TEMA 1 –Introducción SO

Definición:

- Es un programa que hace de intermediario entre el usuario y el hardware de un computador
- El programa que siempre se está ejecutándose en el computador. Usualmente se le llama *kernel* o núcleo del SO. Es solo una pequeña parte del S.O. El resto de cosas son programas del sistema y programas de aplicación.

Definición funcional

- Es un gestor de recursos o programas
- Tiene que ser de alto rendimiento
- Tiene que ser eficiente a la hora de gestionar los recursos

Tareas del SO

- Comunicación con los periféricos
- Administración de recursos
- Coordinación de procesos simultáneos (multiproceso y multitarea) – LAS APLICACIONES NO SE EJECUTAN NUNCA A LA VEZ, ES EL SO EL QUE SE ENCARGA DE HACER CREER TANTO A USUARIO COMO AL PROGRAMA QUE ES UNICO
- Gestión de memoria
- Gestión de datos
- Gestión de comunicaciones y redes

COMPONENTES DE UN SISTEMA INFORMATICO

USUARIO conecta con PROGRAMAS usa el SO para conectar con el HARDWARE

NI HADWARE NI SO CONOCEN AL USUARIO NINGUN COMPONENTE SE PUEDE SALTAR AL ANTERIOR

PUNTO DE VISTA DEL PROGRAMADOR

El SO proporciona al programador una serie de abstracciones que le permiten acceder al hardware de forma sencilla.

SISTEMAS OPERATIVOS

Solo es importante GNU

LIBERTADES DE GNU

- Libertad 0: libertad para ejecutar el programa con cualquier propósito
- Libertad 1: libertad para estudiar y modificar el programa
- Libertad 2: libertad de copiar el programa y poder pasarlo a quien se desee
- Libertad 3: libertad de mejorar el programa, y hacer públicas, las mejoras, de forma que se beneficie toda la comunidad
- **Las libertades 1 y 3 obligan a que se tenga acceso al código fuente.**
- **Sin embargo, el proyecto GNU no significa software sin licencia**

- **La Licencia GPL (General Public Licence) garantiza las libertades anteriores**

FIRMWARE

Se ejecuta al arrancar el ordenador, inicializa el sistema y busca la imagen del kernel.

FUNCIONAMIENTO DEL COMPUTADOR

- Los dispositivos E/S y CPUs funcionan de forma paralela
- Cada controladora se encarga de un tipo de dispositivo concreto
- Controladoras tienen un buffer local
- CPU se encarga de mover datos entre memoria principal y los buffers locales de las controladoras, mediante el uso de interrupciones para informar a la CPU cuando han finalizado su trabajo. La interrupción transfiere el control a la rutina de atención de la interrupción (vectores de interrupción). Se almacena la dirección de la instrucción interrumpida y mientras que se atiende una interrupción se deshabilita la llegada de nuevas interrupciones.
- **EL KERNEL DEL SO SON LAS RUTINAS DEL TRATAMIENTO DE INTERRUPCIÓN.**
- El SO es el encargado de preservar el estado del procesador, guardando el contenido de los registros y del contador de programa para poder restaurarlo cuando finalice la interrupción.
- Aparte de interrupciones hardware existen interrupciones software (causadas por peticiones del usuario) y las excepciones causadas por un error.

ESTRUCTURA DE ALMACENAMIENTO

El SO se encarga de administar la memoria principal y la secundaria.

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

(Del dibujo es importante saber que toca quien lo toca y donde lo vuelca)

ESTRUCUTRA DE E/S

Los controladores (drivers) son el software que permite al SO comunicarse con el hardware a través de la controladora. Hay un driver para cada controladora que proporciona una interfz uniforme mediante la cual comunicarse con el dispositivo hardware.

OPERACIONES DEL S.O

Los errores de un o programa pueden afectar a otros programas o al propio S.O y eso no se puede permitir por lo tanto el SO **Se encarga de que cada aplicación sea independiente.**

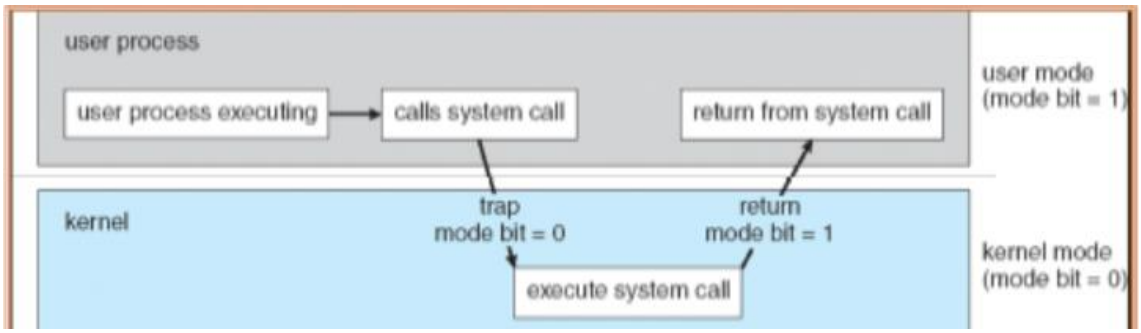
MODO DUAL

La CPU permite en el modo dual permite diferenciar entre el modo usuario o el modo kernel.

El modo kernel permite modificar el S.O (principalmente las interrupciones o instrucciones concretas como por ejemplo modificar el modo) si estoy en el usuario y lanzo alguna instrucción prohibida salta una excepción.

El cambio de bit de modo se lanza en una rutina de tratamiento de interrupción.

Trap = interrupción software



Actualmente vienen en anillos 0-3 siendo el 0 el kernel el 3 el usuario pero el 1 y el 2 no se usan habitualmente (principalmente para maquinas virtuales).

MULTIPROGRAMACION

El SO se encarga de ordenar las operaciones para que el procesador este siempre en ejecución. Aprovecha los huecos entre operaciones para aplicar otras operaciones. Como los tiempos de espera son mayores que los de ejecución permite poner diferentes aplicaciones a la vez.

