APUNTES TEMAS 5, 6, 7 y 8 - SISTEMAS OPERATIVOS

Programación para Examen

TEMA 5: FICHEROS Y SISTEMAS DE FICHEROS

Conceptos Fundamentales

1. ¿Qué es un fichero en el contexto de sistemas operativos?

Un fichero es una abstracción proporcionada por el SO que representa datos persistentes. Puede ser:

- Datos que persisten tras la ejecución del proceso
- Un dispositivo
- Almacenamiento volátil (ramfs)
- Una interfaz general para cualquier recurso

Requisitos fundamentales:

- Gran tamaño
- Durabilidad (persistencia)
- Acceso concurrente múltiples procesos
- Protección

2. Operaciones básicas sobre ficheros

Las operaciones POSIX básicas son:

- **open()**: Abre un fichero y retorna un descriptor
- **read()**: Lee bytes del fichero
- **write()**: Escribe bytes en el fichero
- **close()**: Cierra el fichero
- **lseek()**: Cambia la posición de lectura/escritura

3. ¿Qué es un descriptor de fichero?

Un descriptor de fichero (file descriptor o FD) es:

- Un número entero que identifica un fichero abierto para un proceso
- El índice en la **tabla de descriptores de fichero** del proceso
- Localizado en la estructura del proceso en el kernel

Descriptores estándar:

- **0**: stdin (entrada estándar)
- **1**: stdout (salida estándar)
- **2**: stderr (salida de errores)

4. Explicación de la llamada open()

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Parámetros:

- `pathname` : ruta del fichero
- `flags` : combinación de banderas
  - `O_RDONLY` : solo lectura
  - `O_WRONLY` : solo escritura
  - `O_RDWR` : lectura y escritura
  - `O_CREAT` : crear si no existe (requiere modo)
  - `O_TRUNC` : truncar a longitud 0
  - `O_APPEND` : escribir siempre al final
  - `O_CLOEXEC` : cerrar automáticamente con exec
- `mode` : permisos (si O_CREAT está presente)

Retorna: descriptor (número ≥ 0), o -1 en error

Comportamiento importante:

- El dueño del fichero será el UID del proceso
- El grupo puede ser el GID del proceso o del directorio (depende del bit set-group-ID)
- Los permisos se afectan por la máscara umask
- El offset comienza en 0

- Si abres el mismo fichero dos veces, obtienes dos descriptores diferentes con offsets independientes

5. ¿Qué es umask?

`umask` es una máscara de creación de ficheros:

- Es una propiedad del proceso (se hereda del padre)
- Los bits a 1 en la máscara se excluyen de los permisos
- Fórmula: `permisos_finales = mode & ~umask`

Ejemplo: Si umask es 077 y creas un fichero con modo 755, los permisos finales serán 700

```
mode_t umask(mode_t mask);
```

6. Llamada read()

```
ssize_t read(int fd, void *buf, size_t count);
```

Comportamiento:

- Lee **como mucho** `count` bytes del fichero
- Puede leer **menos bytes** sin ser error (lectura corta)
- Si retorna 0 bytes: fin de fichero (EOF)
- Si retorna -1: error

Importante: `read()` es una llamada bloqueante. Si no hay datos, espera.

7. Llamada write()

```
ssize_t write(int fd, const void *buf, size_t count);
```

Comportamiento:

- Escribe `count` bytes en el fichero
- Si retorna número ≠ `count` : se debe considerar error
- El offset se actualiza automáticamente

8. Llamada lseek()

```
off_t lseek(int fd, off_t offset, int whence);
```

Parámetros whence:

- `SEEK_SET` : offset absoluto desde el principio
- `SEEK_CUR` : offset relativo desde posición actual
- `SEEK_END` : offset relativo desde el final

Retorna: nueva posición del offset, o -1 en error

9. Llamadas pread() y pwrite()

```
ssize_t pread(int d, void *buf, size_t nbyte, off_t offset);
ssize_t pwrite(int fildes, const void *buf, size_t nbyte, off_t offset);
```

Son versiones de read/write que **no modifican el offset del descriptor**. Útiles para acceso concurrente.

10. Llamada close()

```
int close(int fd);
```

- Cierra el fichero y libera recursos
- Retorna -1 si hay error
- Importante: mantener libres los descriptores 0, 1 y 2 (entrada, salida, errores)

11. Llamada access()

```
int access(const char *pathname, int mode);
```

Comprueba si se puede acceder al fichero:

- `F_OK` : existe
- `R_OK` : se puede leer
- `W_OK` : se puede escribir
- `X_OK` : se puede ejecutar
- Retorna 0 si sí, -1 si no

12. Llamada stat()

```
int stat(const char *pathname, struct stat *statbuf);
```

Lee los metadatos de un fichero en `struct stat` :

- `st_ino` : i-nodo del fichero
- `st_mode` : permisos y tipo
- `st_nlinks` : número de enlaces duros (nombres)
- `st_uid` : propietario (UID)
- `st_gid` : grupo (GID)
- `st_size` : tamaño en bytes
- `st_atime` : última lectura
- `st_mtime` : última modificación de datos
- `st_ctime` : último cambio de metadatos

Nota: `stat()` atraviesa enlaces simbólicos. Para el enlace sí mismo, usa `lstat()` .

