Sistemas Operativos

David Lopez

Contents

1 conceptos basicos

Un sistema operativo no es mas que un progama.

1.1 Funciones del s.o

Este programa, no obstante, es bastante caracteristico, pues se encarga de **gestionar los recursos hardware**, ofrece a los demas programas una api de **llamadas al sistema** y proporciona **mecanismos de interaccion con el usuario** como el shell.

Es necesario gestionar los recursos hardware por que podemos tener varios programas ejecutandose a la vez, y se hace mas importante con los sistemas multiusuario. Los programas competiran por los recursos, por lo que sera necesario administrarlos correctamente.

Sobre las llamadas al sistema habra algunos ejercicios para familiarizarnos, hay que tener en cuenta, que a diferencia de un curso de desarrollo sobre el kernel de un sistema operativo, aqui nos centraremos en programas de espacio de usuario y nos limitaremos, al menos por el momento a invocar a estas llamadas, no a implementarlas en incluirlas en el s.o , cosa que tambien se puede hacer.

1.2 partes principales del s.o

Podemos distinguir tres partes principales: **nucleo o kernel, servicios e interfaz de usuario**

1.3 clasificacion segun su estructura

- Monolitico (como linux): Tiene todos los componentes integrados en un mismo programa
- Por capas: cada capa ofrece una interfaz a la capa siguiente
- Cliente-Servidor (microkernel): el kernel se reduce a la minima expresion y el resto de utilidades se desarrollan en proceso de usuario

1.4 Como lo usamos ?

Podemos poner el sistema operativo a trabajar, deliberadamente y para que haga la funcion que queremos, escribiendo un programa que realize llamadas al sistema, las interrupciones de los perifericos y los errores tambien provocara la activacion del s.o

1.5 ejemplo de programa que invoca llamadas al sistema

escribir una funcion en C, que actue como el comando cat de unix, es decir, que imprima el contenido del archivo por la salida estandar

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

#define BUFFSIZE 512

void main(int arg, char *argv[]){

  int fd = 0;
  int cnt = 0;
  char buff[BUFFSIZE];

  fd = open(argv[1], O_RDONLY);

```

```
15    while((cnt = read(fd, buff, BUFFSIZE)) > 0){
16      write(1, buff, cnt);
17    }
18
19    close(fd);
20
21
22
23
24  }
```

En este codigo, las funciones *open, read, write y close* son llamadas al sistema, cuyo man se puede encontrar en: manual de llamadas al sistema, es fundamental saber utilizar esta herramienta y manejar con solvencia las llamadas al sistema mas habituales.

2 Procesos

2.1 Conceptos basicos

Un proceso es la unidad de procesamiento gestionada por el sistema operativo.

El sistema operativo mantiene una tabla de procesos. En esta tabla se almacena un **bloque de control de proceso** *(estructura task_struct en linux)* por cada proceso. El BCP contiene la siguiente informacion:

- Identifiacion: PID, USER y relaciones padre-hijo
- Estado del procesador
- Informacion de control del proceso
- Memoria asignada
- Recursos asignados

2.2 threads

Un proceso puede contener varios threads, o hilos de ejecucion que comparten algunos recursos e informacion entre ellos.

Cada thread se define como una funcion que se puede lanzar en paralelo con otras.

Los threads del mismo proceso comparten parte de su informacion: Memoria, variables globales, archivos abiertos, procesos hijos, temporizadores y semaforos. Que compartan la memoria genera que no haya proteccion de memoria y los threads son independientes unos de otros hasta cierto punto.

Pero no comparten: contador de programa, pila, registros y estado. En resumen, el contexto de ejecucion.

Las ventajas que proporciona este mecanismo de "dividir el trabajo de los procesos en tareas mas sencillas" es claro y sigue el paradigma de divide y venceras, sin embargo, cuando entramos en el terreno de la programacion concurrente hemos de ser cuidadosos por los tipicos problemas que pueden surgir.

2.3 senales

Se puede describir una senal como una interrupcion a un proceso. El comportamiento de las senales es similar al de las interrupciones en la cpu.

Las senales pueden provenir o bien de un proceso que la manda a otro proceso con su mismo *uid* o el sistema operativo ante las tipicas excepciones que vemos cuando fallan nuestros programas (seg fault, /0...)

Hay muchas senales diferentes y se engloban en tres tipos fundamentales: De hardware, de comunicacion y de E/S asincrona.

Para que un proceso pueda enviar una senal ha de estar preparado para recibirla. Para ello tenemos que indicarle al SO el nombre de la rutina del proceso que ha de tratar ese tipo de senal. Si un proceso recibe una senal sin estar preparado para ello, se suele matar al proceso.

2.4 Servicios de procesos

Gestion de procesos:

- Identificador de proceso: PID (pid_t)
- Obtener el identificador de proceso: getpid() devuelve el identificador del proceso que hizo la llamada
- Obtener el identificador del proceso padre: getppid()

- Obtener el identificador de usuario real: getuid()
- Obtener el identificador de usuario efectivo: geteuid()
- Obtener el identificador de grupo real: getgid()
- Obtener el identificador de grupo efectivo: getegid()
- Obtener el valor de una variable de entorno: *getenv(const char * name)
- Fijar el entorno de proceso: *setenv(char **env)
- Crear un proceso: fork(). Clona al proceso que lo creo, pasando a ser su hijo
- Ejecutar un programa: familia exec. Cambia el programa que esta ejecutando un proceso
- Terminar la ejecucion de un proceso: exit(int status). Similar a los return
- Esperar a la finalizacion de un proceso hijo: wait(int *status) o wait_pid(pid_t , int * status, int options)
- Suspender la ejecucion de un proceso: sleep(unsigned int seconds)

Gestion de threads:

- Atributos de un thread: (cada thread tiene asociado unos atributos, y los valores de estos se almacenan en un objeto de tipo *pthread_attr_t*
  - Crear atributos de un thread: int pthread_attr_init(pthread_attr_t *attr);
  - Destruir atributos: int pthread_attr_destroy(pthread_attr_t *attr);
  - Asignar tamano al thread: int pthread_attr_setstacksize(pthread_attr_t *attr);
  - Obtener el tamano de la pila: int pthread_attr_getstacksize(pthread_attr_t *attr)

  - Establecer el estado de terminacion: int pthread_attr_setdetachstate(pthread_attr_t * attr, int detachstate). El valor del segundo argumento, indicara la independencia o no del thread con respecto a otros.
  - Obtener el estado de terminacion: int pthread_attr_setdetacstate(pthread_attr_t * attr, int * detachstate)

- Creacion, identificaciony terminacion de threads:
  - Crear un thread: int pthread_create(pthread_t *thread, pthread_attr_r *attr, void * (*start_routine) (void *), void * arg). El primer argumento es el indentificador, el segundo atributos de ejecucion, el tercero el nombre de la funcion a ejecutar cuando se lance y el ultimo es un puntero a void.
  - Obtener el identificador de un thread: pthread_t pthread_self(void)
  - Esperar la terminacion de un thread: int pthread_join(pthread thid, void **value). Es necesario indicar el thread por el que se quiere esperar.
  - Finalizar la ejecucion de un thread: int pthread_exit(void * value)