LABORATORIO 1: "El Nacimiento de un Proceso"

Objetivo

Entender la diferencia entre `fork()` y `exec()` viendo cómo la shell crea procesos.

nacimiento.c

bash

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

int main() {
    printf("ANTES: Soy PID %d\n", getpid());

    pid_t hijo = fork();

    if (hijo == 0) {
        // ========== HIJO ==========
        printf("HIJO: Soy PID %d, mi padre es %d\n", getpid(), getppid());
        printf("HIJO: Voy a dormir 10 segundos para que me observes...\n");

        sleep(10);  // ← Pausa de 10 segundos, tiempo suficiente para verlo

        printf("HIJO: Ahora me transformo en 'date'...\n");
        execlp("date", "date", NULL);
        printf("HIJO: Esta línea NUNCA se ejecuta\n");
    } else {
        // ========== PADRE ==========
        printf("PADRE: Mi hijo es PID %d, espero...\n", hijo);

        printf("\n>>> AHORA EJECUTA EN OTRA TERMINAL: pstree -p | grep
nacimiento <<<\n");
        printf(">>> (Tienes 10 segundos) <<<\n\n");

        sleep(10);  // ← Padre también espera, para sincronizarse con el hijo

        wait(NULL);
        printf("PADRE: Mi hijo terminó. Yo sigo siendo PID %d\n", getpid());
    }

    return 0;
}
```

Compilar

```
gcc nacimiento.c -o nacimiento
```

Experimento A: Observar el árbol de procesos

bash

```
# Terminal 1: ejecutar con espera
```

```
./nacimiento &

# Terminal 2: ver el árbol en tiempo real
pstree -p | grep nacimiento
watch -n 0.5 'pstree -p | grep nacimiento'
```

Experimento B: Ver las syscalls con strace

bash

```
# Ver el flujo completo de syscalls
strace -e trace=process,write ./nacimiento 2>&1

# Contar cuántas de cada syscall ocurren
strace -c ./nacimiento 2>&1
```

Preguntas de análisis (discusión en clase)

Pregunta	Respuesta	Porque?
¿Cuántos printf de "ANTES" aparecen?		
¿Cuántos procesos hay después del fork?		
¿Por qué no aparece "Esta línea NUNCA se ejecuta"?		
¿El hijo conserva su PID después de `exec()`?		
¿Cuántos `write` syscalls hace `printf`?		

Variante para modificar

Cambio 1: ¿Qué pasa si quitamos `wait()`?

bash

```
# Editar: comentar la línea wait(NULL);
# Recompilar y ejecutar varias veces rápido
./nacimiento ; ./nacimiento ; ./nacimiento
```

Observar: Los mensajes del padre e hijo se mezclan aleatoriamente → **condición de carrera**

Cambio 2: ¿Qué pasa si usamos `execl` en lugar de `execlp` con ruta completa?

c

```
execl("/bin/date", "date", NULL);  // vs execlp("date", "date", NULL);
```

Observar: Ambos funcionan, `execlp` busca en PATH

LABORATORIO 2: "El Costo del Trap"

Objetivo

Medir cuánto cuesta cambiar de modo Usuario a Kernel.

costo_trap.c

bash

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/time.h>

long long ahora_us() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000LL + tv.tv_usec;
}

int main() {
    long long inicio, fin;
    int variable = 0;
    int fd = open("/dev/null", O_WRONLY);  // Dispositivo que descarta todo

    printf("Midiendo 100,000 operaciones...\n\n");

    // TEST 1: Operación en memoria (sin syscall)
    inicio = ahora_us();
    for (int i = 0; i < 100000; i++) {
        variable = i * 2;  // Solo CPU, sin llamar al kernel
    }
    fin = ahora_us();
    printf("1. Variable local (user space):  %lld µs total\n", fin - inicio);

    // TEST 2: Syscall write() al agujero negro
    char dato = 'X';
    inicio = ahora_us();
    for (int i = 0; i < 100000; i++) {
        write(fd, &dato, 1);  // Cada vuelta: User → Kernel → User
    }
    fin = ahora_us();
    printf("2. write() syscall:              %lld µs total\n", fin - inicio);

    // TEST 3: Syscall close/open (más pesado)
    inicio = ahora_us();
    for (int i = 0; i < 1000; i++) {  // Menos iteraciones, muy lento
        close(fd);
        fd = open("/dev/null", O_WRONLY);
    }
    fin = ahora_us();
    printf("3. open/close (1000 veces):      %lld µs total\n", fin - inicio);

    close(fd);
    return 0;
}
```

