Sistemas Operativos

Ezequiel Postan

Abstract

Apuntes de la materia Sistemas Operativos de LCC-FCIEA. Curso dictado en 2011. El apunte está basado en la carpeta Emilio Lopez con agregados y correcciones. Quedan correcciones pendientes, particularmente revisar vocabulario y ortografía.

Docentes

- Guido Macchi
- Esteban Ruiz
- Federico Bergero
- Guillermo Grimgblat

[["The mythical Man-moth" F.brooks (Abstract: Habla de los problemas que enfrento al programar OS360)]]

Bibliografia

- Modern Operating Systems - Tanenbaum
- Operating Systems Concepts - Silberschatz & Galvin
- Operating Systems - Stallings
- Operating Systems - Milenkovic
- The Design and Implementation of the BSD 4.4 Operating System - Mckusik et al.
- The Design of the UNIX Operating System - Bach

Contents

1 Historia

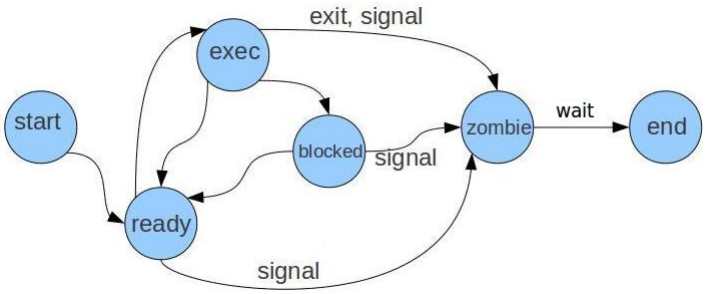
- 1955 Monitores
- 1963 Sistema 360 con OS360 con maquina virtual
- 1965 Proteycto MULTICS (MULTiplexed Information Control System)
- 1967 Dijkstra: The OS. Usa un modelo de capas para estructurarse
- 1969 AT&T se retira de MULTICS. Ken Thompson vuelve a Bell Labs. Hace un kernel para un PDP/7, bautizado como UNICS (UNiplexed Information Control System)
- 1970 D. Ritchie. Toma BCPL de Richardson y crea B. Ritchie lo agranda y crea C. Con esto, portan UNICS en una minicomputadora PDP/11
- 1974 Se renombra como UNIX y se difunde a Berkeley
- 1975 BSD01. Es elegido por el DoD (Department of Defense) para implementar TCP/IP
- 1977 BSD con paginacion
- 1982 BSD3 con TCP/IP
- 1990 BSD386. Demanda de AT&T (algo asi como por invasión de derechos intelectuales)
- 1994 BSD 4.4. Muere CSRG. Sale Linux 1.0.

2 Introducción

2.1 Definiciones

- Nucleo (kernel): proceso que se ejecuta en modo supervisor
- Programa: un archivo con valores iniciales para Text y Data
- Proceso: una sucesión de estados de sistemas.
  - Su memoria se compone de tres regiones:
    * Text Segment (instrucciones),
    * Data Segment (datos)
    * System Segment.

- Estados de un proceso:



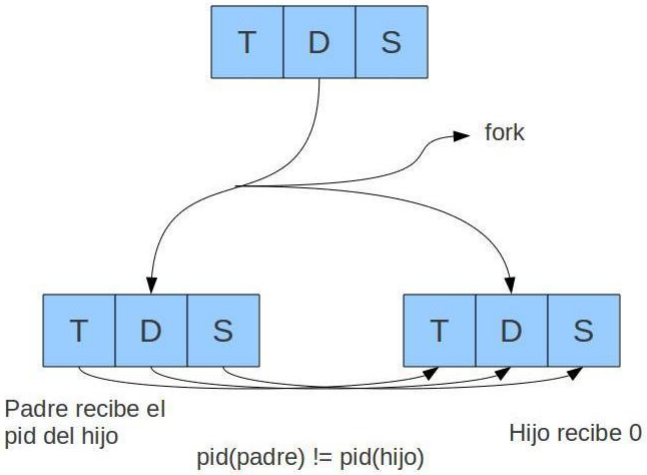
  - start que arranca el proceso,
  - ready que es cuando tengo los recursos disponibles para el proceso,
  - exec que es cuando se está ejecutando,
  - blocked cuando esta esperando que le larguen los recursos,
  - zombie que es cuando esta esperando para poder terminar (el caso en el que el padre espera a que termine el hijo)
  - end.

2.2 Creación/fin de procesos.

Tomaremos como ejemplo a UNIX.

Tenemos las primitivas:

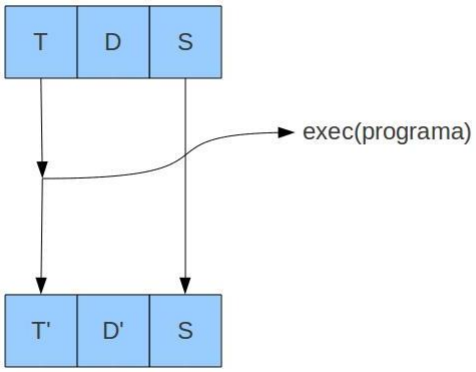
- **fork:** clona un proceso. int fork()




```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("vamos!\n");       // Si no vacio el buffer (printf("vamos!");) sin el \n
    if(fork())                // entonces fork me realiza el proceso para padre e hijo

       printf("padre!\n");    // con printf("vamos!"); en el buffer entonces mi salida es
    else                      // vamos!padre!
```

```
        printf("hijo\n");      // vamos!hijo!
    return 0;
}
```

- **exec:** es una familia de programas donde cada uno de ellos crea un proceso a partir de un programa (T,D,S)—>exec(programa)—>(T',D',S)



  La familia se compone

  - int execl(char *prog, char *arg0, char *arg1,...,NULL); //NOTA: arg0 es el programa mismo
  - int execv(char *prog, char **args); //**args termina con un NULL para indicar que ya no tengo mas argumentos

  Ejemplo

```
/* ls -l --sort=time */
#include <stdio.h>
#include <unistd.h>
static char *args[]={
    "ls",
    "-l",
    "--sort=time",
    NULL
};
int main()
{

    execv("/bin/ls",args);        // con execl seria execl("/bin/ls","ls","-l","--sort=time",NULL)
    perror("no esta en el bin!"); // si la ejecucion del programa llega a este punto es porque
                                  // no salio bien
    return 0;                     // execv. Sino execv habria reemplazado estas lineas con su
                                  // programa
}
```

- **exit:** termina un proceso (un exit(N) en main equivale a un return N)

```
int exit(int n)
```

- **wait:** es una primitiva que permite al padre esperar a que un hijo termine (normal o anormalmente).

```
int wait(int *)
```

  - En el entero retorna el PID del hijo
  - En el puntero devuelve el status de terminacion del hijo

El status es una mezcla del valor de retorno del hijo más información extra. Esta información se maneja con macros definidas en sys/wait.h (ver man 2 wait)

Incluye:

- WIFEXITED(status): 1 si termino con exit, 0 si no
- WEXITSTATUS(status): el valor de exit (si la anterior dio true).
  Convención: un proceso que anda bien devuelve cero, <> 0 en caso contrario

Ejemplo:

Un esqueleto de un shell.

