Práctica 1: Introducción a la Shell, Procesos e Hilos

Sistemas Operativos - Programación en C con POSIX

FECHA LÍMITE: 3-6 MARZO

Contenidos de la Práctica

Introducción a la Shell

Comandos básicos y filosofía Unix

Programación en C

Control de errores y espera inactiva

Procesos

Creación y gestión de procesos

Hilos

Programación multihilo con pthread

¿Qué es una Shell?

Una **shell** es una interfaz de usuario que permite usar los recursos del sistema operativo. Aunque los entornos gráficos pueden considerarse shells, el término se refiere normalmente a los Intérpretes de Líneas de Comandos (CLI).

Los sistemas Unix disponen de diversas shells con distintas capacidades y sintaxis. La más común es **Bourne-Again Shell (bash)**, desarrollada en 1989 por Brian Fox para el GNU Project.

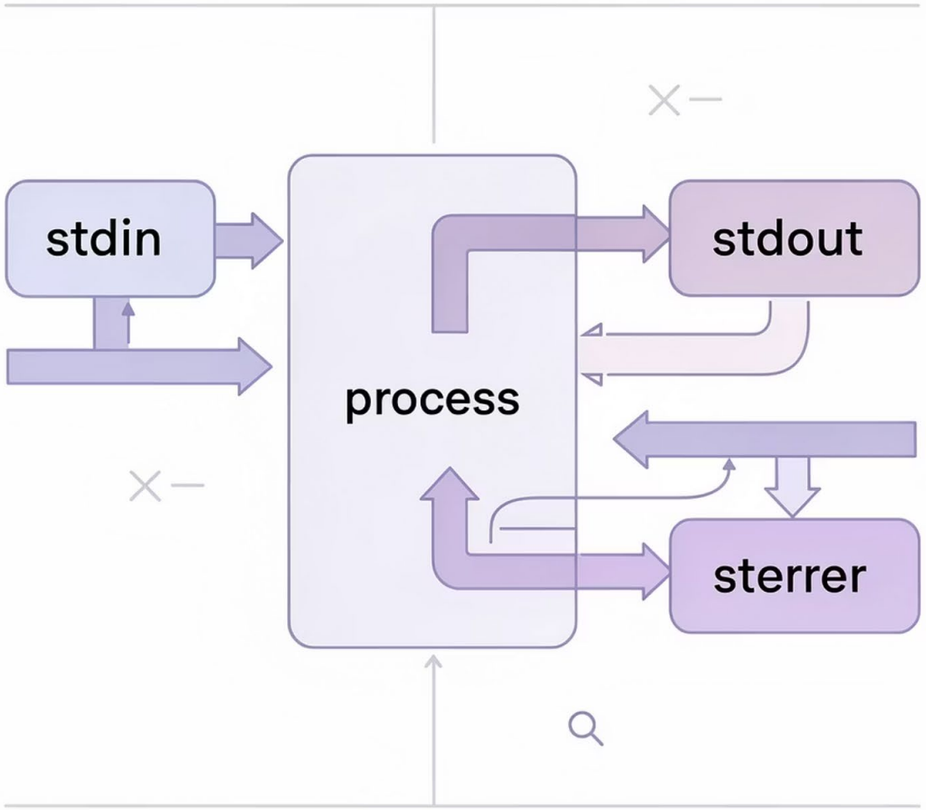
Familias de Shells Unix

Bourne Shell Compatible

La familia más extendida incluye: sh, bash, ash, dash, ksh. Heredan características de la Bourne Shell original de 1977.

C Shell Compatible

Sintaxis más parecida a C. Incluye: csh, tcsh. Menos común pero útil para programadores de C.



Canales de Comunicación Estándar

01

Entrada estándar (stdin)

Canal 0 - Recibe datos de entrada

02

Salida estándar (stdout)

Canal 1 - Escribe resultados

03

Salida de errores (stderr)

Canal 2 - Comunica mensajes de error

Filosofía Unix

Principio Fundamental

"Todo es un fichero"

Los programas deben leer de stdin y escribir a stdout en formato texto comprensible por otros programas.

- Cada programa hace una única cosa pero la hace correctamente
- Los programas se pueden encadenar mediante tuberías
- Las redirecciones permiten reutilizar programas de formas no previstas
- Sin salida ni errores, el programa no escribe nada

Operadores de Redirección

<

Redirige entrada desde fichero

>

Redirige salida a fichero (sobrescribe)

2>

Redirige errores a fichero

>>

Añade salida al final del fichero

|

Tubería: conecta salida con entrada

Comandos de Ayuda

man - Manual del Sistema

El comando más útil. Muestra el manual de cualquier comando, detallando objetivo, opciones y parámetros.