13. Llamadas dup() y dup2()

```
int dup(int fildes);
int dup2(int fildes, int fildes2);
```

- `dup()` : duplica el descriptor en el primer slot libre
- `dup2()` : duplica en el slot específico, cerrándolo primero si está abierto
- Comparten offset

Útil para: redirecciones en shell, pipes

14. Llamada unlink()

```
int unlink(const char *pathname);
```

- Elimina un nombre (enlace duro) del fichero
- Si es el último nombre y no hay procesos con el fichero abierto, se libera
- Retorna -1 en error

Discos y Particiones

15. ¿Qué es un bloque de disco?

- **Bloque físico (sector):** Unidad de HW, tradicionalmente 512 bytes, ahora 4 KB
- **Bloque lógico:** Algunos discos traducen de bloques lógicos a físicos
- **Bloque de SO:** El SO también maneja bloques a su nivel de abstracción

16. Direccionamiento en disco: CHS vs LBA

- **CHS (Cylinder, Head, Sector):** Esquema antiguo que usa geometría real del disco
- **LBA (Linear Block Addressing):** Esquema moderno, direccionamiento lineal

17. Algoritmos de planificación de disco

Organizan las operaciones de I/O:

1. **FIFO:** Primero en llegar, primero en servir
   - Justo, no altera orden
   - Lento en discos mecánicos
2. **Shortest Seek First (SSF):**
   - Atiende la solicitud con menor desplazamiento
   - Injusto, puede provocar hambruna
   - Desordena operaciones
3. **Ascensor (Elevator):**
   - Compromiso entre justicia y eficiencia
   - No provoca hambruna
   - Desordena operaciones

18. Particiones BIOS (MBR)

- **MBR (Master Boot Record):** Primer bloque, 512 bytes
- **Cargador primario:** 440 bytes
- **Tabla de particiones:** 4 entradas de 16 bytes
  - Arrancable/no arrancable
  - Tipo de partición
  - Direcciones CHS y LBA del primer/último bloque
- **Limitación:** Direcciones de 32 bits → máximo 2 TB

19. Particiones UEFI (GPT)

- **GPT (GUID Partition Table):** Reemplaza MBR
- **LBA 0:** Legacy MBR
- **LBA 1:** Cabecera GPT
- **LBA 2-X:** Tabla de particiones (mínimo 16 KB)
  - Entradas de 128 bytes
  - Información: tipo, LBA inicio/fin, atributos, nombre
- **Final del disco:** Copia de tabla y cabecera (redundancia)
- **Ventaja:** Direcciones de 64 bits, mucho más espacio

20. Volúmenes Lógicos (LVM)

Permiten combinar múltiples discos físicos:

- **Physical Volume (PV):** Partición o disco físico
- **Physical Extent (PE):** Unidad dentro del PV
- **Volume Group (VG):** Agrupación de PVs
- **Logical Volume (LV):** Dispositivo virtual que comparte PEs de distintos PVs

Sistemas de Ficheros

21. ¿Qué es un sistema de ficheros?

Componente que proporciona la abstracción de fichero:

- Implementa operaciones: open, read, write, close
- Gestiona la asignación de bloques a ficheros
- Suele estar en el kernel, pero puede estar en espacio de usuario (FUSE)

22. Métodos de asignación de bloques

1. **Asignación contigua:** Bloques consecutivos
   - Rápido para lectura secuencial
   - Fragmentación externa
2. **Asignación enlazada:** Bloques con puntero al siguiente
   - Evita fragmentación externa
   - Lento para acceso aleatorio
3. **Asignación enlazada con tabla (FAT):**
   - Tabla centralizada de punteros
   - Menos viajes al disco para acceso aleatorio

4. **Asignación indexada (i-nodo):**
    - Array de punteros a bloques en una estructura especial (i-nodo)
    - Rápido, flexible, usado por Unix/Linux

23. VFS (Virtual Filesystem Switch)

Interfaz común en el kernel para todos los sistemas de ficheros:

- Permite múltiples FS simultáneamente
- Los FS se implementan como módulos del kernel

24. FUSE (Filesystem in Userspace)

Módulo del kernel que permite:

- Implementar sistemas de ficheros en espacio de usuario
- Integración transparente con VFS

25. Espacio de nombres (namespace)

El árbol de ficheros del sistema:

- Se construye con operaciones de montaje
- Se puede montar nuevos árboles (FS) en puntos específicos
- En Linux moderno, se pueden tener namespaces diferentes por grupos de procesos

Preguntas de Examen - Tema 5

P1. Explica qué sucede cuando haces `int fd = open("archivo.txt", O_RDONLY | O_CREAT);`
Respuesta esperada: Error. O_CREAT requiere un modo como tercer argumento. La llamada correcta sería `open("archivo.txt", O_RDONLY | O_CREAT, 0644);` pero incluso así, O_RDONLY solo lectura con O_CREAT no tiene sentido práctico.

P2. ¿Cuál es la diferencia entre `read()` retornando 0 y -1?
Respuesta: 0 indica fin de fichero (EOF), -1 indica error (consultar errno).

P3. Si abres un fichero dos veces con `open()`, ¿comparten offset?
Respuesta: No. Cada descriptor tiene su propio offset independiente.

P4. ¿Qué hace `umask(077)` y cuál es el resultado final de crear un fichero con modo 666?
Respuesta: Máscara 077 excluye permisos de grupo y otros. 666 & ~077 = 600 (rw-------)

P5. Explica los pasos en MBR vs GPT para particiones
Respuesta: MBR usa 512 bytes con 4 particiones, limitado a 2TB. GPT usa estructura moderna con redundancia, soporta 64 bits y más particiones.

TEMA 6: GESTIÓN DE MEMORIA

Asignación Dinámica de Memoria

1. ¿Qué es la asignación dinámica de memoria?

Proceso de reservar y liberar bloques de memoria en tiempo de ejecución:

- **Gestión explícita:** Manual (malloc/free en C)
- **Gestión implícita:** Automática (garbage collector en Java)

2. Fragmentación externa

Ocurre cuando quedan huecos inservibles entre bloques asignados:

- **Ejemplo:** 12 butacas libres en cine pero no hay 4 contiguas
- **Ley del 50%:** En media, 50% de la memoria se pierde por fragmentación
- **Solución:** Compactación (no aplicable con memoria dinámica sin soporte del SO)

3. Fragmentación interna

Espacio desperdiciado dentro de un bloque asignado:

- **Ejemplo:** Pedir 100 bytes pero obtener bloque de 128 bytes
- **Solución:** Mejores ajustes de tamaño

4. Políticas de asignación dinámica

1. **First Fit:** Primer bloque donde cabe
    - Rápido, simple
    - Comportamiento bueno
2. **Next Fit:** Donde cabía la anterior, desde ahí
    - Empíricamente peor que First Fit
    - Más fragmentación
3. **Best Fit:** Bloque que mejor se ajusta

    - Tiende a dejar fragmentos pequeños
    - Más lento
4. **Worst Fit:** Bloque que peor se ajusta
    - Peor rendimiento general
5. **Quick Fit:** Listas de tamaños populares
    - Segregación por tamaño
    - Rápido, comportamiento bueno
    - Más complejo

5. Ejemplo real de malloc en libc

- `<= 64 bytes` : Quick Fit con trozos segregados
- `64 - 512 bytes` : Política mixta
- `>= 512 bytes` : Best Fit puro
- `>= 128 KB` : Región nueva con mmap (no en heap)

6. Mecanismos de implementación

Estructura de datos:

- Lista enlazada de bloques libres/ocupados
- Headers (cabeceras) y footers (pies) para metadatos
- Coalescing: fusión de bloques libres contiguos

Metadatos en header/footer:

- Estado (libre/ocupado)
- Tamaño del bloque
- Puntero al siguiente/anterior libre

7. ¿Qué es coalescing?

Proceso de fusionar dos bloques libres contiguos:

- Reduce fragmentación
- Puede hacerse: inmediatamente al liberar, o de forma lazy
- Headers/footers facilitan localizar bloques adyacentes

Memoria Virtual

8. ¿Por qué memoria virtual?

Beneficios:

- **Protección:** Un proceso no accede a memoria de otro o del kernel
- **Simplicidad:** Para el proceso, memoria es contigua
- **Abstracción:** Proceso cree que está solo en su máquina
- **Debugging:** Misma organización para todas las ejecuciones
- **Vinculación:** Resolución de símbolos en tiempo de ejecución
- **Compartición:** Misma región física en múltiples procesos
- **Swapping:** Usar disco como extensión de RAM

9. ¿Cómo funciona la memoria virtual?