Su ciclo vital es:

1. Leer una linea con el nombre del programa
2. Ejecutarlo
3. Volver al principio

1° intento (erroneo)

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main()
{
    char linea[1024];
    for(;;){
       printf("prompt> ");
       fgets(linea, sizeof linea,stdin);
       // stdin: Standard Input...normally keyboard
       linea[strlen(linea)-1]=0;     // elimino el \n al final de la linea
       execl(linea,linea,NULL);
    }
    return 0;
}
```

Tiene un error! execl si anda bien REEMPLAZA este proceso por el otro. Tendriamos que tomar una copia de shell

2° intento.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
int main()
{
    char linea[1024];
    for(;;){
       printf("prompt> ");
       fgets(linea, sizeof linea,stdin);
       linea[strlen(linea)-1]=0;
       if(!fork()){ /* hijo */
         execl(linea,linea,NULL);
         perror(linea);  // esta linea y la proxima se pueden sintetizar como
         exit(-1);           // exit((perror(linea),-1))
```

```
        }else{ /* padre */
          int status;
          wait(&status);
          // siempre y cuando un padre solo puede tener 1 hijo, verdad!?
          if(WIFEXITED(status))
            printf("El programa termino bien.\n");
        }
    }
    return 0;
}
```

wait es un ejemplo de sincronización. El proceso padre queda detenido hasta que el proceso hijo haya terminado. En UNIX este tipo de cosas se conocen como IPC (Inter Process Communications)

2.3 Pipes

Tratemos de comunicar dos procesos entre si.

Por comunicación queremos significar el intercambio de datos entre espacios de direcciones distintos.

address space 1 <—–> address space 2

Posibles soluciones

Proc1	Proc2
printf(...)	fgets(...)

Otra forma: Crear un archivo en memoria rígida donde uno de los procesos erscribe con write y el otro lo lee con read ambos pasando por el Sistema Operativo. El problema es que los procesos en disco corren un poco más lento que en la memoria.

Sería bueno que primero el Sistema Operativo en vez de hacer las cosas directamente sobre el disco (lo cual haría que tarde bastante) realice las acciones en un cache: el proceso que escribe recibe la orden de que está todo bien y el Sistema Operativo lo pasa a disco cuando lo cree conveniente. El read antes de leer desde disco se fija primero si lo que busca está en cache ahorrandome mucho timepo. A esto se lo llama pipe.

Un pipe es un buffer del kernel, que permite ser leído y escrito. Se crea con la primitiva pipe.

Prototipo:

```
int pipe(int a[2]);
```

donde tengo algo asi como que un número para leer a[0] y un handler para escribir a[1].

Ejemplo

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    int pip[2], cto;
    char l[1024];
    pipe(pip);
    write(pip[1], "hola mundo\n",11);
    cto=read(pip[0],l,1024);
    l[cto]=0;
    printf("%s",l);
    close(pip[0]);
    close(pip[1]);
    return 0;
```

```
}
```

Hagamos esto entre 2 procesos.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main()
{
    int pip[2];
    pipe(pip);
    if(fork()){ /* padre */
        write(pip[1],"hola mundo",10);
        wait(NULL);
    } else{ /* hijo */
        char l[1024];
        int cto;
        cto=read(pip[0],l,1024);
        l[cto]=0;
        printf("%s\n",l);
    }
    close(pip[0]);
    close(pip[1]);
    return 0;
}
```

"Delicias de las IPC"

Veamos un pequeña variación del programa anterior

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main()
{
    int pip[2];
    pipe(pip);
    if(fork()){ /* padre */
        write(pip[1],"hola mundo",10);
        wait(NULL);
    } else{ /* hijo */
        char c;
        while(read(pip[0],&c,1)==1){
           putchar(c);
           fflush(stdout);
        }
    }
    close(pip[0]);
    close(pip[1]);
    return 0;
}
```

Al ejecutarlo aparece
    hola mundo

y los procesos no terminan nunca.

El problema es que read es una llamada bloqueante, esto quiere decir que se va a quedar bloqueada hasta que haya algo nuevo para leer o que ya no haya mas posibilidades para leer (entradas abertas). Mientras haya un canal de comunicación abierto para poder leer va a quedar bloquedo esperando.

Pordemos arreglar el problema de la siguiente manera

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {

    int pip[2];
    pipe(pip);
    if(fork()){ /* padre */
        write(pip[1],"hola mundo\n",11);
        close(pip[1]); /* Che no te voy a escribir mas */
        wait(NULL);
    } else{ /* hijo */
        char c;
        close(pip[1]); /* Eu eso que abriste antes de hacer el fork cerralo */
        while(read(pip[0],&c,1)==1){
           putchar(c);
           fflush(stdout);
        }
    }
    close(pip[0]);
    close(pip[1]);  //por qué este close?
    return 0;

}
```

Listo ahora aparece
    hola mundo

y los procesos terminan.

Problema en el siguiente programa, conseguir que Hola mundo vaya al archivo KKK SIN TOCAR la zona de exclusión.

```
#include <unistd.h>
/* begin zona de exclusion */
void f() {

    write(1,"Hola mundo",10);

}
/* end zona de exclusion */
int main()
{

    f();
    return 0;

}
```

La primitiva **dup**

Prototipo **int dup(int handler)**

Los procesos continen una tabla de file descriptors que indca que archivos está manejando (por ejemplo stdin, stdout y otros archivos que utilice el proceso), a cada file descrptor se le asigna un handler (un entero) para identificarlo.

El entero que retorna dup representa el lugar del duplicado en la tabla de descriptores. Lo que hace es copiar el descriptor que se identifica con "handler" en la tabla de descriptores asignando otro handler más. O sea que duplica el el file descriptor.

Se garantiza que el duplicado queda en el primer lugar libre de la tabla.