- **man man**: Manual del propio comando man
- **man -k <palabra>**: Busca palabra en todos los manuales
- **man <n_sec> <comando>**: Accede a sección específica

Ejercicio 1: Uso del Manual

Práctica con man

a) Buscar funciones de manejo de hilos que comienzan por "pthread"

b) Consultar en qué sección están las llamadas al sistema y buscar información sobre **write**

Comandos de Navegación

cd

Cambia el directorio actual

- cd ~ : directorio usuario
- cd / : directorio raíz
- cd .. : directorio padre
- cd - : último visitado

ls

Lista ficheros del directorio

- ls -l : info extendida
- ls -a : muestra ocultos
- ls <dir> : lista directorio

Comandos de Manejo de Texto

- cat

  Concatena y muestra ficheros

- head / tail

  Primeras/últimas líneas

- grep

  Búsqueda de patrones

- wc

  Cuenta letras, palabras, líneas

- sort

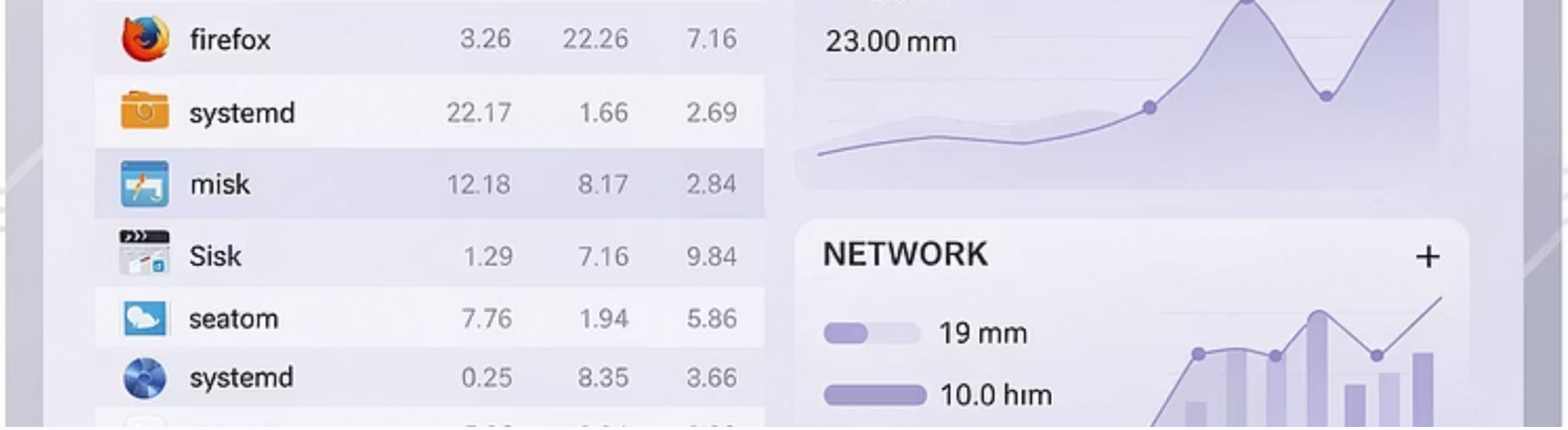
  Ordena líneas de texto

- uniq

  Filtra líneas repetidas

- more / less

  Visualización paginada



Comandos de Gestión de Procesos

top

Muestra dinámicamente información de procesos: CPU, memoria, etc.

ps

Información estática de procesos. Opciones: -A (todos), -u (usuario), -fl (detallado), -L (hilos)

pstree

Jerarquía de procesos en árbol. Con PID muestra subárbol desde ese proceso.

Ejercicio 2: Comandos y Redireccionamiento

Práctica de Pipelines

1. Buscar líneas con "molino" en don_quijote.txt y añadirlas a aventuras.txt
2. Contar número de ficheros en directorio actual
3. Contar líneas distintas al concatenar dos listas de compra, ignorando errores

Control de Errores en C

La mayoría de funciones C y POSIX pueden fallar. Es crucial comprobar el valor de retorno para detectar errores.

La variable global **errno** (tipo int) almacena el tipo de error producido. Los identificadores de error están declarados en **errno.h**.

Funciones de Error

- **strerror**: Obtiene mensaje de error apropiado
- **perror**: Imprime mensaje directamente

Importante: Copiar errno a otra variable si se desea tratar el error más tarde, ya que cualquier función puede alterarlo.