SINO HAY NADA QUE EJECUTAR EL S.O PONE A EJECUTAR EL PROCESO NULO.

TIEMPO COMPARTIDO

El S.O asigna un tiempo concreto máximo a cada aplicación y si en ese tiempo no se cumple toda la acción manda esa aplicación otra vez a la cola de espera y pasa a la siguiente, si en ese tiempo se cumple la operación en cuanto acaba pone la siguiente operación.

Gestión de procesos

Proceso = programa en ejecución

Los procesos necesitan recursos, que al finalizar son recursos recuperados por el S.O. El sistema tiene varios procesos en ejecución algunos de usuario y otros de S.O.

El S.O es responsable de

- Crear y eliminar los procesos de usuario y del sistema
- Suspender y reanudar procesos.
- Proporcionar mecanismos para la comunicación y sincronización de procesos
- Proporcionar mecanismos para el tratamiento de los interbloqueos.

La memoria principal es fundamental en el funcionamiento de un sistema informatico. Los datos e instrucciones deben estar en la memoria principal para qe el procesador pueda utilizarlos.

En cuanto a la gestión de memoria, el S.O. se encarga de controlar que partes de la memoria están en uso y por quien, decirdir que datos y procesos añadir o quitar de la memoria y de asignar espacio de memoria.

El S.O proporciona una visión uniforme de los medios de almacenamiento y algunas de las responsabilidades del S.O en cuanto a la gestión de almacenamiento son creación y borrado de ficheros y directorios, soporte de primitivas para manipular archivos y directorios y gestión de permisos

SERVICIOS DEL SISTEMA OPERTATIVO

El S.O proporciona unas funciones para los usuarios(interaz, ejecutar programas, operaciones E/S, manipulación de archivos, comunicaciones, detección de errores) y otras para garantizar la eficiencia del propio sistema (asignación de recursos, contabilidad y protección)

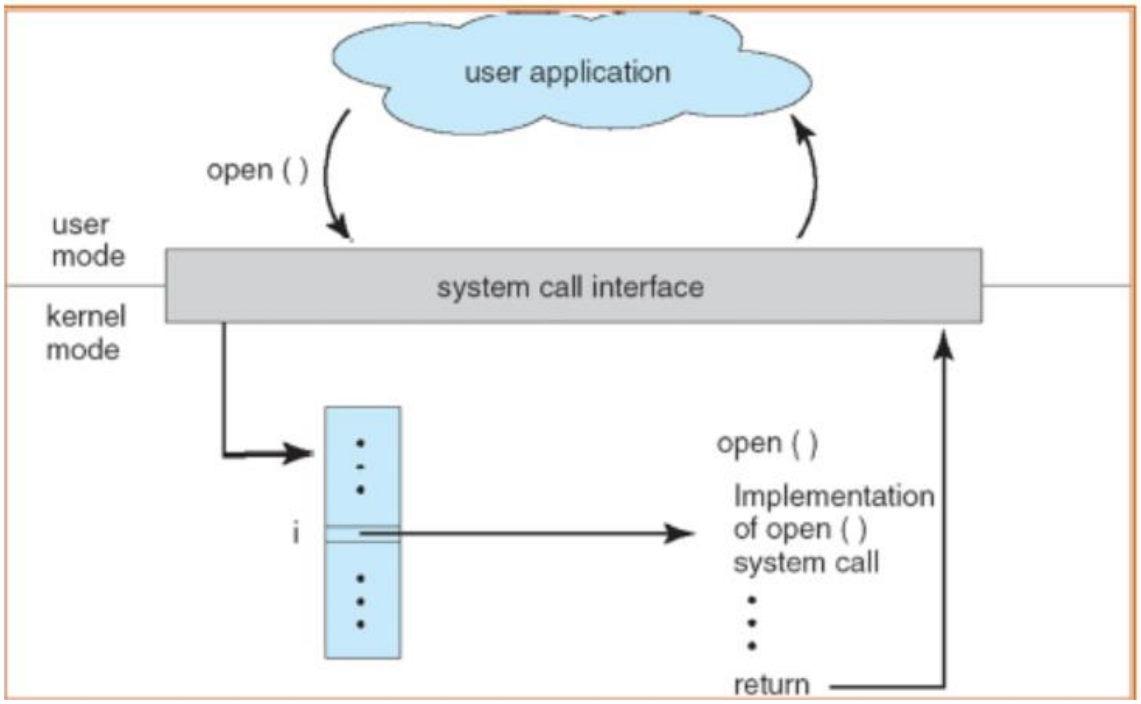
INTERFAZ DE USUARIO

Permite al usuario interactuar con el sistema. Puede ser de dos tipos modo texto (cumple simplemente las ordenes que le demos) o interfaz grafica (diversas pantallas y menos estándar)

LLAMADAS AL SISTEMA

Proporciona una interfaz de programación para invocar los servicios que ofrece el S.O. Estan escritas en C. El programador utiliza una API para las llamadas al sistema. Las apis mas comunes son Win32 API para Windows y la API POSIX para UNIX.

Para la implementación es tener asociada a cada llamada al sistema a un numero y se mantiene una tabla indexada con las direcciones de las llamadas. El interfaz de llamadas al sistema invoca a la llamada correspondiente del sistema operativo y devuelve el estado y los valores de retorno. Solo hay que conocer cómo utilizar la API. Los parámetros se pasan de diferente forma ya que estamos en diferentes formas. Se meten en registros, tablas de memoria o en la pila (este ultimo no limita ni el numero ni el tamaño de los parámetros).



TIPOS DE LLAMADAS DEL SISTEMA

- Control de procesos
- Administración de archivos
- Administración de dispositivos
- Mantenimiento de información
- Comunicaciones

PRORGRAMAS DEL SISTEMA

Son programas que proporcionan un entorno cómodo para desarrollar y ejecutar programas.