La **MMU (Memory Management Unit)**:

- Traduce direcciones virtuales → direcciones físicas
- El programa nunca ve direcciones reales
- El kernel instala tablas de traducción en hardware

Por proceso:

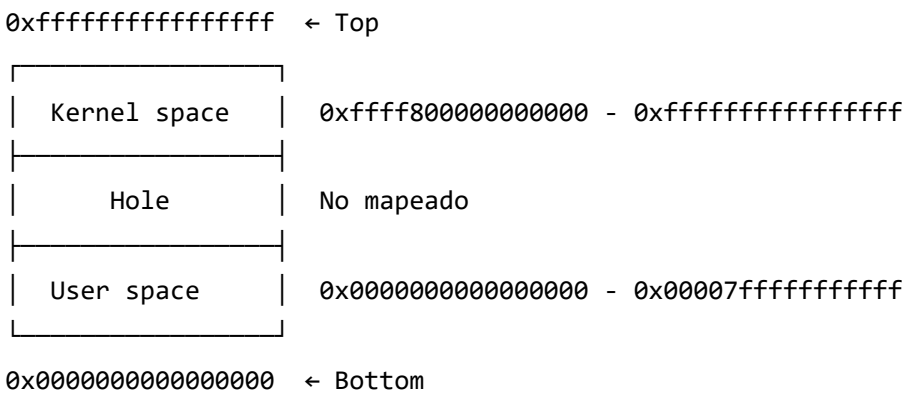
- Cada proceso tiene su propio espacio de direcciones virtuales
- El kernel mantiene tabla de páginas para cada proceso
- Al cambiar contexto, se cambia tabla de páginas en hardware

10. Espacio de direcciones virtual en un proceso

Estructura típica (de menor a mayor dirección):

- **.text:** Código del programa (read-only)
- **.data:** Datos inicializados (read-write)
- **.bss:** Datos no inicializados (read-write)
- **Heap:** Memoria dinámica (crecimiento ascendente)
- **Stack:** Pila de ejecución (crecimiento descendente)

11. División kernel/usuario en Linux x86_64



- **User space:** Donde ejecutan procesos normales
- **Kernel space:** Exclusivo del kernel
- **Hole:** Brecha de protección

12. Paginación

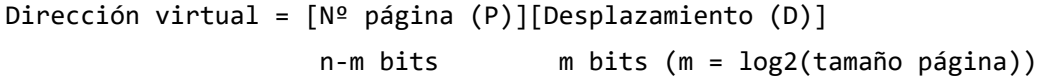
Dividen memoria en bloques del mismo tamaño:

- **Marco:** Bloque de memoria física
- **Página:** Bloque de memoria virtual
- **Tamaño:** Típicamente 4 KB, determinado por hardware

Ventaja: Permite fragmentación no contigua (usamos marcos no contiguos)

13. Tabla de páginas

Estructura de traducción página → marco:



Traducción:

- P indexa tabla de páginas → obtiene marco M
- D desplazamiento se usa directo
- Dirección física = [M][D]

14. Problema de tamaño de tabla de páginas

Para espacios muy grandes (248 direcciones):

- Tabla de páginas sería gigante si entera
- **Solución:** Paginación multinivel

15. Paginación multinivel

Divide tabla en N niveles:

- **Ejemplo 2 niveles:** Dirección = [p1][p2][d]
  - p1: índice en tabla exterior
  - p2: índice en tabla interior
  - d: desplazamiento

Ventaja: Solo asignar tablas necesarias

Desventaja: Hasta N+1 accesos a memoria (mitigado por TLB)

16. TLB (Translation Look-aside Buffer)

Cache pequeño de tabla de páginas:

- ≈ 100-1000 entradas
- Acceso muy rápido (SRAM, 1 ciclo)
- Especial para cada proceso

Funcionamiento:

1. Si traducción está en TLB → acceso directo
2. Si no → buscar en tabla de páginas → insertar en TLB

Métrica: Hit Ratio = % de aciertos

Tiempo efectivo:

```
T_efectivo = Hit_Ratio × T_acierto + (1-Hit_Ratio) × T_fallo
```

17. Page Table Entry (PTE) - Bits importantes

- **Bit de presente:** Indica si página tiene traducción válida
- **Bit de modo:** Lectura/escritura/ejecución
- **Bit de acceso:** Registra si se ha accedido
- **Bit de dirty:** Registra si se ha modificado
- **Bits especiales:** Caché, ejecución, etc.

