutar

e) Van a comportarse de forma errática

PREGUNTA 10

```
#!/bin/sh
for i in $*
do
        echo $1
        shift
done
```

a) Da un error de sintaxis porque el for está mal escrito

b) Imprime tantas líneas como argumentos se le pasan al script pero siempre imprime el mismo argumento

c) Imprime una línea con cada uno de los argumentos que se le pasan al script

d) Imprime líneas todo el rato, es un bucle infinito

e) Da un error, no hay ningún comando llamado shift

PREGUNTA 11

Si diferentes threads de la biblioteca de pthreads de Linux llamas a esta función concurrentemente y suponemos que el resto del código es correcto y que todo está inicializado debidamente.

```
int counter;
pthread_spinlock_t lk;
int
validecounter(){
        pthread_spin_lock(&lk);
        if (counter>0){
                if (counter<Maxvalue){
                        pthread_spin_unlock(&lk);
                        return 1;
                }
                else{
                        pthread_spin_unlock(&lk);
                        return 0;
                }
        }
        pthread_spin_unlock(&lk);
        return 0;
}
```

a) Está mal porque tiene una condición de Carrera

b) No tiene problemas de concurrencia, pero se podría escribir mucho mejor.

c) Está mal, la variable lk debería ser una variable local

d) Está perfecto

e) Está mal porque provocará un interbloqueo

PREGUNTA 12

El comando:

```
gcc -o x x.o
```

a) Compila el fichero fuente del programa y crea un fichero objeto

b) Compilar el fichero objeto y crea un fichero ejecutable

c) Enlaza el fichero fuente de la librería de C y crea un fichero objeto

d) Enlaza el fichero objeto con la librería de C y crea un fichero ejecutable

e) Enlaza el código fuente del programa y crea un fichero objeto

PREGUNTA 13

La siguiente llamada a atoi, suponiendo que el resto del código está bien:

```
int n;
char *nstr=”patata”;
n=atoi(nstr);
printf(“%d”,n);
```

a) Dará un error de ejecución.

b) No compilará por un error de tipos

c) No imprimirá nada

d) Imprimirá 0

e) imprimirá un valor arbitrario.

PREGUNTA 14

```
int
main(int argc, char *argv[]){
        int i;
        argc--;
        for(i=0;i<argc; i++) {
```

```
            printf("%s\n", argv[i]);
        }
        return 0;
    }
    $> programa uno dos tres
```

a) Hace exactamente lo mismo que el comando echo

b) Imprime tres líneas, programa uno, dos

c) No puede decrementar argc

d) main nunca puede retornar

e) Imprime dos líneas, dos tres

PREGUNTA 15

```
    int fd;
    char buf[Bufsize];
    int nr;
    while(1){
        nr=read(fd,buf,Bufsize);
        if (nr==0)
            err(EXIT_FAILURE,"error en lectura");
        if (write(1,buf,nr)!=nr)
            err(EXIT_FAILURE,"error en la escritura");
    }
```

a) Está mal porque es un bucle infinito

b) Está perfecto, escribe el fichero completo en la salida y termina con éxito

c) Está mal porque no se comprueba correctamente el error del read

d) Está mal porque no se puede hacer un write en el descriptor 1

e) Está mal porque no se comprueba correctamente el error del write