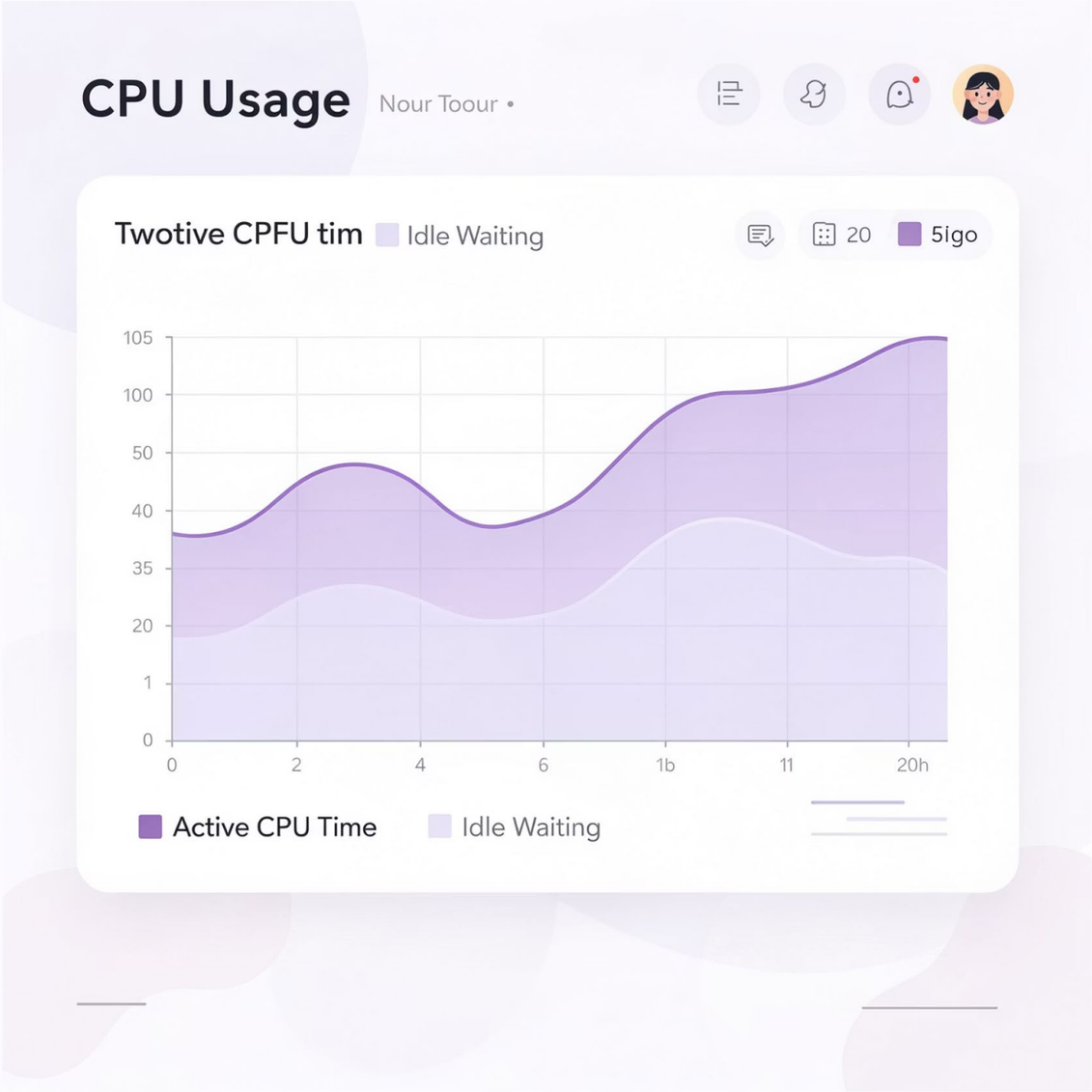
Ejercicio 3: Control de Errores

Práctica con fopen y perror

Escribir programa que abra fichero con fopen. En caso de error, mostrar mensaje con perror.

1. ¿Qué mensaje al abrir fichero inexistente? ¿Valor de errno?
2. ¿Qué mensaje al abrir /etc/shadow? ¿Valor de errno?
3. ¿Cómo garantizar que perror muestre el error correcto de fopen?

Espera Inactiva vs Activa



Funciones de Espera

sleep y **nanosleep** permiten esperar sin consumir CPU. El planificador ejecuta otros hilos y despierta al hilo cuando pasa el tiempo indicado.

El planificador no despierta inmediatamente - puede pasar algo más de tiempo (orden de milisegundos).

Ejercicio 4: Espera Activa e Inactiva

- Comparación de Métodos

  a) Escribir programa con espera de 10 segundos usando **clock** en bucle.
  Ejecutar **top** en otra terminal. ¿Qué se observa?

  b) Reescribir usando **sleep** y volver a ejecutar top. ¿Ha cambiado algo?

¿Qué es un Proceso?

Un **proceso** es un programa en ejecución. Todo proceso Unix tiene:

- Identificador de proceso (PID)
- Proceso padre (puede tener procesos hijo)
- Propietario (usuario que lo lanzó)
- El proceso **init** (PID=1) es el padre de todos los procesos

Visualización de Procesos con ps

```
$ ps -ef
UID   PID  PPID  C STIME TTY      TIME CMD
root     1     0  0 11:48 ?        00:00:00 /sbin/init
practica 1712     1 18 12:08 ?        00:00:00 gnome-terminal
practica 1713 1712  0 12:08 ?        00:00:00 gnome-pty-helper
practica 1714 1712 20 12:08 pts/0    00:00:00 bash
practica 1731 1714  0 12:08 pts/0    00:00:00 ps -ef
```

Columnas: UID (usuario), PID (identificador proceso), PPID (PID del padre), CMD (comando ejecutándose)

Creación de Procesos: fork()

La llamada **fork** crea un proceso hijo, copia exacta del padre excepto PID y PPID. Ambos procesos ejecutan concurrentemente.

