Sistemas operativos.

Comandos linux - manejo de ficheros en C

- ls - Muestra el contenido de directorios del sistema (DIR)
  - lo podemos combinar con (”-a”) muestra tambien archivos y directorios ocultos
  - -R (recursivamente muestra los subdirectorios)
- mkdir - para hacer un directorio (”mkdir <nombre_directorio>”)
- touch - para crear un archivo (”touch archivo.c”)
- ls -l - para ver los permisos de un archivo (”ls -l archivo.c”)
- rm - elimina archivos o directorios, (”rm <archivo>”)
- chmod - cambia los permisos de acceso al archivo del directorio que especifiquemos usando el sistema octal, (”chmod <modo> <archivo>”) tres cifras octales

Tratamiento de ficheros.

- cat - concatena y MUESTRA el contenido de nuestros archivos, la salida (por defecto) será por pantalla (cat > nuevo.txt) te da para crear un nuevo archivo y escribir lo que quieras
- “>” - redirige lo de la izquierda a la derecha
- passwd - es utilizado para cambiar la clave de acesso al sistema (debemos apuntarla dos veces)
- wc - cuenta el numero de lineas, palabras y caracteres de un fichero.
- grep-busca las lineas que siguen un patron

Compilacion de nuestro archivo.c

- usaremos el comando (”gcc -o <nombre_destino> <fichero.c>”) (DEBERIAMOS USAR MAKEFILE).

Operaciones de e/s, manejo de ficheros.

- C tiene el tipo de dato **FILE**, que se usa como apuntador al informacion del fichero, debemos:
  - crear el apuntador de tipo FILE *
  - abrir el archivo utilizando la funcion “fopen” y asignarle el resultado de la llamada a nuestro apuntador
  - hacer las diversa operaciones
  - cerrar el archivo con “fclose”

- Apertura, creacion y cierre de ficheros:
  - la funcion “fopen” sirve para abrir y crear ficheros en un sistema de ficheros →
    - FILE *fopen(cons char *”Nombre_fichero”, const char *”mode”);
  - “mode” son los “Permisos_con_los_que_se_desea_abrir_el_fichero”
    - “r” abrir en forma de lerctura
    - “w” sobre escribir el archivo si existe y si no simplemente escribe
    - “r+” lectura y escritura, el fichero debe existir
    - “w+” lectura y escritura, se crea si no existe o lo sobre escribe si existe
  - valor devuelto es de tipo “File *”
- La funcion “fclose” sirve para cerrar ficheros en el sistema de ficheros, su linea de comando es:
  - int fclose(FILE *fp); el valor de retorno 0 indica que el fichero se ha cerrado correctamente, si hubo un error el resultado que retorna es eof

(”end of file”)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
   FILE *fp;
   fp = fopen ( "nombre_del_archivo", "modo_de_uso");
   fclose (fp);

   return 0;
}
```

- Funcion FEOF (fin del fichero):
  - int feof(FILE *fichero)
  - devuelve 0 si es falso(no se acabo el fichero), y devuelve un valor distinto de 0 en caso de verdadero
- Funcion de vuelta al origen.
  - void rewind(FILE *fichero)
  - permite situar el cursor de lectura/escritura al principio

Formas de lectura de un fichero

- char fgetc(FILE *archivo)
- char *fgets(char *buffer, int tamaño, FILE *archivo)
- size_t fread(void *puntero, tamaño, cantidad, FILE *archivo)
- int fscanf(FILE *fichero, const char formato, argumento, ...)
  - fgetc- lee un caracter, **SI** lo encuentra devuelve el caracter leido, sino devuelve EOF.

- fgets - lee cadena de caracteres y lo leera hasta tamaño de caracteres o hasta que lea un retorno de linea, el retorno de linea tambien es leido, fgets(nombre de una variable char [100], tamaño 100, FILE *archivo)
- fscanf (FILE*archivo, “%s”, char buffer[100])

Formas de escritura en un fichero

- int fputc(int caracter, FILE *archivo)
- int fputs(const char *buffer, FILE *archivo)
- int fprintf(FILE *archivo, const char *formato, argumento, ...)

Otras formas de hacer lo mismo

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *camino, int flags);
int open(const char *camino, int flags, mode_t modo);
```

