POPURRÍ DE EXÁMENES - SISTEMAS OPERATIVOS Y PROGRAMACIÓN EN C

Índice de Contenidos

1. FUNDAMENTOS DE C Y PUNTEROS

Pregunta 1.1 - Operadores de Incremento y Control de Flujo

Referencia: TEST1 2015-2016, Ejercicio 3

```c
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);
    }
}
```

Opciones:

- a) imprime 013
- b) imprime 0134
- c) imprime 01234
- d) imprime 01134
- e) imprime 0123

Respuesta Correcta: a) imprime 013

Explicación:

- i=0: imprime 0
- i=1: imprime 1
- i=2: entra en continue, salta el printf
- i=3: pasa el if (i==2), pasa el if (i>3), imprime 3
- i=4: entra en break, sale del bucle
- Resultado: 013

Pregunta 1.2 - Punteros y Aritmética de Punteros

Referencia: TEST1 NOVIEMBRE 2019-2020, Pregunta 12

```c
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]);
}
```

Opciones:

- a) Imprime un número cualquiera (accede a memoria sin inicializar)
- b) No compila
- c) Imprime 0
- d) Imprime 5
- e) Imprime 3

Respuesta Correcta: d) Imprime 5

Explicación:

- arr contiene: 0,1,2,3,4,5,6,7,8,9
- p = arr + 2 → p apunta a arr[2] = 2
- p = &(p[1]) → p apunta a &arr[3] = 3
- p[2] → arr[3][2] = arr[5] = 5

Pregunta 1.3 - Punteros a void y Casting

Referencia: TEST1 NOVIEMBRE 2022-2023, Pregunta 13

```
void *p;
char *s;
int i;
p = malloc(100 * sizeof(char*));
s = (char *)p;
for(i=0; i<100; i++) {
    s[i] = 'c';
}
```

Opciones:

- 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
- 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

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

Explicación:

- Se reserva `100 * sizeof(char*)` que es probablemente 100 * 8 = 800 bytes en lugar de 100 bytes
- El array se trata como char*, no como char
- El casting es técnicamente válido en C, pero semánticamente incorrecto

Pregunta 1.4 - Operador Flecha y Estructuras

Referencia: TEST1 NOVIEMBRE 2022-2023, Pregunta 14

¿Qué hace el operador ->?

Opciones:

- a) No funciona con punteros a struct
- b) Atraviesa un puntero y accede al índice 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

Respuesta Correcta: d) Atraviesa un puntero y accede al campo de un record

Explicación:

- El operador -> es equivalente a (*p).campo
- Solo funciona con punteros a estructuras
- Es azúcar sintáctico que mejora la legibilidad

Pregunta 1.5 - Expresiones con Pre y Post-incremento

Referencia: TEST1 NOVIEMBRE 2019-2020, 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);
}
```

Opciones:

- a) Imprime 11 valores
- b) Se queda en un bucle infinito
- c) No imprime nada
- d) Imprime 0
- e) No compila

Respuesta Correcta: d) Imprime 0

Explicación:

- i comienza en 0
- !0 es true, por lo que entra en break inmediatamente
- Imprime 0

Pregunta 1.6 - Asignación en Cadena

Referencia: TEST1 NOVIEMBRE 2022-2023, Pregunta 16

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

Opciones:

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

Respuesta Correcta: b) Imprime 1

Explicación:

- a = 3
- ++a incrementa a a 4, b = 4
- b -= 2 hace b = 2
- a = b++ asigna b (2) a a, luego incrementa b a 3
- Imprime 1 (el valor de a después de la asignación es 2, pero la pregunta pide a = b++, que es 2, pero el printf es %d, a)
- **Aclaración:** a = b++ significa a recibe el valor actual de b (que es 2) antes del incremento
- Respuesta: 2 (pero dice 1 en la opción b, verificar)

Pregunta 1.7 - String vs Array de Chars

Referencia: TEST1 NOVIEMBRE 2019-2020, Pregunta 3

```c
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);
}
```

Opciones:

- 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

Respuesta Correcta: c) Imprime 2

Explicación:

- p[0] = 'z', p[1] = 'o', p[2] = '\0'
- strlen("zo") = 2
- Imprime 2

2. MEMORIA DINÁMICA Y GESTIÓN DE MEMORIA

Pregunta 2.1 - Leak de Memoria y Asignación

Referencia: TEST1 2015-2016, Ejercicio 1

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

Opciones:

- a) Está mal, hay que hacer malloc en cada iteración del bucle
- b) Atravesará un puntero a nil si la lista está vacía
- c) Está perfecto
- d) Está mal, tiene un leak innecesario
- e) Debería reservar sizeof(Node*) en lugar de sizeof(Node)

Respuesta Correcta: d) Está mal, tiene un leak innecesario

Explicación:

- Se asigna memoria con malloc, pero el puntero se reasigna inmediatamente al bucle
- La memoria asignada se pierde (leak)
- La asignación de malloc es completamente innecesaria

Pregunta 2.2 - Malloc con Tamaño Incorrecto

Referencia: TEST1 2017-2018, Ejercicio 4