Valor de retorno de fork:

- En el padre: PID del hijo
- En el hijo: 0
- Error: -1

Funciones Útiles

- **getpid()**: Obtiene PID propio
- **getppid()**: Obtiene PID del padre

Nota: Los PID se almacenan en variables tipo pid_t. Para imprimirlos, convertir a intmax_t y usar %jd.

Espera de Procesos Hijo

Todo proceso padre debe recoger el resultado de sus hijos con **wait** o **waitpid**.

WIFEXITED

Determina si terminó por sí mismo

WIFSIGNALED

Determina si terminó por señal

WEXITSTATUS

Obtiene valor de retorno

WTERMSIG

Obtiene señal que causó muerte

Procesos Zombis y Huérfanos

Proceso Zombi

Proceso hijo que ha terminado pero su padre no ha recogido el resultado. Ha liberado recursos pero mantiene entrada en tabla de procesos.

Evitar dejando procesos zombis.

Proceso Huérfano

Proceso hijo cuyo padre terminó sin esperarlo. Históricamente init (PID=1) los recogía, ahora puede ser otro ancestro.

Todo padre debe esperar a sus hijos.

Ejercicio 5: Creación de Procesos

Análisis de Código con fork

Dado código que crea NUM_PROC=3 procesos:

1. Analizar texto impreso. ¿Se puede saber orden a priori? ¿Por qué?
2. Cambiar para que hijo imprima su PID y el del padre
3. ¿A cuál de dos árboles de procesos corresponde? ¿Modificaciones para el otro?
4. ¿Por qué deja procesos huérfanos?
5. Cambios mínimos para no dejar huérfanos

Espacio de Memoria en Procesos

Procesos padre e hijo **no comparten memoria**: son completamente independientes.

Aunque el hijo hereda una copia de la memoria del padre, las modificaciones en uno no afectan al otro.

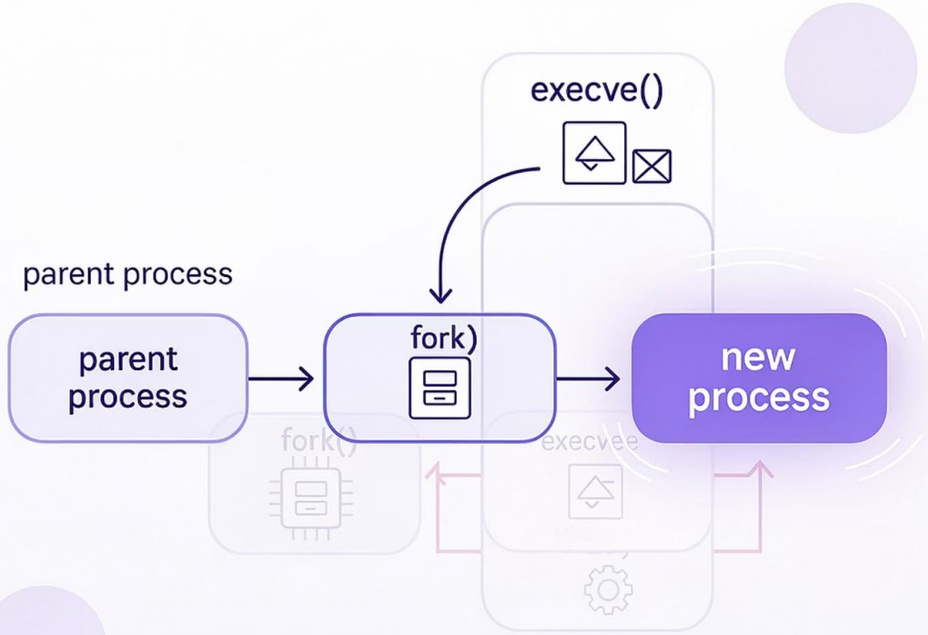
Ejercicio 6: Espacio de Memoria

Análisis de Memoria Compartida

Código que reserva memoria en padre e inicializa en hijo con strcpy:

1. ¿Qué ocurre al ejecutar? ¿Es correcto? ¿Por qué?
2. El código tiene fuga de memoria. ¿Dónde liberar: padre, hijo o ambos? ¿Por qué?

Con esto se puede realizar parte a) del ejercicio de programación



Familia de Funciones exec

Las llamadas **exec** reemplazan el código del proceso actual por otro programa. Permiten que un hijo ejecute código diferente al padre.

Importante: No hay retorno tras exec exitoso. Solo se ejecuta tratamiento de errores si falla.