Con esto, agreguemos al programa:

```
#include <unistd.h>
#include <fcntl.h>
/* begin zona de exclusion */
void f()
{
write(1,"Hola mundo",10);
}
/* end zona de exclusion */
/* CAMBIA MAIN */
int main()
{

    int arch;
    /*1*/ arch=open("KKK",O_WRONLY|O_CREAT|O_TRUNC,0666);
    /*2*/ close(1);
    /*3*/ dup(arch);
    /*4*/ close(arch);
    f();
    return 0;

}
```

Veamos como se modifica la tabla de descriptores en cada paso

/*1*/

File Descriptor	File
⋮	⋮
1	stdout
⋮	⋮
arch	KKK

/*2*/

File Descriptor	File
⋮	⋮
1	-
⋮	⋮
arch	KKK

/*3*/

File Descriptor	File
⋮	⋮
1	KKK
⋮	⋮
arch	KKK

/*4*/

File Descriptor	File
⋮	⋮
1	KKK
⋮	⋮
arch	-

Otro problema: ordenar una serie de palabras.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
char *a[]={
    "esta","es",
    "una","p...",
    "prueba",
    NULL
};
// Podríamos implementar algunos de los sort que conocemos pero ya el Linux viene con un sort.
int main() {
    int pip[2];
    int i, cto;
    char L[1024];
    pipe(pip);
    if(fork()){ /* padre */
        for(i=0;a[i]!=NULL;i++){
            write(pip[1],a[i],strlen(a[i]));
            write(pip[1],"\n",1);
        }
        while((cto=read(pip[0],L,1024))>0){
            L[cto]=0;
            printf("%s",L);
        }
        // no falta un wait(NULL); acá?
    } else{

        close(0); dup(pip[0]); close(pip[0]); // 0 es el stdin (cambialo por pip(0))
        close(1); dup(pip[1]); close(pip[1]); // 1 es el stdout (cambialo por pip(1))
        execl("/usr/bin/sort","sort",NULL);
        exit(-1);
    }
    return 0;
}
```

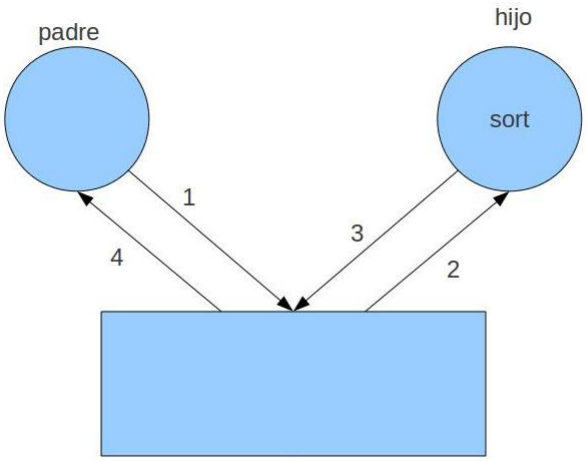
Esto anda bien el 5 por mil de las veces, el resto de las veces mostrara las palabras como estan en el arreglo y no termina. Es un problema de sincronizacion.

La secuencia correcta es:

1. El padre escribe
2. El hijo lee y luego escribe

3. Luego el padre lee.

Si la secuencia es asi entonces anda bien.

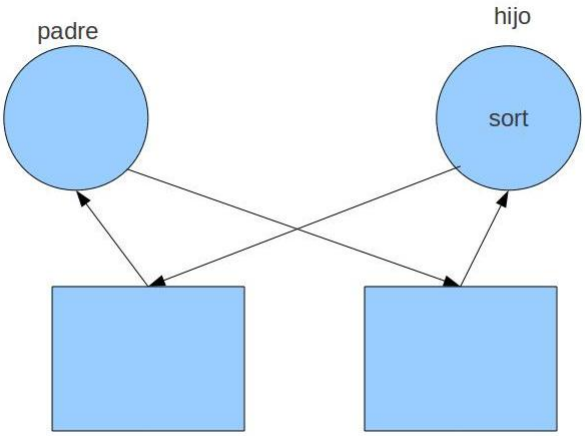


Ahora la mayoría de las veces pasara esto:

El padre va a escribir con el for y va a leer (lo que acaba de escribir) con el while. El hijo se queda esperando a leer en la nada y el padre se queda esperando a que termine el sort...que nunca sucede.

Que pasaría si uso más de un pipe? Que de el pipe de la izquierda solo lea el padre y solo puede escribir el hijo. En el otro solo puede escribir el padre y solo puede leer el hijo!!

Me aseguro el flujo correcto!



La solución al problema queda asi:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
char *a[]={  "esta","es",  "una","p...",  "prueba",  NULL };
// Podriamos implementar algunos de los sort que conocemos pero ya el Linux viene con un sort.
int main() {

    int ph[2], hp[2], cto;
    pipe(ph);
    pipe(hp);
    if(fork()){ /* padre */
```

```
            int i;
            char linea[1024];
            close(ph[0]); // cierro lo que no voy a usar
            close(hp[1]);
            for(i=0;a[i]!=NULL;i++){
               write(ph[1],a[i],strlen(a[i]));
               write(ph[1],"\n",1);
            }
            close(ph[1]);
            while((cto=read(hp[0],linea,sizeof linea))>0){
               linea[cto]=0;
               printf("%s",linea);
            }
            wait(NULL);
        }else{
            close(ph[1]);
            close(hp[0]);
            close(0); dup(ph[0]); close(ph[0]);
            close(1); dup(hp[1]); close(hp[1]);
            execl("/usr/bin/sort","sort",NULL);
            perror("!!!");
            exit(-1);
        }
        return 0;
    }
```

2.4 Señales

Un evento se dice síncrono si el proceso actúa esperando su aparición. Caso contrario se denomina asíncrono. Estos últimos también se conocen como excepciones y pueden ser errores o eventos anormales.

UNIX brinda a un proceso la posibilidad de tratar la aparición de estos eventos mediante un mecanismo conocido como señales. Este mecanismo tiene dos implementaciones la de AT&T y la de BSD. Veamos la primera.

En signal.h están definidos distintos eventos

Nombre	Evento	Acción por Defecto
SIGINT	Ctrl+C	Termina
SIGQUIT	Ctrl+\	Termina
SIGABRT	abort	Aborta
SIGFPE	floating-point	Aborta
SIGVBUS	violacion de segmento	Aborta
SIGIO	cambio de condicion en I/O	Ignorada
SIGCHLD	fin de hijo	Ignorada
SIGSTOP	Ctrl+Z	Freezado*
SIGCONT	-	Despierta un frizado
SIGKILL	-	Termina*
*no pueden ser capturadas		

Luego define una función que permite que, si aparece un evento particular de los anteriores sea invocada una función. Esta funcion se llama signal y tiene el siguiente prototipo:

```
void (*signal (int sig, void (*func) (int))) (int);
```

Si usamos typedef podemos hacer

```
typedef void (*pfiv) (int)
```

y escribir

```
pfiv signal(int sigh, pfiv pf);)
```

El puntero devuelto correspode a la función anterior que hacía.
(signal() returns the previous value of the signal handler, or SIG_ERR on error.)