Compilamos:

```
gcc costo_trap.c -o costo_trap
```

Experimento: Ejecutar y analizar

bash

```
./costo_trap
```

Resultado típico

```
Midiendo 100,000 operaciones...

1. Variable local (user space):  89 µs total
2. write() syscall:              456,000 µs total
3. open/close (1000 veces):      2,100,000 µs total
```

Discutir

```
¿Por qué write() es ~5000x más lento?
```

Variante: Buffering vs No-buffering

bash

```
# Ver cómo printf optimiza
strace -e write -c ./costo_trap 2>&1 | tail -5

# Ahora forzar sin buffer
stdbuf -o0 ./costo_trap 2>&1 | strace -e write -c ./costo_trap
```

Parte	Significado
`strace`	Traza las syscalls
`-e write`	Solo muestra las llamadas `write()`
`-c`	Muestra estadísticas al final (count)
`2>&1`	Redirige errores a la salida normal
`| tail -5`	Muestra solo las últimas 5 líneas (el resumen)
`stdbuf`	Comando que modifica el buffering de E/S
`-o0`	**Output buffer = 0** (sin buffer, inmediato)

```
Comportamiento sin buffering

"H"  → write() → TRAP → Kernel → Pantalla
"o"  → write() → TRAP → Kernel → Pantalla
```

```
"l"  → write() → TRAP → Kernel → Pantalla
"a"  → write() → TRAP → Kernel → Pantalla
...  (11 syscalls para 11 caracteres)
```

Comparacion visual:

```
CON BUFFER (normal)                    SIN BUFFER (stdbuf -o0)
═══════════════════                    ═══════════════════════

printf("A");                            printf("A");
printf("B");                            → write("A")  [TRAP]
printf("C");                            printf("B");
...                                     → write("B")  [TRAP]
                                        printf("C");
[acumula en memoria]                    → write("C")  [TRAP]
      ↓
[al final del programa]
      ↓
write("ABC...")  [1 TRAP]               [1000 TRAPS para 1000 chars]
```

Situación	**Comando**	**Uso**
Logs en tiempo real	`stdbuf -o0`	Ver mensajes inmediatamente (debug)
Pipes interactivos	`stdbuf -o0`	Que el siguiente programa reciba datos ya
Medir syscalls reales	`stdbuf -o0`	Para `strace` educativo

LABORATORIO 3: "Todo es Archivo"

Objetivo

Demostrar que en Unix dispositivos, archivos y procesos se acceden igual.

Experimento A: Escribir en dispositivos

bash

```
# Crear archivo de prueba
echo "Hola" > /tmp/prueba.txt

# Ver qué dispositivos existen
ls -l /dev/tty /dev/null /dev/zero /dev/random /tmp/prueba.txt
```

Experimento B: El mismo programa, diferentes "archivos"

escritor.c

bash

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

int main(int argc, char *argv[]) {
    if (argc < 2) return 1;

    int fd = open(argv[1], O_WRONLY | O_CREAT, 0644);
    write(fd, "X", 1);
    close(fd);
    return 0;
}
```

Compilamos

```
gcc escritor.c -o escritor
```

Tabla de experimentos

Comando	¿Qué pasa?	Análisis
./escritor archivo.txt		
./escritor /dev/tty		
./escritor /dev/null		

Con mas de 1 carácter

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

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

int main(int argc, char *argv[]) {
    if (argc < 2) return 1;

    int fd = open(argv[1], O_WRONLY | O_CREAT, 0644);

    // String más largo (10 caracteres)
    const char *texto = "HOLAMUNDO\n";
    write(fd, texto, strlen(texto));  // ← strlen calcula automáticamente

    close(fd);
    return 0;
}
```

Realizar lo mismo

Pregunta clave

¿Por qué el mismo código (`open`, `write`, `close`) funciona con un archivo en disco, tu pantalla?

LABORATORIO 4: "La Terminal Cruda"

Objetivo

Ver cómo `read()` cambia de comportamiento según el modo del dispositivo.