Variantes de exec

Letra 'v'

execv, execvp, execve: Argumentos en array terminado en NULL

Letra 'l'

execl, execlp, execle: Argumentos como parámetros separados, terminando en (char *)NULL

Letra 'e'

execve, execle: Reciben nuevo conjunto de variables de entorno

Letra 'p'

execvp, execlp: Buscan ejecutable en directorios de PATH

Ejercicio 7: Ejecución de Programas

Práctica con exec

Código que ejecuta comando ls usando execvp:

a) ¿Qué sucede si se sustituye "ls" por "mi-ls" en argv? ¿Por qué?

b) ¿Qué modificaciones para usar execl en lugar de execvp?

Nota: Obtener ruta completa de ls con: which ls

Introducción a Hilos

Los **hilos** son unidades de trabajo ejecutadas por el sistema operativo. Permiten ejecución simultánea de tareas dentro del mismo programa.

El SO mantiene todas las CPU ocupadas ejecutando hilos y alterna su ejecución si hay más hilos que CPU.

Características de los Hilos

Memoria Compartida

Todos los hilos del programa acceden a la misma memoria y recursos.

Comunicación Rápida

Comunicación directa entre hilos, pero requiere mayor cuidado del programador.

Pila Propia

Cada hilo tiene su propia pila local de tamaño fijo para variables automáticas.

Biblioteca pthread

Para programas multihilo en C se usa la biblioteca **pthread** (POSIX threads).

Requisitos:

- Incluir cabecera: #include <pthread.h>
- Compilar con: gcc -lpthread (o -pthread)

Hilo Principal

Al inicio, el programa tiene un único hilo que ejecuta main. Cuando main retorna o se llama a exit, todos los hilos finalizan.

Funciones Principales de pthread

1. **pthread_create**
   Crea nuevo hilo para ejecutar función específica
2. **pthread_self**
   Obtiene identificador del hilo actual
3. **pthread_join**
   Espera a que hilo termine y recupera valor retorno
4. **pthread_exit**
   Finaliza el hilo actual
5. **pthread_detach**
   Desliga hilo para que no pueda ser esperado

Gestión de Errores en pthread

Diferencia Importante

Las funciones pthread retornan el error directamente, no usan errno.

En programas multihilo, errno no es variable global sino **thread-local**: cada hilo tiene su propio errno.

Esto evita condiciones de carrera al acceder a información de errores.

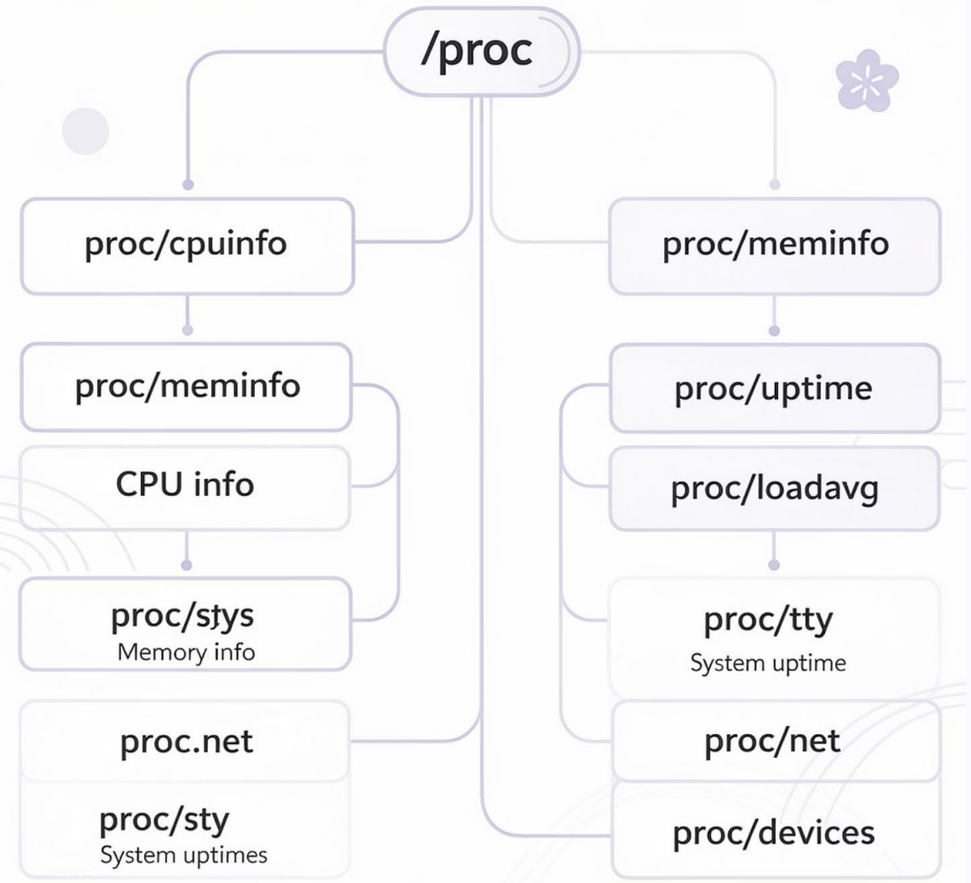
Ejercicio 8: Finalización de Hilos

Análisis de Código Multihilo

Código con función slow_printf que crea dos hilos:

1. ¿Qué hubiera pasado sin pthread_join? Probar eliminando las llamadas.
2. Con código modificado, ¿qué ocurre si se reemplaza exit por pthread_exit?
3. Sin pthread_join, ¿qué código añadir para gestión correcta de recursos?