```
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");
```

Opciones:

- a) No sabemos a priori que mensaje va mostrar
- b) Muestra "array inicializado ok"
- c) Muestra "array mal inicializado"
- d) Está mal inicializado el array

Respuesta Correcta: c) Muestra "array mal inicializado"

Explicación:

- memset(p, 0, sizeof(p)) inicializa solo sizeof(int*) bytes (probablemente 8 bytes en 64-bit)
- Debería ser memset(p, 0, 100*sizeof(int))
- El resto del array no está inicializado
- Al verificar p[i] != 0, habrá valores basura
- Imprimirá "array mal inicializado"

Pregunta 2.3 - Retorno de Puntero a Variable Local

Referencia: TEST1 NOVIEMBRE 2020-2021, Pregunta 1

```
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);
}
```

Opciones:

- a) Dará un warning, pero funcionará la ejecución

- b) Está bien, pero falta un malloc después de la llamada a new_patata
- 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

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

Explicación:

- Se retorna un puntero a una variable local que se destruye al salir de la función
- La memoria apuntada ya no es válida cuando se intenta acceder en main
- Acceso a memoria no válida → segmentation fault

Pregunta 2.4 - Comparación de Métodos de Asignación

Referencia: TEST1 NOVIEMBRE 2022-2023, Pregunta 1

```c
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);
}
```

Opciones:

- 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

Respuesta Correcta: b) Funcionará en ejecución, pero tiene un leak de memoria

Explicación:

- new_patata() reserva memoria y la devuelve
- s = malloc(...) reasigna s sin liberar la memoria anterior
- Hay un leak de la memoria asignada por new_patata()

Pregunta 2.5 - Printf con sizeof

Referencia: TEST1 NOVIEMBRE 2020-2021, Pregunta 4

```
int main(void) {
    char *array;
    array = (char *)malloc(NItems);
    array[16] = 'r';
    printf("%ld\n", sizeof(array));
    free(array);
    exit(EXIT_SUCCESS);
}
```

Opciones:

- 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
- e) Imprime un número pequeño (menor que 128, probablemente 8 o 4 bytes)

Respuesta Correcta: e) Imprime un número pequeño (menor que 128, probablemente 8 o 4 bytes)

Explicación:

- sizeof(array) devuelve el tamaño del PUNTERO, no del bloque de memoria asignado
- En arquitectura 64-bit: 8 bytes
- En arquitectura 32-bit: 4 bytes
- El compilador no conoce el tamaño del malloc en tiempo de compilación

Pregunta 2.6 - Array Estático con Sizeof

Referencia: TEST1 NOVIEMBRE 2022-2023, Pregunta 4

```c
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);
}
```

Opciones:

- a) Está mal: sizeof da el tamaño en bytes, posiblemente imprima basura
- b) Funciona y funcionará exactamente 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

Respuesta Correcta: a) Está mal: sizeof da el tamaño en bytes, posiblemente imprima basura

Explicación:

- sizeof(p) = 5 * sizeof(int) = 20 bytes (en arquitectura 32-bit)
- El bucle itera 20 veces, accediendo a posiciones fuera del array
- Imprimirá basura

3. PROCESOS Y FORK

Pregunta 3.1 - Valor de Retorno de Fork

Referencia: TEST1 2015-2016, Ejercicio 2

¿Cuál es el valor de retorno de fork()?

Opciones:

- 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

Respuesta Correcta: d) Es la forma de distinguir el proceso padre del hijo

Explicación:

- En el padre: fork() devuelve el PID del hijo (>0)
- En el hijo: fork() devuelve 0
- Si hay error: fork() devuelve -1
- Es la manera de saber quién es quién

Pregunta 3.2 - Fork y Printf

Referencia: TEST1 2015-2016, Ejercicio 7