TEMA 2 INTERPRETE DE MANDATOS

INTERPRETE DE MANDATOS O SHELL: Es un programa que hace de interfaz de texto entre el usuario y el S.O. Tiene la función de interpretar mandatos y lanzar programas. Tambien proporciona funcionalidades de programación en forma de scripts.

Es una aplicación que forma parte del S.O y ofrece una interfaz de la cual usuarios y admnistrados pueden operar el sistema.

- Consultar el estado del sistema
- Recorrer el árbol de directorios
- Lanzar otras aplicaciones

Sintaxis típica: (prompt) mandato [parámetro][parámetro]

La Shell recorre uno a uno los directorios del path que están dentro del sistema buscando la orden del sistema.

Linux

Para activar los terminales ctrl+alt+f1..f6 y para salir ctrl+alt+f7

Ctrl+c finaliza un mandato y con ctrl+z se paraliza

Mandatos en primer plano: no deja que se ejecute otro mandato hasta que no acaba un mandato

Mandatos en segundo plano: mandatos que se ejecutan con & antes de que finalice la Shell

Jobs muestra los procesos que están en segundo plano o pausados.

Bg: manda un proceso de pausado a segundo plano

fg: manda un proceso a primer plano

nano: editor en modo texto

FICHEROS

i-nodos: son las representaciones de los ficheros en el disco duro.

Bloques de arranque: Tienen el kernel del sistema.

Superbloque: indica que empiezan los i-nodos.

Hay un inodo con los atributos de los ficheros, un identificador con una cabecera y apuntadores a i-nodos.

EN UNIX TODO SON I-NODOS INCLUIDOS LOS DIRECTORIOS. LOS I-NODOS APUNTAN A OTROS I-NODOS NO A DATOS.

Todo cuelga de un directorio principal o raíz /.

Para encontrar un directorio se puede con la ruta absoluta (siempre empieza con /) o con la ruta relativa (nombre solo del directorio).

. es el directorio actual o .. directorio anterior.

DIRECTORIOS IMPORTANTES DEL SISTEMA

- **/etc**: contiene los archivos de configuración del sistema
- **/bin**: contiene ejecutables (o binarios, de ahí su nombre) básicos del sistema
- **/sbin**: contiene ejecutables que son utilidades para la administración del sistema
- **/dev**: contiene las definiciones de los dispositivos del sistema (discos duros, medios extraibles, puertos, etc...)
- **/boot**: contiene la información necesaria para el arranque del sistema (kernel, configuraciones del gestor de arranque, etc...)
- **/home**: contiene los directorios de trabajo de cada usuario del sistema
- **/lost+found**: cuando se recupera alguna información "perdida" del disco se almacena en este directorio
- **/media /mnt**: se utiliza para albergar puntos de montaje habituales
- **/opt**: en este directorio se suelen instalar complementos u opciones de algunas aplicaciones
- **/root**: es el directorio de trabajo del usuario "root" (administrador principal del sistema)

- **/proc**: contiene archivos que proporcionan información de todo el sistema
- **/tmp**: directorio donde se almacena información temporal
- **/lib**: librerías básicas del sistema
- **/usr**: directorio donde se guardan datos y programas compartidos por varios usuarios:
  - /usr/bin
  - /usr/lib
  - /usr/doc
  - ...
- **/var**: datos variables, como colas de impresión, servidor web

Montar un sistema de ficheros:

```
mount -t iso9660 /dev/hdc /media/cdrom
mount -t vfat /dev/hdb2 /media/dos
mount -t ntfs /dev/sda1 /media/windows
mount -t ntfs /dev/sda3 /home/
```

Desmontar un sistema de ficheros:

```
umount /media/cdrom
```

Mostrar los sistemas de ficheros montados:

```
mount
```

vfat son los pendrive, (hd y sd) hard o sata disk, la letrita corresponde al número del disco duro y el numero a la partición.

Imaginemos que el fichero /media esta con un file txt y hago un mount -t ntfs/dev/sdb/media ¿Qué pasa con file.txt? no puedo acceder a el ¿se ha borrado file.txt? no se ha borrado sigue en el sistema. ¿y si hago unmount /media? Recupera file.txt porque nunca a dejado de apuntar a él en el i-nodo y el sistema de ficheros no ha cambiado

¿Cuántos MANDATOS EN 1 PLANO PUEDE HABER? Respuesta 1 o ninguno

¿Cuántos mandatos en 2 plano puede haber? Los que te deje la maquina

Para cambiar de directorios cd (.. o ruta), ls (lista), ls -a (muestra los ocultos), -l te los lista con detalle, mkdir crea un directorio, cat muestra el contenido de un fichero, less y mor permite moverte por los ficheros, wc cuenta palabras, líneas y bites, cut selecciona columnas especificas de un fichero, -d indica el separador, -f el numero de columna, touch cambia la fecha de modificación, ln crea un enlace de un fichero a otro, Enlace fuerte (es el de por defecto) si borro el fichero se borra, Enlace simbolico (-s) crea un acceso directo si pierdo el fichero el enlace se queda invalido, find(busca ficheros y directorios, atributos y es recursivo) y grep (muestra la líneas de un fichero que se presentan un patrón. Busca un contenido en un fichero (palabras, expresiones regulares..). Los permisos te los muestra con ls- l y tiene para user (u), grupo (g), resto usuarios (o) y 3 permisos (r,w,x) (leer, escritura, ejecución)

EL EJERCICIO DE SCRIPT 4 ES MUY MUY IMPORTANTE → REPASAR

TEMA 3 → ENTRA SOLO ENTRADA /SALIDA SOBRE SISTEMA

Programacion C

```
#include <stdio.h>

int main(){printf (“Hola Mundo”); return 0}
```

/*..*/ o // para comentarios

→ directiva del procesador

<> copia en estas lineas todo el codigo del archivo que vaya entre los angulos.

Main casi siempre es int ya que devuelve un numero, 0 todo ok otro error.