- La llamada al sistema open() convierte una ruta en un descriptor de fichero (es decir en un numero) como cuando vas a la biblioteca y pides un libro, te daran un numero de ticket que no es el libro como tal pero es con ese numero que puedes pedir el libro.
- SI OPEN FALLA devuelve (-1)
- el parametro flags indica como debemos abrir el fichero, podemos utilizar O_RDONLY, O_WRONLY, O_RDWR, **Tenemos que combinarlo con “|” , O_CREAT** si el fichero no existe.
- close(), sirve para cerrar el fichero usando el descriptor como argumento, int close(int descriptor);

- read() intenta leer n bytes del fichero cuyo descriptor es fd, y lo guarda en la zona de memoria buf, si n bytes es 0 entonces read() devuelve 0, pero si es mayor que n bytes entonces devuelve un resultado indefinido.
    - ssize_t read(int fd, void *buf, size_t nbytes)
- write escribe a un descriptor de fichero fd hasta numbytes desde el bufer que comienza en buf
    - ssize_t write(int fd, const void *buf, size_t num)
- lseek es la funcion que utilizamos en O_RDWR para despues de leer volver al inicio y sobreescribir
    - lseek(int fd, -bytesleidos,

PRACTICA 2- PROCESOS E HILOS

Introduccion: Cada proceso tiene un identificador denominado pid, al que se le solicita la creacion de un nuevo proceso se le llama proceso padre, cuando se termine la ejecucion debe seguir un orden, primero morir el hijo y luego el padre.

Identificadores de procesos: (getpid() y getppid())

```
#include <sys/types.h>
#include <unistd.h>

pid_t pid, ppid; /* Declaración de variables*/
uid_t uid;
pid = getpid(); /* Asignación de los valores devueltos por las
 llamadas al sistema*/
pid = getppid();
uid = getuid();
```

Creacion de procesos: {fork()}→crea un proceso hijo→0

```
pid = fork();

if (pid == -1) {
printf(“Error en la creación del proceso\n”);
exit (-1);
}
if (pid == 0) { /* Proceso hijo */

printf("Este es el proceso hijo %d, cuyo padre es %ld\n", i,
(long)getppid());
exit(999);
} else { /* Proceso padre */
printf("Este es el proceso padre con ID %ld\n",
(long)getpid());
}
```

Llamadas al sistema de wait, waitpid y exit:

- wait: esta llamada al sistema permite que un proceso espere a que termine la ejecucion de su hijo.
- waitpid: esta espera puede ser por un proceso en concreto o por uno cualquiera de sus hijos wait

Llamada exec:

Pueden ejecutar comandos del sistema o comandos del usuario, cuando un proceso hijo ejecuta exec muere instantaneamente.

- Lo que tenemos que pasarle como entrada es una cadena que finalice con NULL
- variaciones del comando exec:

int execl//pasando los argumentos como una lista de parámetros.
int execv//como un arreglo de cadenas
int execle//como una lista y permitiendo especificar el entorno.
int execve//como arreglos de cadenas.
int execlp//como una lista y buscando el ejecutable en el PATH.
int execvp//como un arreglo y buscando el ejecutable en el PATH.

Terminacion de procesos:

si los procesos no los termina el padre, quedan los denominados procesos zombies y estos son acabados por el proceso init.

PRACTICA3-COMUNICACION ENTRE PROCESOS: PIPES/TUBERIAS

Introduccion: comunicacion entre procesos a traves de los pipes o tuberias, efectos de comunicacion bloqueantes/no bloqueantes, las pipes son tuberias sin nombre, diferentes a las FIFO.

pipe: la llamada al sistema pipe crea un par de descriptores de ficheros que apuntan a una tuberia donde uno se refiere a la lectura y el otro a la escritura[0, 1]-**SU LIMITACION** es que solo pueden comunicarse procesos relacionados.