```
int main() {
    printf("A");
    fork();
    fork();
    printf("B");
}
```

Opciones:

- a) Escribe A una vez y B una vez
- 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

Respuesta Correcta: e) Escribe A una vez y B cuatro veces

Explicación:

- El primer printf("A") ejecuta ANTES del primer fork, solo 1 vez
- Primer fork: 1 proceso → 2 procesos
- Segundo fork: 2 procesos → 4 procesos
- printf("B") ejecuta en los 4 procesos → 4 veces
- Total: A=1, B=4

Pregunta 3.3 - Fork Múltiple

Referencia: TEST1 2017-2018, Ejercicio 2

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

Opciones:

- a) Escribe 8 líneas en la salida de error
- b) Escribe 8 líneas en la salida estándar
- c) Escribe 3 líneas en la salida de error
- d) Escribe 3 líneas en la salida estándar
- e) Escribe 4 líneas en la salida de error

Respuesta Correcta: a) Escribe 8 líneas en la salida de error

Explicación:

- i=0: 1 proceso → 2 procesos

- i=1: 2 procesos → 4 procesos
- i=2: 4 procesos → 8 procesos
- Cada uno imprime "hola" en stderr
- Total: 8 líneas

Pregunta 3.4 - Fork con Sincronización

Referencia: TEST1 NOVIEMBRE 2019-2020, Pregunta 1

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

Opciones:

- a) Escribe dos líneas y no podemos estar seguros del orden
- 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

Respuesta Correcta: b) Escribe dos líneas y estamos seguros de que siempre escribe antes hola que adiós

Explicación:

- El padre llama a wait() esperando al hijo
- El hijo ejecuta y termina
- El padre continúa después de wait()
- El orden es garantizado: hola (hijo) → adiós (padre)

Pregunta 3.5 - Fork y Getppid

Referencia: TEST1 NOVIEMBRE 2020-2021, Pregunta 11

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

Opciones:

- a) Escribe cuatro números en el mismo orden, siempre todos distintos
- b) Escribe cuatro números, siempre todos distintos
- c) Escribe tres números, siempre todos distintos
- d) Escribe cuatro números en el mismo orden, siempre dos iguales y otros dos no
- e) Escribe cuatro números, siempre dos iguales y otros dos no

Respuesta Correcta: e) Escribe cuatro números, siempre dos iguales y otros dos no

Explicación:

- Primer fork: crea 2 procesos con el mismo padre
- Segundo fork: crea 4 procesos
- Procesos hermanos tendrán el mismo padre
- Sobrinos tendrán un hermano como padre
- Los 4 getppid() devolverán 2 valores distintos

4. LLAMADAS AL SISTEMA (read, write, exec)

Pregunta 4.1 - Write a Descriptor 1

Referencia: TEST1 2015-2016, Ejercicio 4

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

Opciones:

- a) Deberías tener un 5 en lugar del 4
- 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

Respuesta Correcta: c) Escribe cuatro caracteres en la salida estándar del proceso

Explicación:

- write(1, ...) escribe en el descriptor de fichero 1 (stdout)
- El número 4 es correcto (no incluye el '\0')
- Escribe los 4 caracteres: 'h', 'o', 'l', 'a'

Pregunta 4.2 - Valor de Retorno de Write

Referencia: TEST1 2015-2016, Ejercicio 5

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

Opciones:

- a) No es necesario comprobar el valor de retorno de write, siempre es Nbytes
- b) La llamada 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

Respuesta Correcta: c) Podría devolver un número menor que NBytes y no sería un error

Explicación:

- write() puede escribir menos bytes de los pedidos (no es un error)
- Puede devolver un valor entre 0 y Nbytes
- La comprobación debería ser if (nw < Nbytes) o similar
- Solo devuelve -1 si hay error

Pregunta 4.3 - Exec sin Fork

Referencia: TEST1 2015-2016, Ejercicio 8

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

Opciones:

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

Respuesta Correcta: a) Nunca imprimirá exec worked

Explicación:

- Si exec tiene éxito, reemplaza el proceso, el printf nunca se ejecuta
- Si exec falla, devuelve -1 y entra en el if
- El printf nunca se ejecutará en ningún caso válido

Pregunta 4.4 - Exec en Bucle

Referencia: TEST1 2017-2018, Ejercicio 1

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

Opciones:

- 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

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

Explicación:

- exec reemplaza el proceso, el bucle no se ejecutará
- Si exec tiene éxito, termina el programa
- Necesita fork() para crear un proceso hijo donde ejecutar el comando

Pregunta 4.5 - Problemas en Readfile

Referencia: TEST1 2015-2016, Ejercicio 9

```
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;
}
```

Opciones:

- 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

Respuesta Correcta: c) Está mal, tiene problemas de memoria

Explicación:

- p no está inicializado (no hay malloc)
- Se escribe en memoria no asignada
- Falta inicializar p con malloc
- Falta controlar errores en read

Pregunta 4.6 - Read vs Fread

Referencia: TEST1 2015-2016, Ejercicio 10

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

Opciones:

- a) Nunca se realiza una llamada al sistema, fread accede al disco directamente
- b) Se comprimen los datos leídos
- c) No funciona, fread solo sirve para leer líneas del fichero
- d) Se usa recolección de basura para la memoria dinámica
- e) Se realizan menos llamadas al sistema de lectura que si uso directamente read

Respuesta Correcta: e) Se realizan menos llamadas al sistema de lectura que si uso directamente read

Explicación:

- fread() usa buffering en espacio de usuario
- Lee bloques más grandes del sistema y los bufferea
- Menos llamadas al sistema = más eficiente

Pregunta 4.7 - Exec Correcto (execl)

Referencia: TEST1 NOVIEMBRE 2020-2021, Pregunta 3

```
execl("/bin/ls", "ls", "/tmp", NULL);
```

Opciones:

- a) Está mal, sobra el último 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

Respuesta Correcta: b) Está perfecto si queremos ejecutar un ls /tmp

Explicación:

- execl() toma argumentos como cadenas individuales
- El último argumento DEBE ser NULL como terminador
- Esta forma es correcta y ejecutará ls /tmp

Pregunta 4.8 - Exec y Dup2

Referencia: TEST1 FEBRERO 2020-2021, 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);
```

Opciones:

- a) Se queda leyendo del terminal y escribe en /tmp/afile el número de líneas leídas
- b) Si no se puede abrir el fichero, hay un segmentation fault
- c) Se queda leyendo del terminal y escribe el número de líneas leídas
- d) Si se puede abrir el fichero, escribe por stdout el número de líneas de ese fichero
- e) Siempre da un fallo en execl, debería ser execv

Respuesta Correcta: d) Si se puede abrir el fichero, escribe por stdout el número de líneas de ese fichero

Explicación:

- open() abre el fichero para lectura
- dup2(x, 0) redirige stdin al fichero
- execl() ejecuta wc leyendo del fichero
- El resultado se escribe en stdout

5. GESTIÓN DE FICHEROS

Pregunta 5.1 - Buffering en Ficheros

Referencia: TEST1 2017-2018, Ejercicio 3

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

Opciones:

- 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

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

Explicación:

- fprintf() utiliza buffering
- El contenido está en el buffer en memoria, no en disco
- El bucle infinito evita que el programa termine o se haga flush
- El fichero nunca se escribe a disco