Para compilar en linex gcc archivo.c -o genera el ejecutable

gcc archivo.c -c -o no enlaza el archivo solo crea el objeto y gcc archivo1.o archivo2.o enlaza ficheros y genera el ejecutable

DATOS

Las declaraciones de variable se hace solo en tiempo de compilacion despues esa linea deja de existir para el procesador.

CARACTERES ESPECIALEs

\n → salto de linea \r → retorno de carro \t→ tabulacion \b→ espacio \\ → /

/’→ ‘ \’’ → “ \0 → nulo

Operadores

&&,\\, !, and, or , not

OTROS

sizeof (variable o tipo) dice lo que ocupa en memoria

ESTRUCTURAS DE CONTROL

if(expresion){..} else{..}

switch(expesion){ case valor1… breake; .. defaul…}

while(expresion){…}

do{}while()

for(i;i<j;act);{} → las inicializaciones se hacen fuera

ARRAYS

NO EXISTE EL INDEXOUTBOUNDEXCEPTICION Y NO SE PUEDE HACER INT A[N]; SE HACEN CON PUNTEROS TODO

PUNTEROS

int *ptr = &variable

*ptr = 127 **modifica el valor de la variable apuntada**

p++ aunmenta en 1 lo que ocupa el tipo de datos pasa de A[i] a [i+1]

FUNCIONES

malloc(size_t size); devuelve un puntero a una zona de memmmoria de size bytes. Como devuelve un puntero void*hay qye convertirlo al tipo de puentero que se quiere usar. ES DECIR SE LE PASA NºELEMENTOS* SIZEOF(TIPO)

free(void *ptr): libera la zona de memoria apuntada por ptr y obtenida previamente con malloc

```
float *a;
int *b; // punteros a enteros

//se reserva espacio en memoria para 24 floats
a = (float *) malloc (24*sizeof(float));

//se reserva espacio en memoria para 16 enteros
b = (int *) malloc (16*sizeof(int));
…
// Utilización de a y b
…
//se libera los espacios de memoria malloc
free(a);
free(b);
```

PARA PASAR POR REFERENCIA HAY QUE PASAR LOS PUNTEROS

void intercambiar(int *a, int *b){

int aux;; aux= *a; *a=*b; *b=aux;}

//llamada a la funcion

int x=10, y = 20; intercambiar(&x, &y)

Para cambiar la direccion del puntero tienes que pasar ** puntero del puntero

RELACION DE PUNTEROS Y ARRAS

vectoro[0] = *vector

vector[1]= *(vector+1)

SI HACES SIZEOF DE UN PUNTERO DARA EL VALOR DEL PUNTERO (8)

STRINGS

char string[]=”HOLA”;

char *string2=”Hola”;

char *string3={‘H’,’O’,’L,’A’, \0} → SIEMPRE EL ULTIMO CARACTER ES \0 Y POR LO TANTO HAY QUE RESERVARLE UNA POSICION DE MEMORIA

OJO CON LOS INT

int *a = [15,25,31] → crea un array de 3 con esos numeros

int *a = 16 → reservo la posicion de memoria 16

FUNCIONES DE STRING

STRLEN→ longitud sin \0

strcat→ concatena dos strings

strcpy→ copia un string sobre otro (ojo b tiene que tener espacio)

```
char *b;
b= (char *) malloc ((strlen(a)+1)*sizeof(char));
```

strdup→duplica un string

strdup→ hace el malloc por si solo asique usar el FREE...

strchr→ busca un carácter en el string

HAY QUE IMPLEMENTAR <string.h>

ESTRUCTURA

typedef struct Complejo{ float real; float img;} Tcomplejo;

para acceder a los elementos se usa el .

Si se trata de un puntero a un registro se puede atajar con el operador →

FUNCION MAIN

int main(int argc, char *argv[])