Hay dos valores distinguidos que pueden ir en el handler:

- SIG_DLF (realiza la acción por default)
- SIG_IGN (ignora la señal)
- La otra opción es la función que uno define y le pasa

Ejemplo: Programa que aguanta tres Ctrl+c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void handler(int sig)
{

    static int cuantas=0; /* al pedo las variables estaticas siempre son 0 por default */
    if(sig==SIGINT && ++cuantas<4)
        printf("Ja!\n");
    else{
        printf("Adios!\n");
        exit(0);
    }
}
int main()
{
    signal(SIGINT,handler);
    for(;;)
        ;
    return 0;
}
```

3 TCP/IP

Detalles de los pipes

Son un buen medio para comunicar dos procesos (por supuesto con las precauciones correspodientes), pero no sirve para comunicar dos procesos arbitrarios: deben tener un ancestro común que haya creado el pipe. Para esto existen FIFOS, queues, named pipes, shared memory, etc. De todas estos veremos dos protocolos: LOCAL (o UNIX) y TCP/IP.

3.1 Historia de TCP/IP.

- 1965 Nace la idea de transmision por paquetes

  Tengo sesiones que deseo transmitir. Teniendo 3 por ejemplo A B y C se dividen en paquetes pequeños con una cabecera que los identifique (se mandaban en parte porque no alcanzaba el ancho de banda) y luego se rearman en el destino.

- 1966 Se prueba con una transmision entre el MIT y la UCLA.

  Tenemos algo que es bueno y anda! Ahora para darle un poco mas de maquina se pide ayuda (economica mas que nada) al Department of Defense. Este lo incluye en ARPA/DARPA (Defense Advanced Research Project Agency)

- 1973 Stanford comisiona a Bolt, Beranek y Newman para que redacten un portocolo.
- 1974 BBN redacta el protocolo TCP/IP
- 1975 Se encarga la implementacion a la UCB (Berkeley)
- 1982 Primera implementacion en BSD 2.0. Nace ARPANET
- 1990 ARPANET sale de la orbita del Department of Defense y se renombra como Internet.

3.2 Estructura

Veamos como se estructura TCP/IP. Adapta la estructuracion por capas!

User	Proceso	
	↕socket	
Kernel	TCP | UDP	Capa de transporte
	Capa de Red	IP: Internet Protocol
	Capa de Enlace	Drivers
	Capa Física	Hardware

- La capa de transporte se divide en dos subcapas
  - TCP: Transmission Control Protocol
    - Es 'caro' mantiene siempre la conexion
    - Orientado a sesión. Aloja recursos aún cuando no se este enviando información
    - Preserva el orden de envío de paquetes
    - Evita duplicados
    - Trata de corregir errores
  - UDP: Universal Datagram Protocol
    - Es obviamente mas barato en recursos que el otro.
    - Orientado a paquetes
    - No hace ningun intento de corregir errores
- Las capas fisica, de enlace, de red y de transporte estan en el kernel, no en el usuario.
- Los programas para usar TCP/IP corresponden al modelo CLIENT/SERVER.
  - Cliente es el que inicia la conexion.
  - Servidor es el que espera la conexion.

En el espacio de direcciones (que identifican a cada unidad de manera única) de TCP/IP las direcciones se forman con dos numeros (IPv4):

- Port (16 bits) /* identifica un proceso en un host */
- Address (32 bits) /* identifica el host y la red en la que esta */

Esta última necesita codificar dos cosas: por un lado necesita codificar una red local (LAN) y una maquina (host) dentro de la red local.

Se parte Address en categorias

Categoría	Primer Byte	LAN	Host
A	1-126	8 bits	24 bits
B	128-191	16 bits	16 bits
C	193-224	24 bits	8 bits
D	225-240	multicast	
E	241-252	experimental	

Tipicamente las IP Address se escriben asi ej:

200 . 3 . 123 . 97

1er byte . 2do . 3ro . 4to

Nota: en todo host la direccion 127.x.x.x denota a ese host

3.3 Agregado para entender...

Fzuente muy instructiva (y recomendable): http://www.chuidiang.com/clinux/sockets/sockets_simp.php

3.3.1 Qués es un socket

- Una forma de conseguir que dos programas se transmitan datos, basada en el protocolo TCP/IP, es la programación de sockets.
- Un socket no es más que un "canal de comunicación" entre dos programas que corren sobre ordenadores distintos o incluso en el mismo ordenador.
- Desde el punto de vista de programación, un socket no es más que un "fichero" que se abre de una manera especial.
- Existen básicamente dos tipos de "canales de comunicación" o sockets:
  - **Orientados a conexión:** ambos programas deben conectarse entre ellos con un socket y hasta que no esté establecida correctamente la conexión, ninguno de los dos puede transmitir datos. Esta es la parte TCP del protocolo TCP/IP, y garantiza que todos los datos van a llegar de un programa al otro correctamente. Se utiliza cuando la información a transmitir es importante, no se puede perder ningún dato y no importa que los programas se queden "bloqueados" esperando o transmitiendo datos. Si uno de los programas está atareado en otra cosa y no atiende la comunicación, el otro quedará bloqueado hasta que el primero lea o escriba los datos.
  - **No orientados a conexión:** no es necesario que los programas se conecten. Cualquiera de ellos puede transmitir datos en cualquier momento, independientemente de que el otro programa esté "escuchando" o no. Es el llamado protocolo UDP, y garantiza que los datos que lleguen son correctos, pero no garantiza que lleguen todos. Se utiliza cuando es muy importante que el programa no se quede bloqueado y no importa que se pierdan datos.

3.3.2 Arquitectura Cliente/Servidor

Uno de los programas debe estar arrancado y en espera de que otro quiera conectarse a él. Nunca da "el primer paso" en la conexión. Al programa que actúa de esta forma se le conoce como servidor.

El servidor es el programa que permanece pasivo a la espera de que alguien solicite conexión con él, normalmente, para pedirle algún dato. El cliente es el programa que solicita la conexión para, normalmente, pedir datos al servidor.

3.3.3 La conexión

Para poder realizar la conexión entre ambos programas son necesarias varias cosas:

- Dirección IP del servidor:
  - Cada ordenador de una red tiene asignado un número único, que sirve para identificarle y distinguirle de los demás, de forma que cuando un ordenador quiere hablar con otro, manda la información a dicho número.
  - El servidor no necesita la dirección de ninguno de los dos ordenadores, al igual que nosotros, para recibir una llamada por teléfono, no necesitamos saber el número de nadie, ni siquiera el nuestro.
  - El cliente sí necesita saber el número del servidor, al igual que nosotros para llamar a alguien necesitamos saber su número de teléfono.
- Servicio (puerto) que queremos crear / utilizar.
  - En un mismo ordenador pueden estar corriendo varios programas servidores, cada uno de ellos dando un servicio distinto.
  - Cuando un cliente desea conectarse, debe indicar qué servicio quiere.
  - Por ello, cada servicio dentro del ordenador debe tener un número único que lo identifique (como la extensión de teléfono). Estos números son enteros normales y van de 1 a 65535. Los número bajos, desde 1 a 1023 están reservados para servicios habituales de los sistemas operativos (www, ftp, mail, ping, etc). El resto están a disposición del programador.
  - Tanto el servidor como el cliente deben conocer el número del servicio al que atienden o se conectan. El servidor le indica al sistema operativo qué servicio quiere atender.

3.3.4 El servidor

Con C en Unix/Linux, los pasos que debe seguir un programa servidor son los siguientes:

- Apertura de un socket, mediante la función socket(). Esta función devuelve un descriptor de fichero normal, como puede devolverlo open(). La función socket() no hace absolutamente nada, salvo devolvernos y preparar un descriptor de fichero que el sistema posteriormente asociará a una conexión en red.
- Avisar al sistema operativo de que hemos abierto un socket y queremos que asocie nuestro programa a dicho socket. Se consigue mediante la función bind(). El sistema todavía no atenderá a las conexiones de clientes, simplemente anota que cuando empiece a hacerlo, tendrá que avisarnos a nosotros. Es en esta llamada cuando se debe indicar el número de servicio al que se quiere atender.
- Avisar al sistema de que comience a atender dicha conexión de red. Se consigue mediante la función listen(). A partir de este momento el sistema operativo anotará la conexión de cualquier cliente para pasárnosla cuando se lo pidamos. Si llegan clientes más rápido de lo que somos capaces de atenderlos, el sistema operativo hace una "cola" con ellos y nos los irá pasando según vayamos pidiéndolo.
- Pedir y aceptar las conexiones de clientes al sistema operativo. Para ello hacemos una llamada a la función accept(). Esta función le indica al sistema operativo que nos dé al siguiente cliente de la cola. Si no hay clientes se quedará bloqueada hasta que algún cliente se conecte.
- Escribir y recibir datos del cliente, por medio de las funciones write() y read(), que son exactamente las mismas que usamos para escribir o leer de un fichero. Obviamente, tanto cliente como servidor deben saber qué datos esperan recibir, qué datos deben enviar y en qué formato.
- Cierre de la comunicación y del socket, por medio de la función close(), que es la misma que sirve para cerrar un fichero.

3.3.5 El cliente

Los pasos que debe seguir un programa cliente son los siguientes:

- Apertura de un socket, como el servidor, por medio de la función socket()
- Solicitar conexión con el servidor por medio de la función connect(). Dicha función quedará bloqueada hasta que el servidor acepte nuestra conexión o bien si no hay servidor en el sitio indicado, saldrá dando un error. En esta llamada se debe facilitar la dirección IP del servidor y el número de servicio que se desea.
- Escribir y recibir datos del servidor por medio de las funciones write() y read().
- Cerrar la comunicación por medio de close().

3.4 Las funciones y estructuras a usar

3.4.1 socket (en cliente y servidor tanto TCP como UDP)

El socket se abre mediante la llamada a la función socket() y devuelve un entero que es el descriptor de fichero o –1 si ha habido algún error.