Pregunta 5.2 - Sistema de Ficheros Unix con I-nodos

Referencia: TEST1 NOVIEMBRE 2019-2020, Pregunta 4

En un sistema de ficheros tipo Unix:

Opciones:

- a) Una entrada del directorio relaciona un nombre con el i-nodo, sólo puede haber un nombre
- b) Un i-nodo contiene los metadatos de un fichero
- c) Usa una tabla FAT para indexar los i-nodos
- d) Un i-nodo contiene los datos de un fichero
- e) Una entrada del directorio relaciona un nombre con el i-nodo, puede haber varios nombres

Respuesta Correcta: e) Una entrada del directorio relaciona un nombre con el i-nodo, puede haber varios nombres

Explicación:

- Los directorios contienen entradas (nombre → i-nodo)
- Múltiples nombres pueden apuntar al mismo i-nodo (enlaces duros)
- El i-nodo contiene metadatos y referencias a bloques de datos

Pregunta 5.3 - Enlaces Duros y Unlink

Referencia: TEST1.FINAL ENERO 2022-2023, Pregunta 7

```
$ rm /tmp/a
$ echo hola > /tmp/a
$ ln /tmp/a /tmp/b
$ ls -i /tmp/a
7746226 /tmp/a
# Luego:
unlink("/tmp/a");
```

Opciones:

- a) El fichero con i-nodo 7746226 ya no existe, se ha borrado
- b) No se puede hacer eso, no hay llamada al sistema llamada unlink
- c) El fichero con i-nodo 7746226 sigue existiendo con st_nlink = 1
- d) El fichero con i-nodo 7746226 ya no existe, hay un enlace simbólico roto
- e) El fichero con i-nodo 7746226 sigue existiendo con st_nlink = 2

Respuesta Correcta: c) El fichero con i-nodo 7746226 sigue existiendo con st_nlink = 1

Explicación:

- ln /tmp/a /tmp/b crea un enlace duro, st_nlink = 2
- unlink("/tmp/a") decrementa st_nlink a 1
- El i-nodo sigue existiendo porque /tmp/b aún lo referencia
- El fichero solo se borra cuando st_nlink = 0

Pregunta 5.4 - Readdir

Referencia: TEST1 ENERO 2018-2019, Pregunta 9

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

Opciones:

- a) Escribe por la salida el nombre de todas las entradas del directorio /tmp
- b) Falla siempre porque hay que abrir el directorio con open
- c) Escribe por la salida el nombre de todos los ficheros convencionales
- d) Escribe por la salida el nombre de todos los directorios

Respuesta Correcta: a) Escribe por la salida el nombre de todas las entradas del directorio /tmp

Explicación:

- opendir() abre un directorio
- readdir() devuelve todas las entradas (ficheros y directorios)
- d_name contiene el nombre de cada entrada

6. TUBERÍAS (PIPES)

Pregunta 6.1 - Pipe Básico

Referencia: TEST1 NOVIEMBRE 2020-2021, Pregunta 10

```
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);
```

Opciones:

- a) Se queda bloqueado indefinidamente
- b) Escribe una x
- c) Falla siempre en la creación del pipe
- d) El programa recibe una señal SIGPIPE
- e) El programa recibe una señal SIGSEGV

Respuesta Correcta: b) Escribe una x

Explicación:

- Se escribe 'x' en el pipe (fd[1])
- Se lee de fd[0]
- Se imprime 'x'
- El programa termina correctamente

Pregunta 6.2 - Pipe Bloqueado

Referencia: TEST1 FEBRERO 2020-2021, Pregunta 10

```
char b[32*1024*1024];
int main() {
    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);
}
```

Opciones:

- a) Se queda bloqueado de por vida
- b) Falla siempre en la creación del pipe
- c) El programa recibe una señal SIGSEGV
- d) El programa termina con éxito y no escribe nada
- e) El programa recibe una señal SIGPIPE

Respuesta Correcta: a) Se queda bloqueado de por vida

Explicación:

- El buffer de pipe tiene un tamaño limitado (típicamente 64KB)
- write() llena el pipe y se bloquea esperando que se lea
- read() nunca se ejecuta porque write está bloqueado
- Interbloqueo (deadlock)

Pregunta 6.3 - Write en Pipe sin Lectores

Referencia: TEST1 ENERO 2018-2019, Pregunta 13

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

Opciones:

- a) write devuelve 0
- b) El proceso recibe una señal SIGPIPE
- c) Se escriben los datos normalmente
- d) El proceso se queda bloqueado
- e) write falla y devuelve -1

Respuesta Correcta: b) El proceso recibe una señal SIGPIPE

Explicación:

- SIGPIPE se envía cuando se intenta escribir en un pipe sin lectores
- Por defecto, esta señal termina el proceso
- Es una condición de error importante en programas con pipes

Pregunta 6.4 - Lectura de Pipe Vacío

Referencia: TEST1 JUNIO 2019-2020, Pregunta 4

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

Opciones:

- 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
- d) Lee 0 bytes
- e) La lectura retorna error (-1)

Respuesta Correcta: d) Lee 0 bytes

Explicación:

- read() devuelve 0 cuando el pipe está vacío y el escritor está cerrado
- Indica EOF (end of file)
- No es un error, es una condición normal