si pones argv[1][3] accedes a la “4” letra del elemento en la posicion 1 de argv

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
	// Imprime numero de argumentos
	printf("Se han recibido %d argumentos\n",argc);
	int i;
	for(i = 0; i < argc; i++)
	{
		// Imprime argumento a argumento
		printf("Argumento %d = %s\n", i,argv[i]);
	}
	return 0;
}
```

ENTRADA/SALIDA

Definidas en stdio.h

printf: imrime en pantalla

para poner cosas especiales (variables o tal) se usa %letra_tipodedato_representado.

printf("Hoy es %s y son las %d:%d|n", dia,hora, min);

Especificador de formato		Descripción
%c		carácter individual
%d	%i	entero decimal con signo
%u		entero decimal sin signo
%o	%ou	entero octal con y sin signo
%x	%X	entero hexadecimal con signo
%xu	%Xu	entero hexadecimal sin signo
%f	%F	double con notación decimal-punto
%e	%E	double con notación exponencial
%g	%G	double
%p		valor de un puntero
%s		cadena de caracteres

scanf: lectura por teclado

scanf("%d", &limite); → & es un paso por referencia

LLAMADAS DEL SISTEMA ESTO ENTRA SI O SI

DESCRIPTORES DE FICHEROS

Es un entero no negativo que identifica un fichero abierto. Esta lista de ficheros abiertos se crea por cada programa.

Los 3 primeros valores estan asignados por defecto 0→ entrada estandar (stdin) 1→ salida estandar (stdout) 2→ salida error (stderr).

Siempre intenta devolver la mas baja posible desde la 3

Abrir ficheros: open(char* name, int flags, [mode]);

Argumentos:

- name: cadena de caracteres con el nombre del fichero
- flags:
  - O_RDONLY: El fichero se abre de sólo lectura
  - O_WRONLY: El fichero se abre de sólo escritura
  - O_RDWR: El fichero se abre de lectura y escritura
  - O_APPEND: Se escribe a partir del final del fichero
  - O_CREAT: Si no existe el fichero, se crea y no da error
  - O_TRUNC: Se trunca el fichero
  - Se pueden aplicar varios a la vez separándolos con |
- mode: permisos del fichero (sólo con flag O_CREAT)

mode: bits de permisos del fichero:

- S_IRUSR, S_IWUSR, S_IXUSR: R, W, X (user)
- S_IRGRP, S_IWGRP, S_IXGRP: R, W, X (group)
- S_IROTH, S_IWOTH, S_IXOTH: R, W, X (others)

PARA APILAR PERMISOS O FLAGS APILAR CON |

Lectura ficheros: size_t read (int fd, void* buf, size_t n_bytes)

fd: descriptor de fichrero del fichero que se va a leer

buf: buffer donde se van a almacenar los Bytes leidos

n_bytes: numero de bytes que se quieren leer

Escritura de ficheros: size_t wirte(fd, ovid* buf, size_t n_bytes);

write no tiene porque escribir todos los bytes, hay que comprobar que ha escrito el mismo numero de bytes que le he dicho

Posiccionamiento de punteros: off_t lseek(int fd, off_t offset, int whence);

whence: tipo de desplazamiento del puntero:

- SEEK_SET: posición del puntero = offset
- SEEK_CUR: posición del puntero = posición actual + offset
- SEEK_END: posición del puntero = tamaño fichero + offset

cierre fuchero: int close(int fd);

DUPLICAR UN DESCRIPTOR DE FICHERO

TODO LO QU ESTA ASIGNADO SE COMPARTE (LOGS, FLAGS, POSICIONES…), SI AVANZO EN UNA AVANZO EN LA OTRA.

Llamada **dup:** duplica en la primera que hay libre

Llamada **dup2:**dice que quiero duplicar y en que numero. Si esta abierta donde voy a copiar la cierra y la duplica.

Si quieres escribir en un fichero duplicas la salida y lo metes en la 1(que es la salida) **OJO SI MACHACO LA SALIDA DE PANTALLA TE JODES, ASIQUE TOCA SALVARLA ANTES EN OTRA POSICION.**

FUNCIONES DE BIBLIOTECA

//COPIAR DEL PPT

ENTRADA/SALIDA CON FICHEROS HIPER IMPORTANTE

//AQUÍ FALTAN MUCHAS COSAS SUPER IMPORTANTES

HAY 3 TIPOS DE SALIDA PARA FPRINTF

FILE* STDERR (2) FILE* STDOUT (1) FILE*STDIN (0)

PARA PRUEBAS PINTAR POR LA SALIDA DE ERROR PORQUE NO HAY BUFFER

APLICACIONES MODULARES

Un archivo de cabecera (.h)

- Define la interfez del modulo, es decir, que ofrece el modullo definido.
- Debe contener definiciones de tipos y declaraciones de funciones (prototipos)

Un archivo fuente (.)

- Define los detalles de implementación del modulo
- Inclusión de su correspondiente fichero de cabecera → su fichero .h
- Implementación de funciones declaradas y no declaradas en el fichero de cabecera.
- Los ficheros .h propios se incluyen #include “file.h”

Librería estatica

- “se copia” el código dentro del ejecutable cuando lo compilamos
- Solo se copia aquella parte de la librería que se necesite
- Contiene los .o

Librería dinámica

- No se copia en nuestro programa al compilarla
- Cuando se ejecuta un programa que la requiere, cada vez que nel código necesite algo de la librería ira a buscarlo a esta
- Ocupan mucha menos memoria
- Permite hacer mejoras de código de la librería y solo hay que compilarla a ella.
- Se utilizan en tiempos de ejecución, por lo tanto el sistema tiene que saber donde esta. Para ello tengo que establecer el $LD_LIBRARY_PATH_= $LD_LIBRARY_PATH.

Crear librerías

- Estatica ar -rv libnombre.a fuente1.o fuente2.o… -r (remplazo)
- Compilarlo gcc -o mi programa miprograma.c -l<path1>

//REVISAR APUNTES

PRACTICA

$./test (head,tail,longlines) -N

Nombre libreria = libreria.h

Cabeceras

Int head (int n);

Int tail(int n);

Int longlines(int n);

Archivo .c -> libreria.c

#include <stdio.h>… #include “libreria.h”

Cabecera de funciones auxiliaries.

Codigo

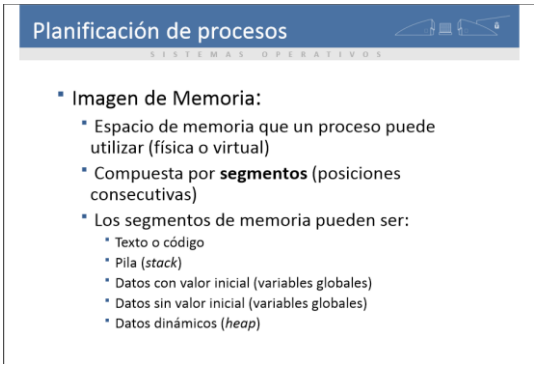
NO HAY FUNCION MAIN→ implica hacer otro fichero que es $test.c programa de pruebas.

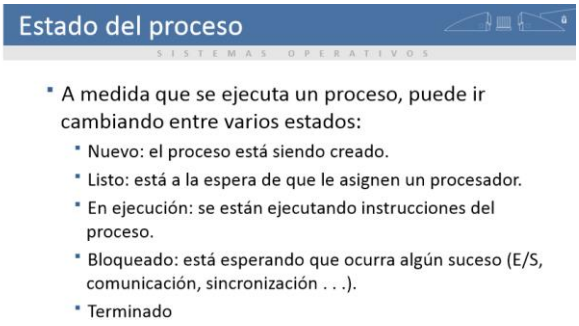
#include …. Include “librería.h”

Int main (argc, argv){ llamara a head, tail, llonglines)

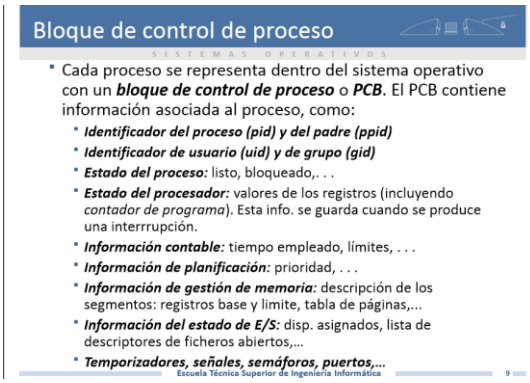
TEMA 4 – GESTION DE PROCESOS