```
int socket(int domain, int type, int protocol);
```

- El primer parámetro es AF_INET o AF_UNIX para indicar si los clientes pueden estar en otros ordenadores distintos del servidor o van a correr en el mismo ordenador.
  - AF_INET vale para los dos casos.
  - AF_UNIX sólo para el caso de que el cliente corra en el mismo ordenador que el servidor, pero lo implementa de forma más eficiente. Si ponemos AF_UNIX, el resto de las funciones varía ligeramente.
- El segundo parámetro indica si el socket es orientado a conexión (SOCK_STREAM) o no lo es (SOCK_DGRAM).
- El tercer parámetro indica el protocolo a emplear. Habitualmente se pone 0.

```
Nota útil: In some documentation, you'll see mention of a mystical "PF_INET". This is a weird etherial b
```

3.4.2 bind (tanto en server TCP como UDP)

Si se ha obtenido un descriptor correcto, se puede indicar al sistema que ya lo tenemos abierto y que vamos a atender a ese servicio. Para ello utilizamos la función bind(). El problema de la función bind() es que lleva un parámetro bastante complejo que debemos rellenar.

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

El parámetro que necesita es una estructura sockaddr. Lleva varios campos, entre los que es obligatorio rellenar los indicados a continuación:

- sin_family es el tipo de conexión (por red o interna), igual que el primer parámetro de socket().
- sin_port es el puerto (número correspondiente al servicio que obtuvimos con getservbyname(). El valor está en el campo s_port de Puerto).
- sin_addr.s_addr
  - En el servidor: es la dirección del cliente al que queremos atender. Colocando en ese campo el valor INADDR_ANY, atenderemos a cualquier cliente.
  - En el cliente: es la dirección IP del servidor

La llamada a bind() lleva tres parámetros:

- Descriptor del socket que hemos abierto
- Puntero a la estructura sockaddr con los datos indicados anteriormente. La estructura admitida por este parámetro es general y valida para cualquier tipo de socket y es del tipo struct sockaddr. Cada socket concreto lleva su propia estructura. Los AF_INET como este caso llevan struct sockaddr_in, los AF_UNIX llevan la estructura struct sockaddr_un. Por eso, a la hora de pasar el parámetro, debemos hacer un "cast" al tipo struct sockaddr.
- Longitud de la estructura addr (sockaddr_in o sockaddr_un).

La función devuelve -1 en caso de error.

3.4.3 listen (en el server TCP)

Tras el bind, podemos decir al sistema que empiece a atender las llamadas de los clientes por medio de la función listen()

```
int listen(int sockfd, int backlog);
```

La función listen() admite dos parámetros:

- Descriptor del socket.
- Número máximo de clientes encolados.
  - Supongamos que recibimos la conexión de un primer cliente y le atendemos. Mientras lo estamos haciendo, se conecta un segundo cliente, al que no atendemos puesto que estamos ejecutando el código necesario para atender al primero. Mientras sucede todo esto, llega un tercer cliente que también se conecta. Estamos atendiendo al primero y tenemos dos en la "lista de espera". El segundo parámetro de listen() indica cuántos clientes máximo podemos tener en la lista de espera. Cuando un cliente entra en la lista de espera, su llamada a connect() queda bloqueada hasta que se le atiende. Si la lista de espera está llena, el nuevo cliente que llama a connect() recibirá un error de dicha función. Algo así como un "vuelva usted mañana".

La función listen() devuelve –1 en caso de error.

3.4.4 accept (en el servidor TCP)

Con todo esto ya sólo queda recoger los clientes de la lista de espera por medio de la función accept(). Si no hay ningún cliente, la llamada quedará bloqueada hasta que lo haya. Esta función devuelve un descriptor de fichero que es el que se tiene que usar para "hablar" con el cliente. El descriptor anterior corresponde al servicio y sólo sirve para encolar a los clientes. Digamos que el primer descriptor es el aparato de teléfono de la telefonista de la empresa y el segundo descriptor es el aparato de teléfono del que está atendiendo al cliente.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

La función accept() es otra que lleva un parámetro complejo, pero que no debemos rellenar. Los parámetros que admite son

- Descriptor del socket abierto en el servidor.
- Puntero a estructura sockaddr. A la vuelta de la función, esta estructura contendrá la dirección y demás datos del ordenador cliente que se ha conectado a nosotros.
- Puntero a un entero, en el que se nos devolverá la longitud útil del campo anterior.

La función devuelve el descriptor del socket aceptado o un –1 en caso de error.

3.4.5 connect (en el cliente TCP)

Envía la petición de conectarse al servidor

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Los parámetros son análogos a accept()

- Descriptor del socket abierto en el cliente.
- Puntero a estructura sockaddr completada con con los datos del servidor.
- Longitud de la estructura addr (sockaddr_in o sockaddr_un).

La función devuelve 0 en caso exitoso o –1 en caso de error.

3.4.6 Envío de mensajes

La diferencia principal con los TCP (orientados a conexión), es que en estos últimos ambos sockets (de cliente y de servidor) están conectados y lo que escribimos en un lado, sale por el otro. En un UDP los sockets no están conectados, así que a la hora de enviar un mensaje, hay que indicar quién es el destinatario.

En los sockets orientados a conexión se envían mensajes con write() o send() y se reciben con read() o recv(). En un socket no orientado a conexión hay que indicar el destinatario, así que se usan las funciones sendto() y recvfrom().

read/write (en TCP) Si todo ha sido correcto, ya podemos "hablar" con el cliente. Para ello se utilizan las funciones read() y write() de forma similar a como se haría con un fichero.

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

recvfrom (en UDP) La función para leer un mensaje por un socket upd es recvfrom().