18. Copy-on-Write

Optimización usada en fork():

- Padre e hijo comparten páginas inicialmente
- Cuando uno intenta escribir → fallo de página
- Se copia la página en ese momento
- Cada uno tiene su copia

19. Paginación en demanda

Carga páginas solo cuando se necesitan:

- Aproximación perezosa (lazy)
- Se ahorra memoria si hay páginas nunca usadas

Proceso:

1. Proceso accede dirección sin mapeo
2. Fallo de página → trap al kernel
3. Kernel busca marco libre
4. Carga contenido (si en fichero)
5. Modifica tabla de páginas
6. Reinicia instrucción

20. mlock() y mlockall()

```
int mlock(const void *addr, size_t len);
int mlockall(int flags);
```

Impide que páginas vayan a swap:

- Solo procesos privilegiados
- Útil para aplicaciones tiempo-real

Intercambio (Swapping)

21. ¿Qué es swap?

Mover memoria de procesos a disco:

- Cuando RAM está llena

- Trae de vuelta cuando se necesita
- **Muy lento:** 1200 Mbit/s (SATA) vs 136 Gbit/s (DDR3)
- Actualmente menos útil, pero sigue usándose

Preguntas de Examen - Tema 6

P1. Explica fragmentación externa vs interna con ejemplos
Respuesta: Externa: 12 butacas libres pero no 4 juntas. Interna: pedir 100B, obtener 128B. Problema diferente, soluciones diferentes.

P2. ¿Cuál es la diferencia en tamaño de dirección entre tabla de un nivel vs dos niveles?
Respuesta: Un nivel: 2^48 × 8 bytes ≈ 256 TB. Dos niveles: solo asigna tablas necesarias, mucho menor.

P3. Si Hit Ratio TLB es 99% con tacierto=20ns y tfallo=100ns, ¿tiempo efectivo?
Respuesta: 0.99 × 120 + 0.01 × 220 = 118.8 + 2.2 = 121 ns

P4. ¿Qué es copy-on-write en fork()?
Respuesta: Padre e hijo comparten páginas. Al escribir, se copia. Ahorra memoria inicialmente.

P5. ¿Por qué la paginación en demanda es mejor que asignar todo de golpe?
Respuesta: Ahorra memoria si hay páginas nunca usadas. Costo: manejar fallos de página.

TEMA 7: COMUNICACIÓN ENTRE PROCESOS (IPC)

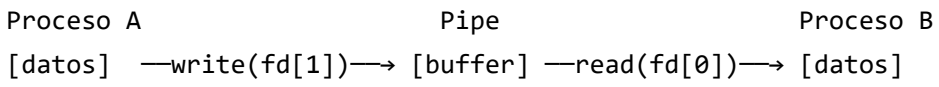
Pipes (Tuberías)

1. ¿Qué es un pipe?

Mecanismo de comunicación entre procesos:

- Extremos conectados: escribir en uno, leer en otro
- Simplex: solo en una dirección
- Buffer limitado (típicamente 65 KB en Linux)

2. Funcionamiento de un pipe



- fd[0]: extremo de lectura
- fd[1]: extremo de escritura

3. Llamada al sistema pipe()

```c
int pipe(int fd[2]);
```

- Crea nuevo pipe
- fd[0] y fd[1] se rellenan con descriptores
- Se debe llamar **antes de fork()** para compartir
- Retorna 0 si OK, -1 en error

4. Reglas de uso de pipes

- Cerrar extremos no usados
- **Leer de pipe vacío:** Bloquea
- **Escribir en pipe lleno:** Bloquea (buffer ~65 KB)
- **Leer si nadie escribiendo:** Retorna 0 bytes (EOF)
- **Escribir si nadie leyendo:** Señal SIGPIPE (error)
- **Importante:** Procesos deben leer/escribir en paralelo, no secuencialmente (riesgo de interbloqueo)

5. Pipes en el shell

```
$> cat archivo.txt | grep "palabra" | wc -l
```

El shell:

1. Crea pipes entre procesos
2. Redirige fd[1] de cat a fd[0] de grep
3. Redirige fd[1] de grep a fd[0] de wc
4. Espera a que terminen

6. Ejemplo práctico: productor-consumidor

```
if (pipe(fd) == -1) {
    perror("pipe");
    exit(1);
}
pid_t pid = fork();
if (pid == -1) {
    perror("fork");
    exit(1);
} else if (pid == 0) {
    // Hijo: solo lectura
    close(fd[1]);
    read(fd[0], buffer, sizeof(buffer));
} else {
    // Padre: solo escritura
    close(fd[0]);
    write(fd[1], "datos", 5);
    wait(NULL);
}
```

FIFOs (Named Pipes)

7. ¿Qué es un FIFO?

Pipe con nombre en el espacio de ficheros:

- Ruta como cualquier fichero
- Se crean con `mkfifo` en shell
- Se elimina con `rm`

8. Creación de FIFO

```
$> mkfifo /tmp/myfifo
```

En C:

```
int mkfifo(const char *path, mode_t mode);
```

9. Comportamiento de FIFO

Lectura:

- Open solo-lectura: bloquea si nadie escribiendo
- Open no-bloqueante: falla si nadie escribiendo

Escritura:

- Open solo-escritura: bloquea si nadie leyendo
- Open no-bloqueante: falla si nadie leyendo

Como pipes: Leer pipe vacío = 0 bytes, escribir sin lector = SIGPIPE

Señales

10. ¿Qué es una señal?