Con esto se puede realizar parte b) del ejercicio de programación



El Directorio /proc

Directorio especial que no contiene ficheros reales. Simula información del sistema operativo como ficheros, siguiendo la filosofía Unix "todo es un fichero".

Contiene carpeta con PID de cada proceso del sistema. El fichero **self** es enlace a la carpeta del proceso que accede a él.

Ejercicio 9: Directorio /proc

Exploración de Información de Procesos

Buscar en /proc para algún proceso:

1. Nombre del ejecutable
2. Directorio actual del proceso
3. Línea de comandos usada para lanzarlo
4. Valor de variable de entorno LANG
5. Lista de hilos del proceso

Nota: Usar tr '\0' '\n' para convertir delimitadores

Funciones de Ficheros en C

Alto Nivel (FILE)

- fopen / fclose
- fprintf / fgets
- fwrite / fread

Objetos FILE con buffer privado

Bajo Nivel (Descriptores)

- open / close
- write / read

Descriptores de fichero (int)

Función open y sus Flags

La función **open** retorna descriptor de fichero (int) en lugar de FILE.

Requiere flags para especificar modo de apertura:

Obligatorios

O_RDONLY,

O_WRONLY, O_RDWR

O_APPEND

Escrituras al final

O_CLOEXEC

Cierra en exec

O_CREAT

Crea si no existe

O_EXCL

Error si existe

O_TRUNC

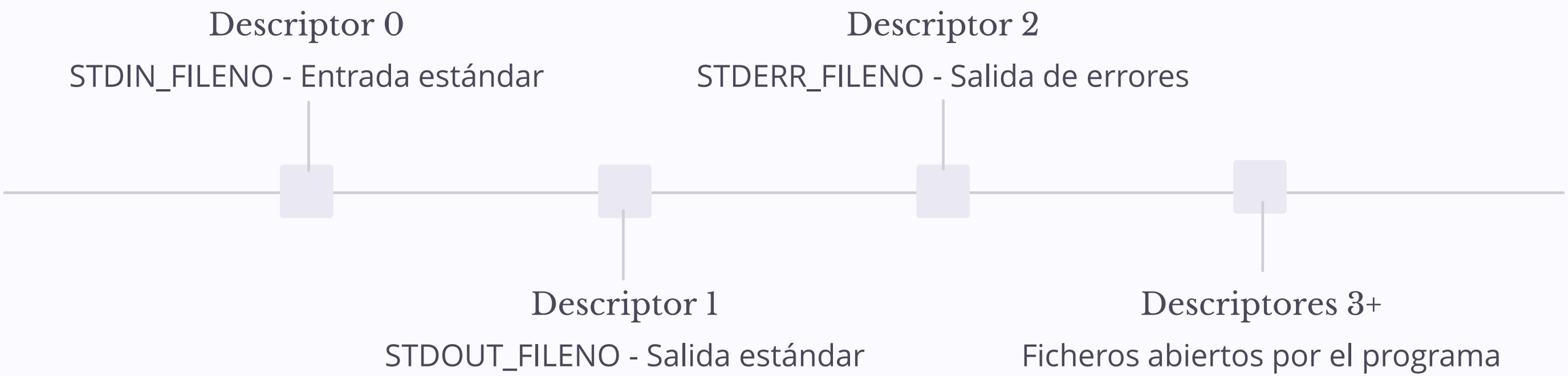
Trunca a tamaño 0

Lectura y Escritura con Descriptores

Funciones **read** y **write** permiten lectura/escritura de número fijo de bytes.

Importante: Pueden leer/escribir menos bytes que los especificados. Siempre comprobar valor retornado y reintentar si es necesario.

Tabla de Descriptores de Fichero



Descripción de Fichero Abierto

Varios descriptores pueden referirse a la misma **descripción de fichero abierto**, incluso de procesos diferentes.

Esto ocurre tras fork: descriptores del hijo se refieren a las mismas descripciones que el padre.

Funciones de Gestión

- **close**: Cierra descriptor
- **unlink**: Borra ruta de acceso

El fichero se borra de disco cuando se borran todas las rutas y todos los procesos lo cierran.

Ejercicio 10: Visualización de Descriptores

Análisis de Tabla de Descriptores

Código que abre/cierra ficheros en varios momentos (Stop 1-7).
Inspeccionar /proc/<PID>/fd en cada parada:

1. Stop 1: ¿Qué descriptores abiertos? ¿Tipo de fichero?
2. Stop 2-3: ¿Cambios en tabla?
3. Stop 4: ¿Se borró FILE1? ¿Se puede acceder vía /proc? ¿Recuperar datos?
4. Stop 5-7: ¿Cambios? ¿Qué deducir sobre numeración de descriptores?