LOS PROCESOS SON INDEPENDIENTES, AUNQUE SEAN INSTANCIAS DE UN MISMO PROGRAMA.

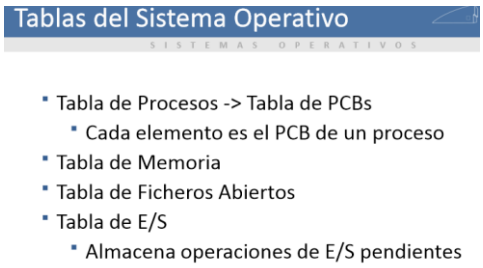




Tras terminar la espera pasaría al estado de listo y es el sistema la que decidirá moverla a ejecución.



← no hace falta saberse donde esta cada cosa



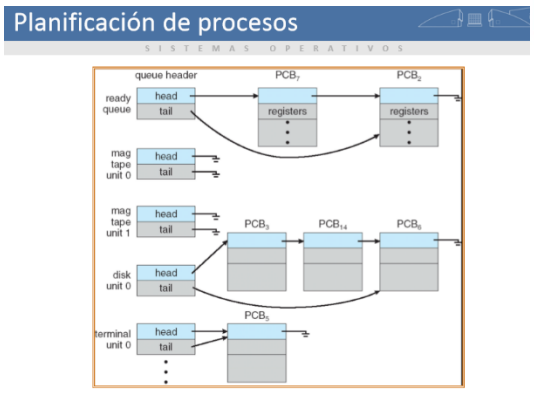
La tabla de procesos que tiene por cada procesos su tabla PCB.

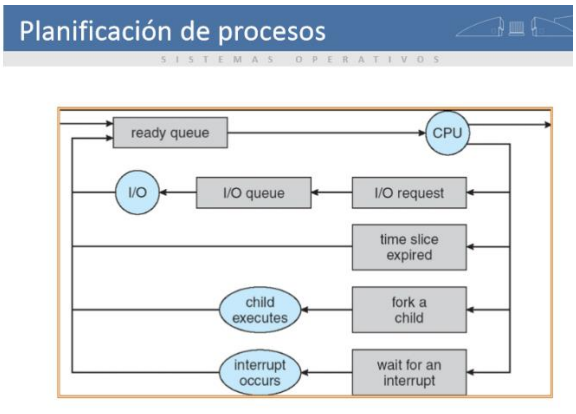
De la tabla de ficheros abiertos, son punteros a los ficheros que utiliza el programa.

Objetivo de la multiprogramación: Admitir varios procesos en ejecución a lmismo tiempo para maximiar el uso de la CPU.

Los procesos se organizan en colas:

- **Cola de trabajos: Todos los del sistema**
- **Cola de procesos preparados: Todos los procesos que están listos esperando para ejecutar**
- **Varias colas de dispositivo que contienen los procesos que están a la espera del alguna operación de E/S.**



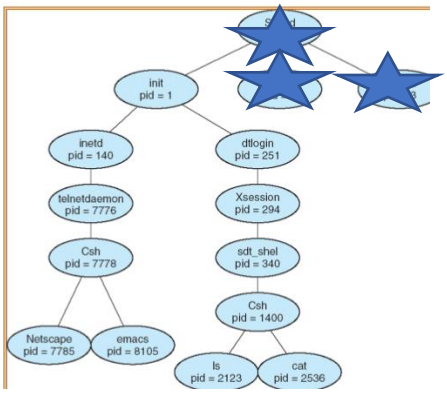


El sistema espera a que alguien le llame, ocupa el procesador y coloca un nuevo proceso, a través de la rutina de excepciones.

El planificador no entra

El cambio de contexto es la fase final de una interrupción. Es el cambio de un proceso a otro.

Creación de procesos:

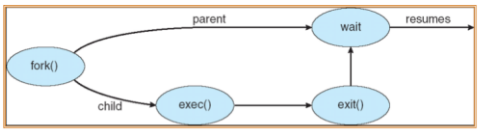


- Cuando un proceso crea otro proceso nuevo existen distintas posibilidades:
  - Ejecución:
    - Padre e hijo ejecutan concurrentemente. bg
    - El padre espera hasta que el hijo termine fg
  - Recursos:
    - Padre e hijo comparten todos los recursos.
    - El hijo solo comparte una parte de los recursos del padre.
    - No se comparte ningún recurso.
  - Espacio de direcciones
    - El hijo tiene un duplicado del espacio de direcciones del padre.
    - El hijo carga un nuevo programa.

Terminacion de procesos

- Un proceso termina cuando ejecuta su última instrucción y pide al SO que lo elimine
  - El proceso puede mandar información de finalización a su proceso padre
  - El SO libera los recursos asignados al proceso
- Un padre puede terminar la ejecución de alguno de sus hijos por diversos motivos:
  - El proceso hijo ha excedido el uso de algunos recursos
  - La tarea asignada al hijo ya no es necesaria
  - El padre abandona el sistema, y el SO (algunos) no permite que un hijo continúe sin padre => terminación en cascada.

- Creación/terminación de procesos en UNIX:
  - **fork()** : llamada al sistema para crear un nuevo proceso.
  - **exec (...)**: llamada al sistema para reemplazar el espacio de memoria actual de un proceso por un programa nuevo.
  - **exit(...)**: llamada para finalizar el proceso.
  - **wait(...)**: llamada para esperar por un proceso. Se recibe la información de finalización del proceso por el que se espera



Cuando hago una copia con fork se crea otro PCB con un pid diferente pero tiene el pid del padre guardado. Pero cuando se hace la copia los 2 tienen el mismo contador de programa.

Si haces un var = fork(); var vale el valor del pid del proceso hijo.

El exec solo es un cambio de imagen de memoria, pero los ficheros seguirán abiertos

Las modificaciones una vez realizado el fork no afectan padre <-> hijo.

Un proceso es independiente sino puede afectar o verse afectado por otros procesos del sistema.

COMUNICACIÓN INTERPROCESOS (IPC)

la cooperación entre procesos requiere mecanismos de IPC puede consister en:

- Avisar a un proceso de la ocurrencia de eventos
  - Señales: mecanismo estándar para informar a un proceso que ha ocurrido un evento
  - Semaforos: mecanismo de bloqueo de un proceso hasta que ocurra un evento
- Transferencia de datos entre procesos:
  - Pipes: permite a un proceso hijo comuncarse con su padre a través de un canal de comunicación..
  - Memoria compartida: Establece una región de memoria a la que pueden acceder los procesos cooperativos. Los procesos se comunican escribiendo o leyendo datos en la memoria compartida
  - Paso de mensajes: permite comunicarse sin compartir el espacio de direcciones de memoria.

Que sea un mecanismo de comunicación implica que puede tener mecanismos de sincronización.

SEÑALES

Mecanismo de comunicación entre procesos en S.O compatiblrs con POSIX.

Se utilizan para informar a un proceso de un evento o error

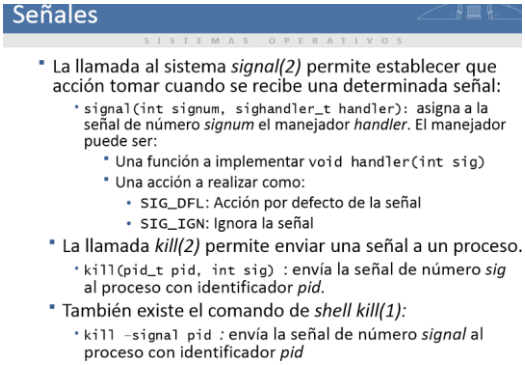
Tienen el mismo comportamiento de las interrupciones, es decir, deja de ejecutar lo que sea y ejecuta la rutina de tratamiento de señal. “Permite crear nuestras propias rutinas de procesos de interrupción del proceso”.

Cuando un proceso recibe una señal puede:

- Ignorar la señal
- Invocar la rutina de tratamiento por defecto
- Invocar la rutina de tratamiento propia.

<#signal.h> este fchero incluye varias señales ya definidas.

NOMBRE	DESCRIPCIÓN	REPROGRAMABLE	IGNORABLE
SIGKILL (9)	Destrucción inmediata del proceso	No	No
SIGSTOP (19)	Detiene el proceso	No	No
SIGTERM (15)	Terminación del proceso de forma normal	Si	Si
SIGINT (2)	Interrumpe el proceso (Por defecto termina el proceso) (Ctrl+C)	Si	Si
SIGUSR1 y SIGUSR2	Para uso del programador (Por defecto termina el proceso)	Si	Si



El mandato ps muestra los procesos.

Ejemplo con 2 procesos

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

int pid;
int main() {
  signal(SIGUSR1, manejador);
  pid = fork();
  if( pid == 0 ) // Hijo
  {
    while(1);
  }
  else // Padre
  {
    kill(pid, SIGUSR1);
    wait(NULL);
    printf("El hijo terminó\n");
  }
  return 0;
}
```

```
void manejador(int sig)
{
 if(pid == 0){
  printf(“Hijo: Recibida señal %d\n", sig);
  kill(getppid(), SIGUSR1);
  exit(0);
 } else {
  printf(“Padre: Recibida señal %d\n", sig);
 }
}
```