Mecanismo de notificación asíncrona a un proceso:

- Interrumpe ejecución
- Puede ignorarse, manejarse, o tomar acción por defecto
- Difíciles de usar correctamente

11. Tipos de señales

Síncronas: Consecuencia de ejecución del proceso (SIGSEGV, SIGFPE)
Asíncronas: Enviadas por otros procesos o eventos externos (SIGINT, SIGTERM)

12. Acciones por defecto

Para cada señal:

- Terminar proceso
- Terminar con core dump
- Ignorar
- Suspender
- Reanudar

13. Señales más importantes

Número	Nombre	Acción defecto	Descripción
1	SIGHUP	Terminar	Línea de terminal desconectada
2	SIGINT	Terminar	Ctrl-C
3	SIGQUIT	Core dump	Ctrl-\
9	SIGKILL	Terminar	Matar proceso (no ignorable)
11	SIGSEGV	Core dump	Violación de segmentación
13	SIGPIPE	Terminar	Escribir en pipe sin lector
15	SIGTERM	Terminar	Señal de terminación software
17	SIGSTOP	Parar	Parar proceso (no capturible)
20	SIGCHLD	Ignorar	Cambio de estado en hijo

14. Llamada al sistema kill()

```
int kill(pid_t pid, int sig);
```

- Envía señal a proceso
- Restricción: UID debe ser mismo que destino (o ser root)
- Excepción: SIGCONT se puede enviar a descendientes

15. Llamada al sistema signal()

```
typedef void (*sig_t) (int);
sig_t signal(int sig, sig_t func);
```

Registra manejador para señal:

- `func` : función manejadora (void retorna, int parámetro = número señal)
- `SIG_DFL` : acción por defecto
- `SIG_IGN` : ignorar

Limitación: Versión simplificada de sigaction(), comportamiento no portable

16. Manejadores de señales

Requisitos:

- Ser reentrantes
- No usar variables globales/estáticas
- No llamar funciones no reentrantes (malloc, free)
- Ser pequeños
- No modificar errno

17. Bloqueo de señales

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
```

- Bloquear señales temporalmente
- Diferente de ignorar: si llega, queda pendiente
- Al desbloquear, se entrega la pendiente

18. Alarmas

```
unsigned int alarm(unsigned int seconds);
```

Programa temporizador que envía SIGALRM:

- `alarm(0)` cancela
- Si se llama dos veces, sobrescribe
- Uso: implementar timeouts

19. siginterrupt() - Reinicio de llamadas

```
int siginterrupt(int sig, int flag);
```

Algunas llamadas son "lentas" (se bloquean): read, write, open, etc.

- Al recibir señal, se interrumpen
- `siginterrupt(sig, 0)` : reinicia automáticamente
- `siginterrupt(sig, 1)` : retorna error EINTR

20. Ignoring SIGCHLD para evitar zombies

```
signal(SIGCHLD, SIG_IGN);
```

Si no esperas por hijos con wait(), ignora SIGCHLD para que no se conviertan en zombies.

Job Control

21. Sesión y grupo de procesos

Sesión:

- Grupo de procesos
- Un proceso líder (SID = PID del líder)

Grupo de procesos:

- Procesos relacionados
- Un proceso líder (PGID = PID del líder)
- Un job es un grupo de procesos

22. Control en terminal

- Como mucho un grupo en foreground
- Terminal envía señales solo a foreground (ej: Ctrl-C envía SIGINT)
- Si proceso en background lee de terminal: recibe SIGTTIN

23. Llamadas para gestionar sesiones

```
pid_t setpgid(pid_t pid, pid_t pgid);
pid_t setsid(void);
```

- `setpgid()` : cambiar grupo de procesos
- `setsid()` : crear nueva sesión

24. Demonios

Procesos de servicio sin terminal:

1. Crear nueva sesión con `setsid()`
2. Cambiar directorio a /

3. Cerrar entrada/salida estándar
4. Redirigir a /dev/null o fichero de log