Como usar la Tuberia o PIPE:

- primero creamos la tuberia y luego el proceso hijo para que estos puedan usar la tuberia heredando los descriptores creados con la llamada al sistema fork()
- Para evitar incosistencias, cada proceso debe cerrar la via de la tuberia que no va a utilizar como la escritura/lectura, es decir cerrar el descriptor de fichero relacionado con la operacion lectura o escritura.

```
#include <stdio.h>
int tubería[2], pid;
...
pipe(tuberia);
pid = fork();
if (pid == 0) {
 char *cadena;
 cadena = “Hola mundo”;
 close(tubería[0]);
 write(tubería[1], cadena, strlen(cadena)+1);
 close(tuberia[1]);
 exit(1);
} else {
 char *cadena;
 int bytesLeidos;
 cadena = (char *) malloc(100 * strlen(char));
 close(tubería[1]);
 bytesLeidos = read(tubería[0], cadena, 100);
 printf(“Mensaje leído: %s\n”, cadena);
 close(tuberia[0]);
 exit(1);
}
```

FIFO tuberias con nombre: permite la comunicacion de procesos que no tengan relacion(son persistentes)

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *pathname, mode_t modo);

//forma de usar el mkfifo:

int resultado = mkfifo("nombreFIFO", 0666)
if(resultado == -1){
    perror("Error al crear el FIFO");
}
```

```
//para eliminarlo:

unlink("nombreFIFO");

//para abrilo:

char *nombreFIFO = argv[posicion en la que este];

int fdFIFO = open(nombreFIFO, O_WRONLY); //para abrirlo en escritura

int fdFIFO = open(nombreFIFO, O_RDONLY); //para abrirlo en lectura

if(fdFIFO == -1){
   perror("Error al abrir el FIFO en escritura/lectura");
   exit(EXIT_FAILURE);
}

//muy importante con los ficheros y los descriptores es cerrarlos

//esto puede ser util:

FILE *fFifo = fdopen(fdFIFO, "r");
if (!fFifo){perror, close, exit};
```

tenemos una comunicacion bloqueante:

- • open(“fifo_ejemplo”, O_WRONLY) o open(“fifo_ejemplo”,O_RD ONLY)

tambien tenemos comunicacion no bloqueante:

- Usando el O_NONBLOCK si no hay nadie del otro lado el programa avanza

Redireccionamiento de una PIPE a la E/S:

el descriptor de fichero 0 esta destinado para stdin, el 1 para el stdout y el 2 para el stderr, el redireccionamiento se utiliza para conectar procesos entre la

entrada y la salida, es decir cuando queremos que con la entrada de un programa (proceso) empiece otro programa.

¿Como hacemos esto? usamos dup o dup2, permiten que un programa envie su salida a reciba su entrada de un programa diferente.

PRACTICA4- COMUNICACION ENTRE PROCESOS : SEÑALES

Introduccion: comunicacion de procesos basadas en señales, son como una "notificacion" que un programa le envia a otro para indicarle que haga algo o que pasó algo.Son asincronas lo que quiere decir que pueden llegar en cualquier momento e interrumpir el programa.

Ejemplo de señales: en la terminal al usar el comando Ctrl+c para acabar un proceso en la terminal, o Ctrl+z para cancelar el prrograma, etc.

Envio de señales: los procesos pueden enviar señales tanto a ellos mismo como a otros procesos, con el comando kill(), ejemplo: kill(pid, SIGUSR1), la llamada devuelve 0 si se ejecuta correctamente.

- si el argumento pid es positivo, entonces la señal sig se enviara al proceso pid
- si el argumento pid es = 0, entonces el sistema envia una señal a cada uno de los procesos que forman parte del grupo de procesos del proceso actual
- si el argumento es -1, entonces se envia la señal a todos los procesos menos al init.
- NO EXISTE UNA COMUNICACION DE SEÑAL CON EL PROCESO INIT PARA NO ACABAR CON EL SISTEMA ACCIDENTALMENTE

Tipos de señales comunes:

SIGINT →termina un proceso con una interrupcion de Ctrl + C;

SIGKILL→termina el proceso inmediatamente;

Como manejar las señales:

- funcion signal(SIGUSR1, manejador),SIGUSR1 va hacer lo que asignemos en la funcion manejador, cual el proceso reciba SIGUSR1 va a ejecutar manejador
- tenemos una forma mas avanzada de tratar con una señal con sigaction(SIGUSR1, &sa, NULL), el cual tiene un tipo estructura definida como:

```
struct sigaction {
    void (*sa_handler)(int);  // Manejador de señal (puntero a función)
    sigset_t sa_mask;         // Máscara de señales que se bloquearán
    int sa_flags;             // Modificadores para el comportamiento
};

struct sigaction sa;
sa.sa_handler = manejador_sigint;  // Asigna el manejador de señales
sigemptyset(&sa.sa_mask);          // Limpia la máscara de señales
sa.sa_flags = 0;
```

```
//importante:

volatile sig_atomic_t terminar = 0;

static volatile sig_atomic_t terminar = 0; //solo para usar dentro de un archi
vo.

void manejador_sigint(int sig){
    terminar = 1;
}

//para usarlo:
```

```
SIGNAL(SIGINT, manejador_sigint);

//para matar proceso:

kill(nombre_del_pid,SIGINT);

//para quedarnos bloqueados hasta que la señal llegue:

while(!terminar){
   pause();
}

//por que usar SIGINT o SIGUSR1 ?:
//SIGINT se utiliza para finalizar un proceso, SIGUSR1 se utiliza para notific
ar de un evento.
```

PRACTICA5-SHELL-DONE

PRACTICA6-SEMAFOROS Y MEMORIA COMPARTIDA

¿que son los semaforos y que tiene que ver con la memoria compartida?

los semaforos protegen zonas criticas, estas zonas criticas son las zonas de memoria compartida, como lo protegen ? permitiendo que un proceso a la vez acceda a los recursos.

¿como funcionan los semaforos?

- inicializamos con un valor inicial (1 es para indicar disponibilidad)
- Tenemos 2 operaciones :
  - P() o wait() decrementa el valor del semaforo, si es menor a 0 el proceso se bloquea
  - V() o signal() aumenta el valor desbloquando a otros procesos que estaban esperando

Como manejar los semaforos:

- con semget() accedemos o creamos un semaforo
- con semcttl() configuramos valores iniciales
- con semop() aumentamos o disminuimos.

Creando e inicializando un semaforo:

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    key_t clave;
    int id_semaforo;

    // Generar una clave única
    clave = ftok("/bin/ls", 'S');


    // Crear un conjunto de semáforos
    id_semaforo = semget(clave, 1, 0666 | IPC_CREAT);

    if(id_semaforo == -1){
        perror("Error al crear o acceder al semaforo");
        exit(EXITTT_FAILURE);
    }


    // Inicializar el semáforo a 1
    union semun {
        int val;
    } arg;
    arg.val = 1; // Valor inicial
    if (semctl(id_semaforo, 0, SETVAL, arg) == -1) {
        perror("Error inicializando semáforo");
```

```
        exit(EXIT_FAILURE);
    }

    //eliminar un semaforo:

    semctl(id_semAforo, 0, IPC_RMID);

}
```

operaciones con semaforos:

```
struct sembuf accion;

// Operación wait (P)
accion.sem_num = 0; // Índice del semáforo en el conjunto
accion.sem_op = -1; // Decremento
accion.sem_flg = 0; // Operación bloqueante
if (semop(id_semaforo, &accion, 1) == -1) {
    perror("Error en operación wait");
    exit(EXIT_FAILURE);
}

// Sección crítica aquí

// Operación signal (V)
accion.sem_op = 1; // Incremento
if (semop(id_semaforo, &accion, 1) == -1) {
    perror("Error en operación signal");
    exit(EXIT_FAILURE);
}
```