```
ssize_t recvfrom(

    int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen
);
```

Esta función admite seis parámetros:

- Primero el descriptor del socket que queremos leer. Lo obtuvimos con socket().
- Segundo el buffer donde queremos que nos devuelva el mensaje. Podemos pasar cualquier estructura o array que tenga el tamaño suficiente en bytes para contener el mensaje. Debemos pasar un puntero y hacer el cast a char *.
- Luego el número de bytes que queremos leer y que compondrán el mensaje. El buffer pasado en el campo anterior debe tener al menos tantos bytes como indiquemos aquí.
- El cuarto son opciones de recepción. De momento nos vale un 0.
- El quinto es un struct sockaddr otra vez. Esta vez tenemos suerte y no tenemos que rellenarla. La pasaremos vacía y recvfrom() nos devolverá en ella los datos del que nos ha enviado el mensaje. Si nos los guardamos, luego podremos responderle con otro mensaje. Si no queremos responder, en este parámetro podemos pasar NULL. Ojo, si lo hacemos así, no tenemos forma de saber quién nos ha enviado el mensaje ni de responderle.
- Por último el puntero a un entero. En él debemos poner el tamaño de la estructura sockaddr que pasamos. La función nos lo devolverá con el tamaño de los datos contenidos en dicha estructura.

sendto (en UDP) La función para envío de mensajes es sendto().

```
ssize_t sendto(

    int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addr
);
```

Esta función admite seis parámetros, que son los mismos que la función recvfrom(). Su signifcado cambia un poco, así que vamos a verlos:

- int con el descriptor del socket por el que queremos enviar el mensaje. Lo obtuvimos con socket().
- char * con el buffer de datos que queremos enviar. En este caso, al llamar a sendto() ya debe estar relleno con los datos a enviar.
- int con el tamaño del mensaje anterior, en bytes.
- int con opciones. De momento nos vale poner un 0.
- struct sockaddr. Esta vez sí tiene que estar relleno, pero seguimos teniendo suerte, nos lo rellenó la función recvfrom(). Poniendo aquí la misma estructura que nos rellenó la función recvfrom(), estaremos enviando el mensaje al cliente que nos lo envío a nosotros previamente.
- int con el tamaño de la estructura sockaddr. Vale el mismo entero que nos devolvió la función recvfrom() como sexto parámetro.

3.4.7 close

Una vez que se han leído / enviado todos los datos necesarios, se procede a cerrar el socket. Para ello se llama a la función close(), que admite como parámetro el descriptor del socket que se quiere cerrar.

```
int close(int fd);
```

Normalmente el servidor cierra el descriptor del cliente (Descriptor_Cliente), no el del socket (Descriptor), ya que este se suele dejar abierto para volver a realizar una llamada a accept() y sacar al siguiente cliente de la cola. Es como si una vez que un amable señor de una empresa nos ha atendido le dice a la telefonista que no atienda más el teléfono. Esas cosas sólo las hacen los jefes.

Organización de los enteros - htons() y htonl() En el mundo de los ordenadores, están los micros de intel (y los que los emulan) y los demás. La diferencia, entre otras muchas, es que organizan los enteros de forma distinta; uno pone antes el byte más significativo del entero y el otro lo pone al final. Si conectamos dos ordenadores con sockets, una de las informaciones que se transmiten en la conexión es un entero con el número de servicio. ¡Ya la hemos liado! hay micros que no se entienden entre sí. Para evitar este tipo de problemas están la funciones htons(), htonl() y similares. Esta función convierte los enteros a un formato "standard" de red, con lo que se garantiza que nos podemos entender con cualquier entero. Eso implica que algunos de los campos de las estructuras que hemos rellenado arriba, debemos aplicarles esta función antes de realizar la conexión. Si enviamos después enteros con write() o los leemos con read(), debemos convertirlos y desconvertirlos a formato red.

3.5 Implementación de cliente/servidor de TCP

3.5.1 Primer intento

Este es un ejemplo simple. Al terminar la sesión el servidor muere.

```
// Código del Servidor
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define PORT 4000 /* de 1 a 1024 solo root */
            /* ports ya usados estan en /etc/services */
typedef struct sockaddr *sad;
void error(char *s)
{
    perror(s); /* exit((perror(s),-1)) */
    exit(-1);
}
int main()
{
    int sock, sock1;
    struct sockaddr_in sin, sin1;
    socklen_t L; /* int L;!?!? */
    char linea[1024];
    int cto;
    if((sock=socket(PF_INET, SOCK_STREAM,0))<0)
        error("socket");
    /* hacemos la direccion por la que recibira la comunicacion */
    sin.sin_family=AF_INET;
    sin.sin_port=htons(PORT); /* host to network short */
    sin.sin_addr.s_addr=INADDR_ANY;
    /* cualquier interfaz...no tengo que poner htonl?! */
                                        /* pegamos la direccion al socket */
    if(bind(sock,(sad) &sin,sizeof sin)<0)
        error("bind");
    /* indicamos que vamos a escuchar */
    if(listen(sock,5)<0)
        error("listen");
    /* esperamos conexion */
    L=sizeof sin1;
    if((sock1=accept(sock,(sad) &sin1,&L))<0)
        error("accept");
    /* procesamos con sock1 */
    printf("De %s,%d ",inet_ntoa(sin1.sin_addr),ntohs(sin.sin_port));
    if((cto=read(sock1,linea,sizeof linea))<0)
        error("read");
    linea[cto]=0;
    printf("llega [%s]\n",linea);
    linea[0]++; /* alteramos y devolvemos */
    if(write(sock1,linea,cto)!=cto)
        error("write");
    /* cerramos sesion */
    close(sock1);
    /* cerramos todo */
    close(sock);
    return 0;
```