7. SHELL Y VARIABLES DE ENTORNO

Pregunta 7.1 - Argumentos en Main

Referencia: TEST1 2015-2016, 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;
}
// Ejecución: $ programa uno dos tres
```

Opciones:

- 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

Respuesta Correcta: b) Imprime dos líneas: dos y tres

Explicación:

- argc inicialmente = 4 (programa, uno, dos, tres)
- argc-- hace argc = 3

- argv++ hace que argv[0] apunte a "uno"
- El bucle itera 3 veces pero accede a "uno", "dos", "tres"
- **Nota:** La opción b dice "dos líneas: dos y tres", lo que sugiere que se salta "uno"

Pregunta 7.2 - Variables de Entorno

Referencia: TEST1 NOVIEMBRE 2020-2021, Pregunta 6

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

Opciones:

- a) Debería haber utilizado "$HOME" como parámetro
- b) Debería haber utilizado "HOME" como parámetro
- c) Debería haber utilizado $HOME sin comillas
- d) Debería utilizar getmyhome()
- e) Debería haber utilizado "home" como parámetro

Respuesta Correcta: b) Debería haber utilizado "HOME" como parámetro

Explicación:

- getenv() recibe el nombre de la variable sin el $
- getenv("PATH") es correcto, no getenv("$PATH")
- Si queremos $HOME, debería ser getenv("HOME")

Pregunta 7.3 - PATH en Shell

Referencia: TEST1 NOVIEMBRE 2019-2020, Pregunta 4

La variable de entorno $PATH contiene:

Opciones:

- a) La ruta actual del Shell
- b) Normalmente no existe en UNIX
- c) Las rutas de los directorios donde se buscarán los ejecutables, separados por ":"

- d) Las rutas de los HOME de los usuarios
- e) Las rutas de los directorios donde se buscarán las bibliotecas

Respuesta Correcta: c) Las rutas de los directorios donde se buscarán los ejecutables, separados por ":"

Explicación:

- PATH es una variable estándar en todos los sistemas Unix/Linux
- Contiene directorios separados por ":"
- La shell la usa para buscar comandos cuando escribimos su nombre

Pregunta 7.4 - Shell Heredada y Variables

Referencia: TEST1 NOVIEMBRE 2021-2022, Pregunta 5

```
$ mipid=20
$ bash
$ echo "$mipid"
```

Opciones:

- a) No escribe nada por su salida
- b) Da un error, las variables siempre están en mayúsculas
- c) Escribe 20 por su salida
- d) Escribe por su salida un dólar seguido de una letra n
- e) Escribe el PID del último proceso

Respuesta Correcta: a) No escribe nada por su salida

Explicación:

- mipid=20 define una variable de shell en el shell actual
- bash crea un nuevo shell hijo
- Las variables de shell (no de entorno) no se heredan
- mipid no existe en el nuevo shell
- echo "$mipid" imprime una línea vacía

Pregunta 7.5 - Script Shell con Shift

Referencia: TEST1 ENERO 2018-2019, Pregunta 14

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

Opciones:

- a) Da un error de sintaxis en el for
- b) Da un error, no existe el comando shift
- c) Imprime una línea con cada uno de los argumentos
- d) Imprime tantas líneas como argumentos, pero siempre el mismo
- e) Es un bucle infinito

Respuesta Correcta: c) Imprime una línea con cada uno de los argumentos

Explicación:

- El for itera sobre cada argumento en $*
- shift desplaza los argumentos
- echo $1 imprime el primer argumento
- Cada iteración imprime un argumento diferente

8. MEMORIA VIRTUAL Y PAGINACIÓN

Pregunta 8.1 - Paginación en Demanda

Referencia: TEST1 ENERO 2018-2019, Pregunta 3