Control de memoria compartida:

```
#include <sys/shm.h>

int main() {
```

```
    key_t clave_memoria;
    int id_memoria;
    int *contador;

    // Crear memoria compartida
    clave_memoria = ftok("/bin/ls", 'M');
    id_memoria = shmget(clave_memoria, sizeof(int), 0666 | IPC_CREAT);
    contador = (int *)shmat(id_memoria, NULL, 0);

    *contador = 0; // Inicializar el contador

    // Acceso protegido con semáforos
    accion.sem_op = -1; // Wait
    semop(id_semaforo, &accion, 1);

    (*contador)++; // Incrementar el contador

    accion.sem_op = 1; // Signal
    semop(id_semaforo, &accion, 1);

    // Liberar recursos
    shmdt(contador);
    shmctl(id_memoria, IPC_RMID, NULL);

    return 0;
}
```

PRACTICA 7: COLA DE MENSAJES

cola de mensajes, es una forma de comunicacion entre procesos que es especialmente util porque puede enviar y recibir mensajes complejos, es persistente, es decir que si un proceso envia un mensaje a la cola y el programa central no esta funcionando, estos mensajes quedaran almacenados en la cola.

Pasos basicos para la creacion de una cola de mensajes:

- crear o acceder a la cola con msgget()
- enviar un mensaje con msgsnd()
- recibir un mensaje con msgrcv()
- liberar la cola cuando ya no sea necesario con msgctl()

Formato de los mensajes:

```
struct msgbuf {
    long mtype;   // Tipo de mensaje
    char texto[100]; // Contenido del mensaje
};
```

Crear o acceder a una cola:

```
key_t clave = ftok("/tmp", 65); // Generar clave única
int id_cola = msgget(clave, 0666 | IPC_CREAT); // Crear o acceder a la cola
if(id_cola == -1){
    perror("Error al crear la cola de mensajes");
    exit(EXIT_FAILURE);
}
```

Enviar un mensaje:

```
struct msgbuf mensaje;
mensaje.mtype = 1; // Tipo del mensaje
strcpy(mensaje.texto, "Hola desde el productor!");

//antes de enviarlo si, el mensaje es grande entonces deberiamos hacer:

memset(&mensaje, 0, sizeof(mensaje)); //esto nos asegura dejar todos los
campos por default en 0

if (msgsnd(id_cola, &mensaje, sizeof(mensaje.texto), 0) == -1) {
        perror("Error al enviar mensaje");
} else {
```

```
        printf("Mensaje enviado: %s\n", mensaje.texto);
}
```

Recibir un mensaje:

```
struct msgbuf mensaje;
if (msgrcv(id_cola, &mensaje, sizeof(mensaje.texto), 1, 0) == -1) {
        perror("Error al recibir mensaje");
} else {
        printf("Mensaje recibido: %s\n", mensaje.texto);
}

//cuando lee el tipo 2 y sale error puede por -1 y
(errno == ENOMSG) //si se cumple esto deberiamos leer el siguiente.

//espera no bloqueante o (no activa)

struct msgbuf mensaje;
if (msgrcv(id_cola, &mensaje, sizeof(mensaje.texto), tipo de prioridad, IPC_
NOWAIT) == -1) {
        perror("Error al recibir mensaje");
} else {
        printf("Mensaje recibido: %s\n", mensaje.texto);
}

//cuando lee el tipo 2 y sale error puede por -1 y
(errno == EINTR) //si se cumple que la cola fue interrumpida por una señal
```

Eliminar la cola:

```
if (msgctl(id_cola, IPC_RMID, NULL) == -1) {
        perror("Error al eliminar la cola de mensajes");
} else {
        printf("Cola de mensajes eliminada\n");
}
```

- **¿Qué pasa si la cola está llena?**

- Si usas `IPC_NOWAIT` en `msgsnd`, recibirás un error en lugar de bloquearte.

- **¿Qué pasa si no hay mensajes en la cola?**
  - Si usas `IPC_NOWAIT` en `msgrcv`, recibirás un error.
- **¿Cómo manejar tipos de mensajes?**
  - Puedes filtrar por tipo usando el cuarto parámetro de `msgrcv`.