Conversión entre FILE y Descriptores

fdopen

Crea objeto FILE asociado a descriptor de fichero arbitrario

fileno

Obtiene descriptor de fichero asociado a objeto FILE

Nota: Algunos FILE (fmemopen, open_memstream) no tienen descriptor asociado.

Diferencias: FILE vs Descriptores

Objetos FILE

- Tienen buffer privado en memoria del proceso
- Minimizan llamadas al sistema
- Escrituras se acumulan hasta llenar buffer
- Garantizan acceso no simultáneo al buffer

Descriptores

- Sin buffer en proceso
- Escritura más inmediata
- SO mantiene caché común a todos los procesos
- Mejor para sincronización precisa

Ejercicio 11: Problemas con el Buffer

Análisis de Buffer en FILE

Código con printf sin '\n' y fork:

1. ¿Cuántas veces se escribe "I am your father"? ¿Por qué?
2. Corregir añadiendo '\n'. ¿Sigue ocurriendo lo mismo? ¿Por qué?
3. Ejecutar redirigiendo salida a fichero. ¿Qué ocurre? ¿Por qué?
4. ¿Cómo corregir definitivamente sin dejar de usar printf?

Tuberías (Pipes)

Mecanismo para comunicar procesos con relación parental. Crea canal de comunicación **unidireccional**.

Consiste en dos descriptores: fd[0] para leer, fd[1] para escribir. Se usan con read y write como ficheros normales.

Creación de Tuberías

Función **pipe** crea tubería simple. Recibe array de dos enteros y crea dos descriptores nuevos.

Importante: La tubería debe crearse **antes** de fork para que padre e hijo tengan acceso.

Comunicación Unidireccional

Solo uno escribe, el otro solo lee. Para comunicación bidireccional se necesitan dos tuberías.

Comportamiento de las Tuberías

Escrituras Atómicas

Escrituras < PIPE_BUF son atómicas: no se mezclan ni son cortas

Capacidad Limitada

Si se llena, el escritor espera automáticamente

Lectura Bloqueante

Si está vacía, el lector espera a que haya contenido

EOF Automático

Leer de tubería vacía sin escritores retorna 0 (EOF)

SIGPIPE

Escribir sin lectores envía señal SIGPIPE (finaliza proceso)

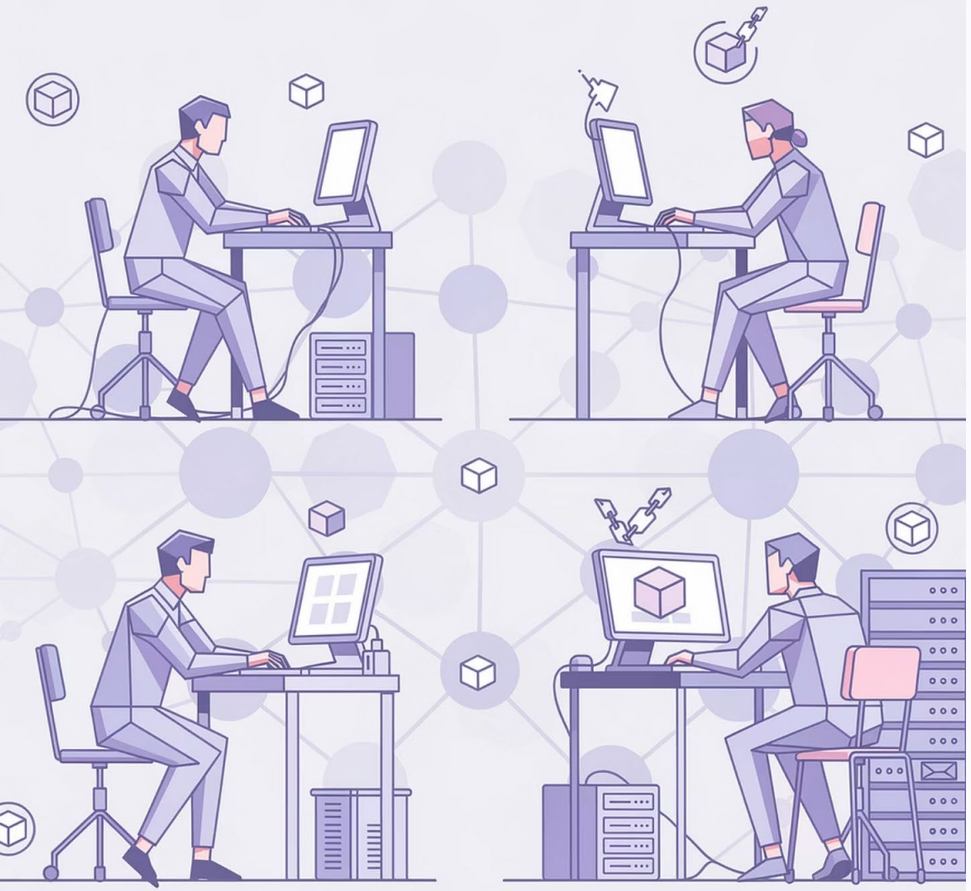
Ejercicio 12: Ejemplo de Tuberías

- Práctica con pipe

  Código donde hijo escribe mensaje en tubería y padre lo lee:

  1. Ejecutar el código. ¿Qué se imprime por pantalla?
  2. ¿Qué ocurre si el padre no cierra el extremo de escritura? ¿Por qué?