```
}
/* Código del Cliente */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define PORT 4000
typedef struct sockaddr *sad;
void error(char *s)
{
    exit((perror(s),-1));
}
int main()
{
    int sock;
    struct sockaddr_in sin;
    char linea[1024];
    int cto;
    if((sock=socket(PF_INET,SOCK_STREAM,0))<0)
        error("socket");
    /* armamos la direccion */
    sin.sin_family=AF_INET;
    sin.sin_port=htons(PORT);
    inet_aton("127.0.0.1",&sin.sin_addr);
    /* conectamos */
    if(connect(sock,(sad) &sin, sizeof sin)<0)
        error("connect");
    if(write(sock,"hola mundo\n",10)<0)
        error("write");
    if((cto=read(sock, linea, sizeof linea))<0)
        error("read");
    linea[cto]=0;
    printf("Devuelve [%s]\n", linea);
    close(sock);
    return 0;
}
```

3.5.2 Segundo intento (servidor paralelo)

En este ejemplo modificamos el servidor para que espere nuevas sesiones y no muera al terminar la primera. El código del cliente queda intacto.

```
/* server TCP paralelo */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/sockets.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>
#define PORT 4000
typedef struct sockaddr *sad;
void error(char *s)
{
    exit((perror(s),-1));
}
int main()
{
    int sock, sock1;
    struct sockaddr_in sin, sin1;
    socklen_t L; /* De nuevo... int L;?!!? */
    char linea[1024];
    int cto;
    if((sock=socket(PF_INET, SOCK_STREAM,0))<0)
        error("socket");
    sin.sin_family=AF_INET;
    sin.sin_port=htons(PORT);
    sin.sin_addr.s_addr=INADDR_ANY; // no es necesario usar
                                    // hotnl(INADDR_ANY) porque INADDR_ANY = 0
    if(bind(sock,(sad) &sin,sizeof sin)<0)
        error("bind");
    if(listen(sock, 5)<0)
        error("listen");
    /* ciclo vital */
    for(;;){
        L=sizeof sin1;
        if((sock1=accept(sock,(sad) &sin1,&L))<0)
            error("accept");
        /* procesamos con sock1 */
        printf("De %s,%d ",inet_ntoa(sin11.sin_addr),ntohs(sin.sin_port));
        if(!fork()){ /* hijo */
            close(sock);
            if((cto=read(sock1,linea,sizeof linea))<0)
              error("read");
            sleep(20);
            /* demoro 20 segundos...por motivos practicos, para ver que funciona */
            linea[0]++; /* alteramos y devolvemos */
            if(write(sock1,linea,cto)!=cto)
              error("write");
            /* cierra sesion y termina */
            close(sock1);
            exit(0);
        }
        /* padre */
        close(sock1); /* cierra sesion */
                        /* vuelve */
    }
    return 0;
}
```

Esta implementación tiene un problema. El padre no hace wait para ir a de zombie a end!!

3.5.3 Tercer intento (servidor paralelo que hace los wait)

En este ejemplo queremos reparar el problema anterior y que el padre llame a wait para eliminar los zombies. La solución se realiza con el uso de señales.

Cambiamos el server de esta manera:

```
Antes de main ponemos
void chld(int sig)
{
    wait(NULL);
}
y en main agrego
signal(SIGCHLD,chld);
```

3.6 Implementación de cliente/servidor de UDP

```
/* server UDP */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define PORT 5000
typedef struct stockaddr *sad;
void error(char *s)
{
    exit((perror(s),-1));
}
int main()
{
    int sock;
    struct socaddr_in sin, sin1;
    socklen_t L;
    char linea[1024];
    int cto;
    /* socket UDP */
    if((sock=socket(PF_INET,SOCK_DGRAM,0))<0)
        error("socket");
    /* direccion del server: todas las interfaces */
    sin.sin_family=AF_INET;
    sin.sin_port=htons(PORT);
    sin.sin_addr.s_addr=INADDR_ANY;

    /* pegamos la direccion a sock */
    if(bind(sock,(sad)&sin,sizeof sin)<0)
        error("bind");
    /* prcesamos */
    L=sizeof sin1;
    if((cto=recvfrom(sock,linea,sizeof linea,0,(sad) &sin1, &L))<0)
        error("recvfrom");
```

```
    linea[cto]=0;
    printf("De (%s,%d) llega [%s]\n",inet_ntoa(sin1.sin_addr),ntohs(sin1.sin_port),linea);
    linea[0]++;
    if(sendto(sock,linea,cto,0,(sad) &sin1,L)!=cto)
        error("sendto");
    /* fin */
    close(sock);
    return 0;
}
/* client UDP */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define PORT 5000
typedef struct stockaddr *sad;
void error(char *s)
{
    exit((perror(s),-1));
}
int main()
{
    int sock;
    struct sockaddr_in sin;
    socklen_t L;
    char linea[1024];
    int cto;
    /* socket UDP */
    if((sock=socket(PF_INET,SOCK_DGRAM,0))<0)
        error("socket");
    /* direccion del server: todas las interfaces */
    sin.sin_family=AF_INET;
    sin.sin_port=htons(PORT);
    inet_aton("127.0.0.1",&sin.sin_addr); /* !? */
    if(sendto(sock, "hola mundo",10,0,(sad) &sin, sizeof sin)!=10)
        error("sendto");
    if((cto=recvfrom(sock, linea, sizeof linea,  0, NULL, NULL))<0)
        error("recvfrom");
    linea[cto]=0;
    printf("[%s]\n",linea);
    close(sock);
    return 0;
}
```

4 Select

4.1 Atención al cliente

Normalmente a un programa servidor se le pueden conectar varios clientes simultáneamente. Por ello un programa servidor debe estar preparado para esta circunstancia.

Hay dos opciones posibles.

- Una opción es crear un nuevo proceso o hilo por cada cliente que llegue, más el proceso o hilo principal, pendiente de aceptar nuevos clientes.
- La otra opción sería que hubiera algo que nos avisara si algún cliente quiere conectarse o si algún cliente ya conectado quiere algo de nuestro servidor. De esta manera, nuestro programa servidor podría estar "dormido", a la espera de que sucediera alguno de estos eventos.

La primera opción, la de múltiples procesos/hilos, es adecuada cuando las peticiones de los clientes son muy numerosas y nuestro servidor no es lo bastante rápido para atenderlas consecutivamente.

La segunda es buena opción cuando recibimos peticiones de los clientes que podemos atender más rápidamente de lo que nos llegan.

4.2 Problema introductorio: hacer un chat

El problema es que fgets (para leer del teclado) y read (si usamos TCP) y recvfrom (si usamos UDP) son BLOQUEANTES entonces en un lado no voy a poder escribir hasta recibir del otro.

Hay que esperar en ambas. Esto se puede hacer con select.