ACLARACIONES PERSONALES:

(char *)→ guarda la direccion de memoria de otro dato, ej: char *palabra = “hola”

(char **) → apunta a otro puntero, se usa para crear el arreglo de cadena

Memoria Dinamica:

malloc()→ asigna memoria especifica en tiempo de ejecucion.

realloc()→ cambia el tamaño de un bloque previamente ya creado

free() → liberra el espacio de memoria

PRACTICA 8-PLANIFICACION DE PROCESOS.

Introduccion: profundizar en las caracteristicas multiproceso construyendo un planificador basado en señales, con un sistema que implementa 3 colas (NO APROPIATIVOS), Los procesos de nivel 1 siguen un “politica” de FCFS, los procesos de nivel 2 siguen una politicas de planificacion de Round Robin de espera 3 segundos y los procesos de nivel 3 siguen una politica de prioridades no apropiativas (menor valor, mayor prioridad). Siendo los procesos de nivel 1 los de mayor prioridad y los de nivel 3, los de menor prioridad(seran los ultimos en atenderse.

1.El planificador de procesos a nivel de usuario prosched vamos hacerlo con una cola de mensajes

2.habra un proceso que este encargado de recibir las solicitudes de ejecucion de procesos

3.Este proceso va a construir una cola de mensajes y la eliminara ordenadamente cuando se necesite, construira la estructura de datos correspondiente para cada proceso y la encolara en la cola de mensajes que proceda.

4.Creara un segundo proceso encargado de la planificacion y liberara todos los recursos(procesos y colas de mensajes) cuando se indique su finalizacion por parte del usuario.

5.el planificador admitira solicitudes por teclado y por un archivo de configuracion, cuando las solicitudes see obtengan por teclado el final del fichero (EOF) se logra pulsando Ctrl + D.

ESPERAS BLOQUEANTES (CUANDO NO SE PUEDAN HACER ESPERAS ACTIVAS):

2. Ejemplos de “espera bloqueante” o sincronización correcta (lo que SÍ debes usar)

1. **Uso de llamadas bloqueantes de colas de mensajes** (System V):

c — Copiar

```c
struct msgbuf mensaje;
// Bloquea hasta que llegue un mensaje de tipo 1
msgrcv(id_cola, &mensaje, sizeof(mensaje) - sizeof(long), 1, 0);
// Aquí el proceso NO consume CPU mientras no haya mensajes
// Se “despierta” cuando un nuevo mensaje tipo 1 aparece.
```

2. **Uso de semáforos (semop) con bloqueo**:

c — Copiar

```c
struct sembuf op_down = {0, -1, 0};

// Bloquea si el valor del semáforo está en 0, hasta que otro proceso lo increm
semop(id_sem, &op_down, 1);
// Continúa cuando obtiene el semáforo.
```

3. **Lectura bloqueante de ficheros, tuberías o FIFOs**:

c — Copiar

```c
int fd = open("nombreFIFO", O_RDONLY);
char buffer[100];

// read bloquea hasta que haya datos en la FIFO
int leidos = read(fd, buffer, sizeof(buffer));
// Si no hay nada que leer, el proceso se “duerme”
// y no consume CPU hasta que alguien escriba en la FIFO.
```

4. **Uso de señales + pause()**:

```c
volatile sig_atomic_t pedidoListo = 0;

// Manejador de señal que pone pedidoListo = 1
void manejador(int sig) {
    pedidoListo = 1;
}

...
while (!pedidoListo) {
    pause(); // Bloquea el proceso hasta recibir CUALQUIER señal
             // No consume CPU en este tiempo.
}
// Al recibir la señal, se despierta y sale.
```

5. **Uso de select/poll/epoll** (en sockets u otros descriptores) en modo bloqueante:

```c
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(fd_socket, &readfds);

// Select bloqueante: se “duerme” hasta que haya un evento de lectura
int ret = select(fd_socket+1, &readfds, NULL, NULL, NULL);
if (ret > 0 && FD_ISSET(fd_socket, &readfds)) {
    // Hay datos disponibles, se despierta y los lee
}
```