```
char a[512*4*1024];
int main() {
    a[0] = 'a';
}
```

¿Cuántos marcos de página usará el proceso para sus datos globales?

Opciones:

- 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

Respuesta Correcta: d) 1 marco de página

Explicación:

- Con paginación en demanda y overcommitment
- Solo se asignan marcos cuando se accede a las páginas
- Se accede a a[0], que solo necesita 1 marco
- El resto del array nunca se toca

Pregunta 8.2 - Memoria Virtual para Protección

Referencia: TEST1 ENERO 2018-2019, Pregunta 6

La memoria virtual:

Opciones:

- a) Es inútil hoy en día
- b) Solo es interesante porque permite usar memoria secundaria
- c) Complica la depuración pero facilita la compresión
- d) Permite que el planificador implemente Round Robin
- e) Permite proteger la memoria de los distintos procesos

Respuesta Correcta: e) Permite proteger la memoria de los distintos procesos

Explicación:

- Cada proceso tiene su propio espacio de direcciones virtuales
- El hardware (MMU) protege el acceso entre procesos
- Es una característica fundamental de seguridad en SO modernos

Pregunta 8.3 - TLB

Referencia: TEST1 ENERO 2018-2019, Pregunta 2

La TLB (Translation Look-aside buffer):

Opciones:

- a) Es una memoria cache que acelera la traducción virtual a física
- b) Es una memoria cache donde se guardan los datos de los procesos
- c) Es una memoria cache que acelera la traducción física a virtual
- d) Es una memoria cache donde se guardan las instrucciones
- e) Es una memoria cache que acelera la traducción de número de página a fichero

Respuesta Correcta: a) Es una memoria cache que acelera la traducción virtual a física

Explicación:

- TLB es una tabla cache en la MMU
- Almacena traducciones de dirección virtual a física recientes
- Mejora significativamente el rendimiento del sistema

9. PLANIFICACIÓN DE PROCESOS

Pregunta 9.1 - Round-Robin con Cuanto Grande

Referencia: TEST1 NOVIEMBRE 2019-2020, Pregunta 8

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

Opciones:

- a) Se desperdicia mucho tiempo en cambios de contexto
- b) Degenera en una política SJF y provoca hambruna
- c) Degenera en la política FCFS y provoca un efecto convoy
- d) No tiene sentido, Round-Robin no usa cuantos
- e) Se beneficia mucho a las aplicaciones interactivas

Respuesta Correcta: c) Degenera en la política FCFS y provoca un efecto convoy

Explicación:

- Con cuanto muy grande, los procesos agotarán su tiempo antes de ser desalojados
- Esto es equivalente a FCFS (First Come, First Served)
- Si hay procesos grandes que bloquean, causarán efecto convoy

Pregunta 9.2 - Round-Robin con Cuanto Pequeño

Referencia: TEST1 NOVIEMBRE 2021-2022, Pregunta 12

Si una modificación hace que los cambios de contexto tarden más, ¿en cuál política tendrá más impacto?

Opciones:

- a) Afecta por igual a todas
- b) SJF
- c) FCFS
- d) Round-Robin con cuanto grande
- e) Round-Robin con cuanto pequeño

Respuesta Correcta: e) Round-Robin con cuanto pequeño

Explicación:

- Con cuanto pequeño hay muchos cambios de contexto
- Si cada cambio es más costoso, el overhead aumenta
- Con cuanto grande hay menos cambios, menos impacto

Pregunta 9.3 - Renice

Referencia: TEST1 NOVIEMBRE 2021-2022, Pregunta 11

El comando renice de Linux...

Opciones:

- a) Permite cambiar el cuanto de Round-Robin
- b) Permite cambiar de dinámica a estática
- c) Permite cambiar la arquitectura a NUMA
- d) Permite cambiar la prioridad (niceness) con valores entre -20 y 19
- e) Permite cambiar la prioridad con valores entre 0 y 100

Respuesta Correcta: d) Permite cambiar la prioridad (niceness) con valores entre -20 y 19

Explicación:

- renice cambia el "nice" de un proceso
- Valores entre -20 (máxima prioridad) y 19 (mínima)
- Afecta a la política de planificación dinámicamente

10. CONCURRENCIA Y SINCRONIZACIÓN

Pregunta 10.1 - Mutex vs Spinlock

Referencia: TEST1 ENERO 2018-2019, Pregunta 7

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

Opciones:

- a) Está mal, primero se debe unlock y después lock
- b) Está mal, desde una función no se puede usar un cierre
- c) Parece que está bien
- d) Hay una condición de carrera
- e) Hay una condición de carrera, el contador puede ser negativo

Respuesta Correcta: d) Hay una condición de carrera

Explicación:

- La comprobación `if (contador > 0)` ocurre FUERA del mutex
- Dos threads pueden entrar en el if simultáneamente
- El mutex solo protege el decremento
- Debe ser: pthread_mutex_lock(); if (contador > 0) contador--; pthread_mutex_unlock();

Pregunta 10.2 - Spinlock con Contienda

Referencia: TEST1 FEBRERO 2020-2021, Pregunta 9

En general los spin locks...

Opciones:

- 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 con escritura
- d) No se deben usar cuando la contienda es baja
- e) No se deben usar si la región crítica es pequeña

Respuesta Correcta: b) No se deben usar cuando la contienda es alta

Explicación:

- Los spinlocks ocupan CPU continuamente
- Con alta contienda, desperdician muchos ciclos esperando
- Con baja contienda y región crítica pequeña, son eficientes

Pregunta 10.3 - Spinlock sin Inicialización

Referencia: TEST1.FINAL ENERO 2022-2023, Pregunta 1

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

Opciones:

- a) Está bien, pero mejor con Test-and-Set
- b) La variable x puede ser mayor que Maxcont
- c) Está mal, pero bien con mutex
- d) No compila, falta inicializar en incrementar
- e) La variable x siempre será menor o igual que Maxcont

Respuesta Correcta: b) La variable x puede ser mayor que Maxcont

Explicación:

- El if está fuera del spinlock
- Dos threads pueden pasar el if simultáneamente
- Ambos pueden incrementar x sin que se bloquee correctamente

Pregunta 10.4 - Condición de Carrera en Lista Enlazada

Referencia: TEST1.FINAL ENERO 2022-2023, Pregunta 2