Constante	Valor	Significado
`O_RDONLY`	0	**Solo lectura** (Read Only)
`O_WRONLY`	1	**Solo escritura** (Write Only)
`O_RDWR`	2	**Lectura y escritura** (Read + Write)

Setup

bash

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

int main() {
    int fd = open("/dev/tty", O_RDWR);
    char c;

    printf("=== MODO CANÓNICO (normal) ===\n");
    printf("Escribe algo y presiona ENTER:\n");
    fflush(stdout);  // Forzar que el mensaje aparezca ANTES del read

    // read() bloquea hasta ENTER, pero leemos solo 1 byte del resultado
    ssize_t n = read(fd, &c, 1);
    printf("Leído: '%c' (n=%zd)\n", c, n);

    // ⚠ LIMPIAR el resto del buffer (hasta el \n)
    char basura;
    while (read(fd, &basura, 1) > 0 && basura != '\n');

    printf("\n=== MODO NO CANÓNICO (crudo) ===\n");
    printf("Presiona CUALQUIER tecla (incluso sin ENTER):\n");
    fflush(stdout);

    struct termios t;
    tcgetattr(fd, &t);
    t.c_lflag &= ~(ICANON | ECHO);  // Sin modo canónico, sin eco
    t.c_cc[VMIN] = 1;   // Mínimo 1 byte para retornar
    t.c_cc[VTIME] = 0;  // Sin timeout (espera infinitamente)
    tcsetattr(fd, TCSANOW, &t);

    n = read(fd, &c, 1);  // Ahora SÍ espera una tecla nueva
    printf("\nLeído instantáneamente: '%c' (n=%zd)\n", c, n);

    // Restaurar terminal
```

```
    t.c_lflag |= (ICANON | ECHO);
    tcsetattr(fd, TCSANOW, &t);

    close(fd);
    return 0;
}
```

Compilamos

```
gcc modos_terminal.c -o modos_terminal
```

Experimento

bash

```
./modos_terminal
```

Observación con strace

bash

```
# En modo canónico: ¿cuántos read() ves?
strace -e read,write ./modos_terminal

# Comparar: en modo canónico, read() bloquea hasta \n
# En modo no canónico, read() retorna inmediatamente
```

Ahora modificando el numero de bytes:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>

int main() {
 int fd = open("/dev/tty", O_RDWR);

 // ========== MODO CANÓNICO ==========
 printf("=== MODO CANÓNICO ===\n");
 printf("Escribe algo y presiona ENTER:\n");
 fflush(stdout);

 char buffer[100];  // ← BUFFER GRANDE, no char c;

 ssize_t n = read(fd, buffer, sizeof(buffer)-1);  // ← Pide hasta 99 bytes
 buffer[n] = '\0';  // ← Cerrar string

 if (n > 0 && buffer[n-1] == '\n') {
 buffer[n-1] = '\0';  // ← Quitar \n para mostrar limpio
 }

 printf("Recibido (%zd bytes): '%s'\n", n, buffer);

    // ========== MODO NO CANÓNICO ==========
```

```
        printf("\n=== MODO NO CANÓNICO ===\n");
        printf("Presiona una tecla:\n");
        fflush(stdout);

        struct termios t;
        tcgetattr(fd, &t);
        t.c_lflag &= ~(ICANON | ECHO);
        t.c_cc[VMIN] = 1;
        t.c_cc[VTIME] = 0;
        tcsetattr(fd, TCSANOW, &t);

        char tecla;  // ← 1 byte está bien aquí porque queremos 1 tecla
        read(fd, &tecla, 1);  // ← Pedimos 1 byte, buffer es de 1 byte ✓

        printf("Tecla: '%c' (ASCII %d)\n", tecla, tecla);

        // Restaurar
        t.c_lflag |= (ICANON | ECHO);
        tcsetattr(fd, TCSANOW, &t);

        close(fd);
        return 0;
    }
```

Que pasa si modificamos el char buffer:

```
    char buffer[10];

    y

    char buffer[5];
```

Conexión con conceptos

Modo	**¿Cuándo retorna `read()`?**	**Uso típico**
Canónico	Cuando presionas ENTER	Shell, editores de línea
No canónico	Inmediatamente por tecla	Juegos, `vi`, `top`

FORMATO DE ENTREGA

No es código, es **reporte de observaciones**:

```
LABORATORIO X: [Título]

Hipótesis (antes de ejecutar):
_____________________________________

Comandos ejecutados:
_____________________________________

Observaciones (qué viste):
_____________________________________

Capturas/evidencia:
[pegar salida de terminal]

Conclusión (qué aprendiste sobre el SO):
_____________________________________
```