```
daemon(1, 1);  // función de libc que hace esto
```

Preguntas de Examen - Tema 7

P1. ¿Qué sucede si escribes en un pipe y nadie está leyendo?
Respuesta: Se envía señal SIGPIPE al proceso escritor, que lo termina por defecto.

P2. Explica cómo el shell implementa `cat file | grep word | wc -l`
Respuesta: Crea tres procesos con pipes entre ellos. fd[1] de cat → fd[0] de grep, fd[1] de grep → fd[0] de wc.

P3. ¿Diferencia entre bloquear una señal e ignorarla?
Respuesta: Ignorar: descarta la señal. Bloquear: la señal se queda pendiente hasta desbloquear.

P4. ¿Qué es un demonio? ¿Cómo se crea?
Respuesta: Proceso de servicio sin terminal. Se crea con setsid(), cambiar dir a /, cerrar stdin/out/err.

P5. ¿Por qué ignorar SIGCHLD previene zombies?
Respuesta: Con SIG_IGN, el kernel automáticamente recolecta procesos terminados sin necesidad de wait().

TEMA 8: CONCURRENCIA BÁSICA

Conceptos Fundamentales

1. ¿Qué es una condición de carrera?

Cuando el resultado depende del orden de ejecución:

- No es reproducible (ocurre aleatoriamente)
- Peor tipo de bug
- Más grave con memoria compartida

2. Ejemplo simple de condición de carrera

```
int x = 0;  // compartida

// Flujo A y B ejecutan esto concurrentemente:
for (int i = 0; i < 10; i++) {
    int aux = x;
    aux = aux + 1;
    x = aux;
}
```

¿Valores finales posibles?

- Mínimo: 2 (si procesos interfieren)
- Máximo: 20 (si ejecutan secuencialmente)
- Típicamente: algo entre 2 y 20

3. Incremento no atómico

```
x++;  // esto parece atómico pero NO lo es
```

Se expande a:

```
MOV EAX, [x]      ; leer x
ADD EAX, 1        ; incrementar
MOV [x], EAX      ; escribir
```

Múltiples procesos pueden interferir entre instrucciones.

4. Atomicidad

Una operación es atómica si:

- Ningún otro flujo puede interferir
- Se ejecuta de forma indivisible
- Requiere soporte hardware

5. Sección crítica

Región del código donde se accede a recurso compartido:

- Necesita exclusión mutua
- Solo un flujo puede estar dentro

6. Exclusión mutua

Mecanismo de sincronización:

- Garantiza que solo un flujo entra a la sección crítica
- Otros deben esperar

Hardware Atómico

7. Test-and-Set (TAS)

Operación atómica de hardware:

```
Si flag == false:
    flag = true
    retorna false
Si flag == true:
    flag = true
    retorna true
```

Permite implementar locks.

8. XCHGL (en AMD64)

```
XCHGL AX, (BX)  ; intercambia AX con *BX atómicamente
```

Equivalente a TAS en x86-64.

9. Implementación de TAS en Plan 9/AMD64

```
TEXT _tas(SB),$0
    MOVL $0xdeadead,AX  ; valor para poner a true
    MOVL 1+0(FP),BX      ; puntero al flag
    XCHGL AX,(BX)        ; intercambia atómicamente
    RET                  ; retorna valor anterior
```

Primitivas de Sincronización

10. Lock (Cerrojo)

Abstracción de más alto nivel que TAS:

Pseudocódigo:

```
acquire(lock):
    while (test_and_set(lock)) {
        // espera activa (spin)
    }

release(lock):
    lock = 0
```

Problema: Espera activa (busy waiting) consume CPU

11. Mutex (Mutual Exclusion)

Lock mejorado:

- Procesos que no pueden coger lock se duermen
- Cuando se libera, se despierta uno
- Más eficiente que spin

12. Semáforo

Variable de contador:

Operaciones:

- `wait()` : decremente. Si 0 antes, espera
- `signal()` : incremente. Si había esperando, despierta

Casos de uso:

- Exclusión mutua (inicializar a 1)
- Sincronización de eventos (inicializar a 0)

13. Variable de condición

Permite que procesos esperen a que se cumpla condición:

Operaciones:

- `wait()` : libera lock, espera condición
- `signal()` : despierta un esperador
- `broadcast()` : despierta todos

14. Monitor

Clase/estructura con:

- Datos privados
- Métodos que acceden sincronizadamente
- Lock implícito
- Variables de condición

Ejemplo en pseudocódigo:

```
Monitor Buffer {
    private dato[100]
    private int size = 0
    private Condition notEmpty, notFull

    put(x) {
        acquire lock
        while (size == 100) {
            wait(notFull)
        }
        dato[size] = x
        size++
        signal(notEmpty)
        release lock
    }

    get() {
        acquire lock
        while (size == 0) {
            wait(notEmpty)
        }
        x = dato[size-1]
        size--
        signal(notFull)
        release lock
        return x
    }
}
```

Problemas Clásicos de Sincronización

15. Productor-Consumidor

Dos procesos:

- Productor: produce datos, coloca en buffer
- Consumidor: toma datos del buffer

Sincronización:

- Si buffer lleno: productor espera
- Si buffer vacío: consumidor espera

16. Lectores-Escritores

Múltiples lectores vs escritor exclusivo:

- Múltiples lectores pueden acceder simultáneamente
- Si escritor: nadie más puede acceder
- Si hay lectores: escritor espera

17. Filósofos Comensales

Problema clásico de interbloqueo:

- 5 filósofos, 5 palillos
- Cada filósofo necesita 2 palillos para comer
- Potencial de interbloqueo si todos toman palillo izquierdo

Solución: Orden específico, máximo 4 comiendo, etc.

Preguntas de Examen - Tema 8

P1. ¿Por qué `x++` no es atómico?
Respuesta: Se expande a: leer, incrementar, escribir. Otros procesos pueden interferir entre estos pasos.

P2. Explica TAS y cómo permite implementar un lock
Respuesta: TAS devuelve valor anterior mientras pone a true. Si retorna false, tienes el lock. Si true, alguien más lo tiene.

P3. ¿Diferencia entre mutex y semáforo?
Respuesta: Mutex es lock para exclusión mutua. Semáforo es contador para sincronización general (puede inicializarse a valores > 1).

P4. En problema Productor-Consumidor, ¿qué pasa si buffer se llena y productor sigue escribiendo?
Respuesta: El productor debe esperar (bloquearse) hasta que consumidor libere espacio.

P5. ¿Qué es un interbloqueo? Da ejemplo.
Respuesta: Dos procesos esperan mutuamente. Ej: A tiene X y espera Y, B tiene Y y espera X.

EJEMPLOS DE CÓDIGO PRÁCTICO

Ejemplo 1: Lectura/Escritura de Ficheros

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>

int main() {
    char buffer[256];
    int fd;

    // Crear y escribir
    fd = open("test.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    if (write(fd, "Hola mundo", 10) == -1) {
        perror("write");
        close(fd);
        return 1;
    }

    close(fd);

    // Leer
    fd = open("test.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    ssize_t n = read(fd, buffer, sizeof(buffer) - 1);
    if (n == -1) {
        perror("read");
        close(fd);
        return 1;
    }
```

```
    buffer[n] = '\0';
    printf("Contenido: %s\n", buffer);
    close(fd);

    return 0;
}
```

Ejemplo 2: Pipe Simple

```c
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>

int main() {
    int fd[2];
    pid_t pid;
    char buffer[256];

    if (pipe(fd) == -1) {
        perror("pipe");
        return 1;
    }

    pid = fork();
    if (pid == -1) {
        perror("fork");
        return 1;
    }

    if (pid == 0) {
        // Hijo: lector
        close(fd[1]);
        ssize_t n = read(fd[0], buffer, sizeof(buffer) - 1);
        buffer[n] = '\0';
        printf("Hijo leyó: %s\n", buffer);
        close(fd[0]);
    } else {
        // Padre: escritor
        close(fd[0]);
        write(fd[1], "Datos del padre", 15);
        close(fd[1]);
        wait(NULL);
    }

    return 0;
}
```

Ejemplo 3: Manejo de Señales

```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void manejador(int sig) {
    printf("Recibida señal %d\n", sig);
}

int main() {
    signal(SIGINT, manejador);
    printf("Esperando Ctrl-C (5 segundos)...\n");
    sleep(5);
    printf("Tiempo terminado\n");
    return 0;
}
```

Ejemplo 4: Sincronización Simple con Spin Lock

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

int contador = 0;
int lock = 0;

void lock_acquire() {
    while (__sync_val_compare_and_swap(&lock, 0, 1) != 0) {
        // espera activa
    }
}

void lock_release() {
    __sync_synchronize();
    lock = 0;
}

void* thread_func(void* arg) {
    for (int i = 0; i < 1000; i++) {
        lock_acquire();
        contador++;
        lock_release();
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_func, NULL);
    pthread_create(&t2, NULL, thread_func, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Contador final: %d (esperado 2000)\n", contador);
    return 0;
}
```

RESUMEN DE CONCEPTOS CLAVE

Tema	Concepto	Definición
5	FD	Número que identifica fichero abierto
5	Bloque	Unidad de almacenamiento en disco
5	i-nodo	Estructura con metadatos de fichero
6	Página	Unidad de memoria virtual
6	Marco	Unidad de memoria física
6	TLB	Cache de traducciones página→marco
6	MMU	Hardware que traduce direcciones
7	Pipe	Canal de comunicación unidireccional
7	FIFO	Pipe con nombre en filesystem
7	Señal	Notificación asíncrona a proceso
8	Carrera	Resultado depende de orden ejecución
8	Atómico	Operación indivisible, no interrumpible
8	Lock	Mecanismo de exclusión mutua