PIPES

LOS Pipes en modo lectura son bloqueantes esto permite que sean usados como mecanismos de sincroniacion.

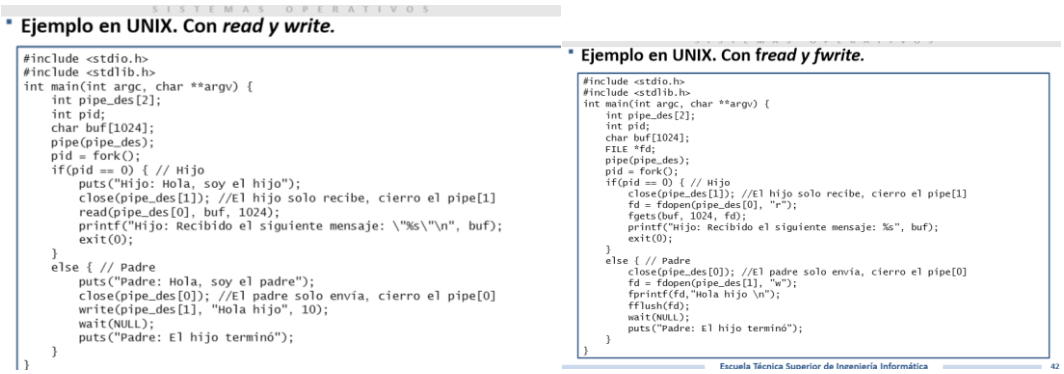
Son 2 entradas de descriptores de ficheros, se utilizan todas las funciones de ficheros.

Para convertir un pipe en un file * y esto permite usar otras funciones (fread, fwrite…) para hacer la conversión es fdopen.

Son unidireccionales.