  Con esto se puede realizar parte c) del ejercicio de programación



Ejercicio de Codificación: Miner Rush

Proyecto incremental que implementa red de mineros de bloques inspirada en Blockchain. Los mineros resuelven pruebas de esfuerzo (POW) de forma concurrente.

Puntuación total: 10 puntos

Prueba de Esfuerzo (POW)

Consiste en resolver reto matemático: hallar operando que permite obtener resultado determinado tras aplicar función hash no invertible.

Dado objetivo *t* y función hash *f*, encontrar solución *s* tal que *f(s)* = *t*

Método de Resolución

Única forma: **fuerza bruta**. Probar todos los valores posibles entre 0 y POW_LIMIT - 1.

Ficheros proporcionados: pow.c y pow.h

Parte a) Sistema Multiproceso (2 puntos)

Requisitos

Ejecutar con tres parámetros:

```
./miner <TARGET_INI> <ROUNDS> <N_THREADS>
```

- Proceso principal **Minero** crea proceso **Registrador** con fork
- Minero determina cuándo completó última ronda
- Minero envía orden de finalización a Registrador por tubería
- Ambos indican código de salida del hijo por pantalla

Parte b) Minero Multihilo (4 puntos)

Proceso Minero resuelve POW usando múltiples hilos en paralelo:

01

División del Trabajo

Divide espacio de búsqueda entre hilos especificados

02

Creación y Espera

Crea hilos y espera a que terminen

03

Finalización

Cuando un hilo encuentra solución, todos terminan

04

Siguiente Ronda

Solución obtenida se fija como siguiente objetivo

Parte c) Registrador del Sistema (3 puntos)

Proceso **Registrador** registra en fichero las soluciones encontradas por Minero.

Minero envía mensajes por tubería indicando objetivo y solución obtenida.

Finalización

Cuando detecta cierre de tubería, termina con:

- EXIT_FAILURE si hay error
- EXIT_SUCCESS si todo correcto

Formato del Fichero Registrador

```
Id: [NUMERO RONDA]
Winner: [ID PROCESO PADRE]
Target: [TARGET RONDA]
Solution: [SOLUCION ENCONTRADA] [(validated)/(rejected)]
Votes: [NUMERO RONDA]/[NUMERO RONDA]
Wallets: [ID PROCESO PADRE]:[NUMERO RONDA]
```

Nombre del fichero depende del PID del proceso padre. De momento asumir (validated), pero realizar pruebas forzando (rejected) aleatoriamente.

Ejemplos de Bloques

Solución Validada

```
Id: 19
Winner: 6837
Target: 27818400
Solution: 53980520 (validated)
Votes: 19/19
Wallets: 6837:19
```

Solución Rechazada

```
Id: 19
Winner: 7072
Target: 27818400
Solution: 00000011 (rejected)
Votes: 19/19
Wallets: 7042:19
```

Parte d) Pruebas y Razonamiento (1 punto)

Análisis de Rendimiento

Realizar pruebas con diferente número de rondas y hilos, tomando tiempos de ejecución.

Pregunta: ¿Qué conclusiones se pueden tomar respecto al número óptimo de hilos?

Ejemplos de Ejecución

```
$ ./mrush 0 5 3
Solution accepted: 00000000 --> 38722988
Solution accepted: 38722988 --> 82781454
Solution accepted: 82781454 --> 59403743
Solution accepted: 59403743 --> 44638907
Solution accepted: 44638907 --> 98780967
Logger exited with status 0
Miner exited with status 0
```

Con solución incorrecta forzada en tercera ronda:

```
Solution rejected: 82781454 !--> 00000011
```

Aspectos Importantes a Considerar

Documentación

Entrega en formato PDF según normativa

Control de Errores

Gestión adecuada de todos los errores

Tuberías

Cierre correcto de extremos no usados

Espera de Procesos

Todo padre espera a sus hijos

Gestión de Hilos

Creación y finalización adecuada

Resumen y Entrega

Fecha Límite

Del 3 al 6 de marzo, antes del comienzo de la clase de prácticas

Contenido de la Entrega

- Código fuente completo
- Documentación en PDF
- Análisis de pruebas y conclusiones

Esta práctica sienta las bases para el proyecto incremental Miner Rush que se completará en prácticas posteriores.