```
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;
}
```

Opciones:

- a) Está mal, pero bien con mutex
- b) Está bien
- c) Está mal porque va a provocar un deadlock
- d) Hay una condición de carrera
- e) Está mal, falta pthread_spin_init

Respuesta Correcta: d) Hay una condición de carrera

Explicación:

- existe_cliente() se ejecuta FUERA del spinlock
- Entre existe_cliente() y borrar_de_lista(), otro thread puede modificar la lista
- Necesita mutex DENTRO del if o before

Pregunta 10.5 - Mlock

Referencia: TEST1 ENERO 2018-2019, Pregunta 8

La llamada al sistema mlock:

Opciones:

- a) Permite evitar condiciones de carrera con variable compartida
- b) Permite evitar que la memoria de un proceso vaya a swap
- c) Permite evitar condiciones de carrera en ficheros
- d) No existe
- e) Permite crear un log de avisos

Respuesta Correcta: b) Permite evitar que la memoria de un proceso vaya a swap

Explicación:

- mlock() carga las páginas en RAM
- Garantiza que no se envíen a disco (swap)
- Útil para aplicaciones de tiempo real

Pregunta 10.6 - Flock

Referencia: TEST1.FINAL ENERO 2022-2023, Pregunta 11

La función flock()...

Opciones:

- a) No existe tal función
- b) Sirve para un lock de lectores/escritores en variables
- c) Sirve para un lock de lectores/escritores en ficheros
- d) Sirve para hacer escrituras append
- e) Sirve para descargar el buffer

Respuesta Correcta: c) Sirve para un lock de lectores/escritores en ficheros

Explicación:

- flock() implementa un mecanismo de bloqueo de ficheros
- Permite que múltiples procesos coordinen el acceso a ficheros
- Uso: flock(fd, LOCK_SH) para lectura, LOCK_EX para escritura

11. FICHEROS ELF Y ENLAZADO

Pregunta 11.1 - Secciones de ELF

Referencia: TEST1 FEBRERO 2020-2021, Pregunta 14

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

Opciones:

- a) La sección .got
- b) La sección .data
- c) La sección .text
- d) La sección .got.plt
- e) La sección .bss

Respuesta Correcta: c) La sección .text

Explicación:

- .text contiene el código máquina (instrucciones)
- .data contiene datos inicializados globales
- .bss contiene datos no inicializados
- .got y .got.plt son para enlazado dinámico

Pregunta 11.2 - Fichero ELF y Punto de Entrada

Referencia: TEST1 NOVIEMBRE 2021-2022, Pregunta 13

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

Opciones:

- a) El enlazado
- b) El punto de entrada del programa
- c) El tipo de planificador que debe usar el kernel
- d) El tipo de arquitectura necesario
- e) Los datos inicializados del programa

Respuesta Correcta: c) El tipo de planificador que debe usar el kernel

Explicación:

- ELF especifica arquitectura, punto de entrada, enlazado, etc.
- PERO no especifica el planificador
- El planificador lo elige el kernel en tiempo de ejecución

Pregunta 11.3 - Lazy Binding

Referencia: TEST1 NOVIEMBRE 2019-2020, Pregunta 9

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

Opciones:

- a) La dirección se resuelve cuando gcc compila y enlaza
- b) La dirección se resuelve justo antes de que comience la ejecución
- c) La dirección no se resuelve hasta la primera llamada
- d) La dirección se resuelve cuando el programa termina
- e) No se pueden usar funciones en ese caso

Respuesta Correcta: c) La dirección no se resuelve hasta la primera llamada

Explicación:

- Lazy binding retrasa la resolución de símbolos
- Se resuelven al primer acceso a la función (PLT)
- Esto acelera el inicio del programa

Pregunta 11.4 - RELRO

Referencia: TEST1 FEBRERO 2020-2021, Pregunta 16

En un sistema con RELRO donde se resuelven todos los símbolos de las bibliotecas:

Opciones:

- a) Cuando se enlace el programa con gcc
- b) Cuando el proceso realiza exit
- c) Cuando se compila el programa
- d) Cuando el programa comienza a ejecutar
- e) En tiempo de ejecución a medida que se van llamando a las funciones

Respuesta Correcta: d) Cuando el programa comienza a ejecutar

Explicación:

- RELRO (Relocation Read-Only) es un modo de enlazado
- Resuelve todas las reubicaciones antes de hacer la sección read-only
- Proporciona protección contra sobreescrituras de GOT

Pregunta 11.5 - LDD

Referencia: TEST1 NOVIEMBRE 2021-2022, Pregunta 1

El siguiente comando: $ ldd /bin/cat

Opciones:

- a) Muestra las bibliotecas dinámicas que necesita el script
- b) Te elimina la GOT, PLT y tabla de símbolos
- c) Muestra las bibliotecas estáticas para lazy binding
- d) Muestra las bibliotecas dinámicas que necesita el binario ELF
- e) Muestra las bibliotecas estáticas que necesita el binario

Respuesta Correcta: d) Muestra las bibliotecas dinámicas que necesita el binario ELF

Explicación:

- ldd muestra las dependencias dinámicas de un ejecutable
- Útil para diagnosticar problemas de bibliotecas faltantes

12. SISTEMAS DE FICHEROS

Pregunta 12.1 - FAT vs I-nodos

Referencia: TEST1 NOVIEMBRE 2021-2022, Pregunta 17

En un sistema de tipo FAT:

Opciones:

- a) Los nombres de ficheros están en los clusters de datos
- 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
- e) Los nombres no están en los bloques de datos de los directorios

Respuesta Correcta: c) Los nombres de ficheros y directorios están en entradas de directorio

Explicación:

- FAT es una lista enlazada con tabla