Crear un pipe se hace con la llamada pipe(2 directores de ficheros) el 1 es la entrada del pipe y el 0 la salida

Ejemplo/// int array[2]; pipe (array);



Ejemplo en UNIX. Con *read y write.*

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    int pipe_des[2];
    int pid;
    char buf[1024];
    pipe(pipe_des);
    pid = fork();
    if(pid == 0) { // Hijo
        puts("Hijo: Hola, soy el hijo");
        close(pipe_des[1]); //El hijo solo recibe, cierro el pipe[1]
        read(pipe_des[0], buf, 1024);
        printf("Hijo: Recibido el siguiente mensaje: \"%s\"\n", buf);
        exit(0);
    }
    else { // Padre
        puts("Padre: Hola, soy el padre");
        close(pipe_des[0]); //El padre solo envía, cierro el pipe[0]
        write(pipe_des[1], "Hola hijo", 10);
        wait(NULL);
        puts("Padre: El hijo terminó");
    }
}
```

Ejemplo en UNIX. Con *fread y fwrite.*

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    int pipe_des[2];
    int pid;
    char buf[1024];
    FILE *fd;
    pipe(pipe_des);
    pid = fork();
    if(pid == 0) { // Hijo
        close(pipe_des[1]); //El hijo solo recibe, cierro el pipe[1]
        fd = fdopen(pipe_des[0], "r");
        fgets(buf, 1024, fd);
        printf("Hijo: Recibido el siguiente mensaje: %s", buf);
        exit(0);
    }
    else { // Padre
        close(pipe_des[0]); //El padre solo envía, cierro el pipe[0]
        fd = fdopen(pipe_des[1], "w");
        fprintf(fd,"Hola hijo \n");
        fflush(fd);
        wait(NULL);
        puts("Padre: El hijo terminó");
    }
}
```

Escuela Técnica Superior de Ingeniería Informática 42

THREADS

Las hebras son una unidad básica de utilización de la CPU. Comparten con otras hebras la sección de código, la sección de datos y otros recursos.

Cada hebra tiene su propio contador de programa, conjunto de registros de la CPU y pila, además de un identificador único.

Ventajas: capacidad de respuesta, compartición de recursos, economía (mas barato en memoria una hebra que un proceso) y uutilizacion en arquitecturas multiprocesador.

Tipos:

- Threads de usuario: proporcionadas por una librería de usuario.
- Threads del kernel: proporcionados por el S.O

#include <pthread.h>

Gcc file .c -lpthread -o programa

Para crear un thread pthread_create(*tid, *attr, *function,*param)

Pthread_exit(res);

Para sincronizar hebras pthread_jon(tid,*res)

//EJEMPLO ES MUY IMPORTANTE

Condicion de Carrera: esto sucede cuando varios procesos acceden a los mismos datos de forma concurrente y el resultado de la ejecución depende del orden concreto en que se realicen los accesos.

Requisitos que debe cumplir cualquier solocuion a la sección critica.

- Exclusion mutua: solo un proceso puede tocar la variable
- Progreso: solo los que están esperando en la sección critica participan en la decisión.
- Espera limitada: un proceso no puede quedarse eternamente en la espera de la sección critica.

Semáforos

Variable entera a la solamente se accede mediante dos operaciones, wait y signal.

El valor del semáforo representa el numero de recursos disponibles

Wait: añade un proceso a la lista de bloqueados.

Signal: saca a un proceso de la lista.

Semáforos binarios o mutex: solo toman los valores 1 o 0

Semaforos contadores: valores negativos (numero ded coches que esperan) si es positivo son procesos que se pueden activar.

Para secciones criticas inicializarlo a 1

Para sincronizar inicializado a 0

- Problema de productor-consumidor
  - Semáforos compartidos por los dos procesos: ***seccion*** inicializ a valor 1, ***full*** inicializado a valor 0, ***empty*** inicializado a valor N

```
//PRODUCTOR

while (true)  {

  //produce un item

  wait (empty);
  wait (seccion);
  . . .
  //inserta el item
  . . .
  signal (seccion);
  signal (full);
}
```

```
//CONSUMIDOR

while (true) {

  wait (full);
  wait (seccion);
  . . .
  //extrae un item
  . . .
  signal (seccion);
  signal (empty);

  //consume el item
}
```

Semáforos - Implementación

SISTEMAS OPERATIVOS

- Los semáforos nos son parte del estándar de *Pthread*, pero se pueden utilizar conjuntamente.
- Para usarlos se debe incluir la directiva: *#include <semaphore.h>*
- **Tipo de datos "semáforo":** *sem_t*
- **Inicialización de un semáforo:** *sem_init(*sem, int shared, int valor)*
  **sem*: puntero a una variable de tipo *sem_t*
  shared: si se puede utilizar solo entre procesos ligeros creados dentro del que inicia el semáforo (0) o se puede heredar en el fork (!=0)
  valor: valor inicial
- **Destrucción de un semáforo:** *int sem_destroy (sem_t *sem);*
- Las operaciones **wait** y **signal** vistas se llaman:
  - *sem_wait(*sem)*: donde *sem* es un puntero a una variable *sem_t*
  - *sem_post(*sem)*: donde *sem* es un puntero a una variable *sem_t*

Mutex con condición: diapositiva 71 caso A. es un while porque si hay mas procesos y puede coger un proceso u otro el mutex tiene que comprobar si la condición de bloqueo y cae seguro

PLANIFICACION DE LA CPU

Planificador o scheduler: es el encargado de seleccionar cual de todos los procesos que están esperando será el siguiente a ejecutarse.

Dispatcher: es el encargado de poner un proceso a ejecutar

//Diapositiva 80 → el caso 2 un proceso es expulsado por durar demasiado, (bucles infinitos) o el 3 cuando un proceso tiene mas prioridad que otro.

Criterios de planificación:

Planificación de la CPU

- Distintos criterios de planificación:
  - **Maximizar la tasa de procesamiento.** Número de procesos que se completan por unidad de tiempo.
  - **Minimizar el tiempo de ejecución.** Tiempo que pasa desde que se ordena ejecutar un proceso hasta que termina su ejecución.
  - **Minimizar el tiempo de espera.** Tiempo que pasa esperando en la cola de procesos preparados.
  - **Minimizar el tiempo de respuesta.** Tiempo que pasa desde que se envía una solicitud hasta que se empiezan a recibir resultados.

- FCFS-> se asigna la CPU a los procesos en función de su orden de llegada, se implementa con una cola
- SJF -> se asigna la CPU al proceso que tiene un tiempo de ráfaga menor, en caso de empate se usa FCFS. Puede ser apropiativo o cooperativo. El apropiativo expulsa al proceso en cuanto llega uno que tiene menos duración. El tiempo medio de espera es el minimo tiempo de espera que se puede conseguir con cualquier algoritmo.