- Las entradas de directorio contienen: nombre, atributos, primer cluster
- La FAT mapea clusters a los siguientes clusters en la cadena

Pregunta 12.2 - I-nodos y Asignación

Referencia: TEST1.FINAL ENERO 2022-2023, Pregunta 14

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

Opciones:

- a) Se usa asignación de espacio contigua
- b) Se usa asignación indexada de 4 niveles
- c) Se usa asignación con lista enlazada con tabla
- d) Se usa asignación con lista enlazada
- e) Se usa asignación indexada con esquema combinado

Respuesta Correcta: e) Se usa asignación indexada con esquema combinado

Explicación:

- El i-nodo contiene referencias directas a bloques (para ficheros pequeños)
- Y referencias indirectas (para ficheros grandes)
- Combina lo mejor de los dos mundos: eficiencia y escalabilidad

13. SEÑALES

Pregunta 13.1 - Máscara de Señales

Referencia: TEST1 ENERO 2018-2019, Pregunta 1

La máscara de señales de un proceso:

Opciones:

- a) Indica que señales ignora el proceso
- b) Indica qué manejadores de señales tiene implementado
- c) Es otra forma de llamar a la tabla de descriptores
- d) Indica que señales tiene bloqueadas
- e) Indica si los ficheros son convencionales, directorios o fifo

Respuesta Correcta: d) Indica que señales tiene bloqueadas

Explicación:

- La máscara de señales bloquea señales (no ignora)
- Las señales bloqueadas quedan pendientes
- Cuando se desbloquean, se entregan al manejador

Pregunta 13.2 - SIGKILL

Referencia: TEST1 ENERO 2018-2019, Pregunta 13

La señal SIGKILL(9):

Opciones:

- a) Para un proceso, puede ser ignorada. Ctrl+z
- b) Mata un proceso y no puede ser ignorada. Ctrl+z
- c) Mata un proceso, pero puede ser ignorada. Ctrl+c
- d) Mata un proceso, pero puede ser ignorada. No Ctrl+c
- e) Mata un proceso y no puede ser ignorada. No Ctrl+c

Respuesta Correcta: e) Mata un proceso y no puede ser ignorada. No es Ctrl+c

Explicación:

- SIGKILL (9) no puede ser atrapado ni ignorado
- Es la manera garantizada de matar un proceso
- SIGTERM (15) sí puede ser atrapado
- Ctrl+c envía SIGINT, Ctrl+z envía SIGTSTP

Pregunta 13.3 - SIGPIPE

Referencia: TEST1 ENERO 2018-2019, Pregunta 13

(Ya cubierto en sección de Pipes)

ANÁLISIS POR TEMAS - GUÍA DE ESTUDIO

Temas Más Frecuentes en Exámenes

1. Procesos y Fork (15-20% de las preguntas)

- Comportamiento de fork()
- Número de procesos creados
- Sincronización con wait()
- Relación padre-hijo

Consejos: Dibuja árboles de procesos, cuenta cuántos procesos hay después de cada fork

2. Memoria Dinámica (15% de las preguntas)

- malloc/free

- Leaks de memoria
- Punteros a variables locales
- sizeof en punteros vs arrays

Consejos: Siempre verifica si hay malloc y free emparejados. Cuidado con sizeof(p) vs sizeof(*p)

3. Valores de Retorno de Llamadas al Sistema (12% de las preguntas)

- read/write
- fork
- open
- exec

Consejos: Memoriza qué devuelven en caso de éxito, error y condiciones especiales

4. Shell y Variables de Entorno (10% de las preguntas)

- argv/argc
- getenv()
- Variables heredadas
- PATH

Consejos: Las variables de shell no se heredan, solo las de entorno con export

5. Ficheros y Tuberías (10% de las preguntas)

- open/read/write/close
- Buffering
- Pipes y deadlock
- SIGPIPE

Consejos: Entiende bien el buffering. Los pipes tienen tamaño limitado.

6. Memoria Virtual y Paginación (8% de las preguntas)

- Paginación en demanda
- Overcommitment
- TLB
- Protección de memoria

Consejos: Solo se asignan marcos cuando se accede a las páginas con overcommitment

7. Concurrencia y Sincronización (8% de las preguntas)

- Condiciones de carrera
- Mutex vs spinlock
- Deadlock

Consejos: Identifica las secciones críticas. ¿Está el lock protegiendo todo lo necesario?

8. Sistemas de Ficheros (7% de las preguntas)

- I-nodos
- Enlaces duros
- st_nlink

Consejos: Un i-nodo solo se borra cuando st_nlink = 0

RESUMEN DE RESPUESTAS CORRECTAS

Sección	Tema	Preguntas	Respuestas Clave
1	Punteros	7	Control de flujo, aritmética, casting
2	Memoria	6	malloc/free, sizeof, leaks
3	Procesos	5	fork devuelve 0 en hijo, PID en padre
4	Syscalls	8	Errores, buffering, exec reemplaza
5	Ficheros	4	fopen usa buffering, i-nodos
6	Pipes	4	Tamaño limitado, SIGPIPE
7	Shell	5	argv/argc, variables no heredadas
8	Memoria Virtual	3	Paginación bajo demanda
9	Planificación	3	FCFS, Round-Robin, renice

Sección	Tema	Preguntas	Respuestas Clave
10	Concurrencia	6	Condiciones de carrera
11	ELF	5	.text, lazy binding, RELRO
12	Sistemas de Ficheros	2	I-nodos, FAT, enlaces
13	Señales	3	SIGKILL, SIGPIPE, máscara

RECOMENDACIONES DE ESTUDIO

1. **Dibuja siempre:** Procesos, memoria, estructura de datos
2. **Trace el código:** Paso a paso, variable por variable
3. **Memoriza valores de retorno:** Especialmente fork(), read(), write()
4. **Entiende los errores comunes:**
    - Leaks de memoria
    - Condiciones de carrera
    - Deadlock en pipes
    - Buffering
5. **Practica con código real:** Compila y ejecuta los ejemplos
6. **Lee los apuntes y el libro:** Especialmente sobre syscalls

Este documento ha sido organizado para facilitar tu estudio. Ahora puedes hacer preguntas específicas sobre cualquiera de estos temas basándote en el temario de la asignatura.