```
int select(int max+1, fd_set *in, fd_set *out, fd_set *exc, struct timeval *tv)
/* fd_set= file descriptor set */
```

El prototipo esta en sys/select.h.

Los fd_set son conjuntos de descriptores de archivos.

select bloquea en los archivos de los fd_set y vuelve si uno o mas de los archivos en ellos cambia de estado:

- in: se puede leer
- out: se puede escribir
- exc: si ha habido alguna excepción //cambio algo
- tv: un timeout (NULL equivale a infinito)

in, out y exc pueden ser NULL.

La estructura timeval esta en time.h y más o menos es lo siguiente

```
struct timeval{

    unsigned tv_sec; /* segundos */
    unsigned tv_usec; /* microsegundos */

}
```

max+1= el maximo fd+1

Para manipular los fd_set se tiene las siguientes macros

- FD_ZERO(fd_set*): pone a cero el fd_set
- FD_SET(int i, fd set*): agrega i al fd_set
- FD_CLEAR(int i, fd set*): saca i de fd_set
- FD_ISSET(int i, fd set*): nos indica si ha habido algo en el descriptor int dentro de fd_set. Cuando select() sale, debemos ir interrogando a todos los descriptores uno por uno con esta macro.

Nota: para saber que interfaces tiene un host y que direcciones tienen esas interfaces /sbin/ifconfig

4.3 Ejemplo de uso

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <time.h>
#define PORT 5000
typedef struct sockaddr *sad;
#define MAX(x,y) ((x)>(y)? (x):(y))
void error(char *s)
{
    perror(s);
    exit(-1);
}
int main(int argc, char **argv)
{
    int sock;
    char linea[1024];
    socklen_t l;
    int cto, mm1;
    struct sockaddr_in sin;
    fd_set in, in_orig;
    if (argc!=2){
        fprintf(stderr,"uso: %s dirección \n" , argv[0]);
        exit(-1);
    }
    //hacemos el fd_set
    FD_ZERO(&in);
    if ((sock=socket(PF_INET, SOCK_DGRAM, 0)) < 0)
        error("socket");
    //direccion
    sin.sin_family=AF_INET;
    sin.sin_port= htons(PORT);
    sin.sin_addr.s_addr= INADDR_ANY;
    if (bind(sock,(sad)&sin, sizeof sin) < 0)
        error("bind");
    //commpletamos aunque jamas puse que iba a ser in_orig por lo tanto puede tener cualquiera adentro a
    FD_SET (0, &in_orig); // TECLADO
    FD_SET(sock, &in_orig);
    mm1= MAX (0, sock) +1;
    //direccion destino
    sin.sin_family= AF_INET;
    sin.sin_port= htons(PORT);
    inet_aton(argv[1], &sin.sin_addr);
    //ciclo vital
    for (;;){
        memcpy(&in, &in_orig, sizeof in);
        if(select (mm1, &in, NULL, NULL, NULL)<0) //podria ser 0
```

```
            error("select");
        if (FD_ISSET(0, &in)) { // veo si me despertó el teclado
            fgets(linea, sizeof linea, stdin);
            if (sendto (sock, linea, strlen(linea), 0, (sad)&sin, sizeof sin)<0)
              error("sendto");
        }
        if (FD_ISSET(sock, &in)){ // o si me despertó un mensaje de otro cliente
            if((cto=recvfrom(sock, linea, sizeof(linea), 0, NULL, NULL))<0)
              error("recvfrom");
            linea[cto]=0;
            printf("%s", linea);
        }
    } //del for
    return 0;
}
```

- Uso: ejecutar y pasar como parámetro la dirección IP de la otra máquina a usar. A la otra pasarle la dir IP de esta máquina.

5 POSIX Threads

- Son una librería que permite threading en C.
- Threads son procesos que comparten memoria.
- ¿Cuál es la máxima velocidad de copiado de un dato entre dos procesos? CERO, si ambos comparten memoria. Para esto está el threading.

5.1 Ejemplo

```
#include <stdio.h>
#include <pthread.h>
//codigo de los threads
void *thread (void *arg)
{
    int quien=*(int*)arg;
    printf("soy %d\n", quien);
    return NULL;
}
int main()
{
    int i,s=5;
    pthread_t t[s];
    int arg[s];
    for (i=0; i<5; i++){
    // 5 threads. OJO, no se sabe nada sobre el orden de ejecucion de los threads!!!
        arg[i]=i;
        pthread_create(&t[i], 0, thread, &arg[i]);
    } //frenamos main
    for (i=0; i<s; i++)
        pthread_join(t[i], NULL);
```

```
    return 0;
}
```

- SE COMPILA CON: gcc nombre.c -lpthread
- **NOTA:** lo único propio (no compartido) de los threads es el STACK. Todo lo demás ES compartido.
- POSIX ejecuta MILES de instrucciones de threads antes de pasar a otro proceso. En la práctica usamos Pascal-FC porque, a diferencia de Posix, pascal-fc ejecuta 'DOS o TRES' threads antes de pasar a otro proceso.*/

5.2 Los threads y las regiones críticas

Utilizacion de MUTEXS - semaforos

Cambiemos la funcion thread del ejemplo anterior por la siguiente:

```
void *thread(void *arg)
{
    int quien=*(int*) arg;
    static int cosa; // recurso compartido (RC)
    for (;;){
        // INICIO REGIÓN CRÍTICA
        cosa=quien;
        fflush(stdout);
        sleep(1);
        printf("%d: %s", quien, cosa==quien?"bien":"auch");
        // FIN REGIÓN CRÍTICA
    }
    return NULL;
}
```

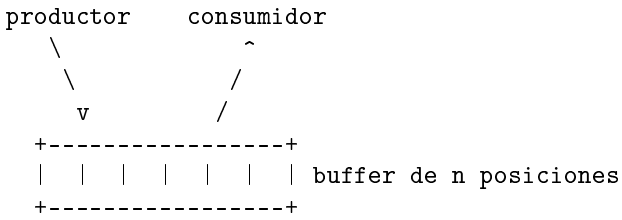
- Para regular el acceso a la RC, pthreads ofrece MUTEXES (mutual exclussion).
- Si un thread trata de tomar un mutex ya tomado, bloquea hasta que este sea liberado.

  Veamos como usar esto en un ejemplo:

```
pthread_mutex_t sem= PTHREAD_MUTEX_INITIALIZER;

void *thread(void* arg)
{
    int quien=*(int *) arg;
    static int cosa;
    //extern sem;
    for(;;){
        pthread_mutex_lock (&sem); //// SIN EN T
        cosa=quien;
        fflush(stdout);
        sleep(1);
        printf("%d: %s", quien, cosa==quien?"bien":"auch");
        pthread_mutex_unlock(&sem); ////
    }
    return NULL;
}
```

5.3 Problema del productor/consumidor con buffer acotado

Escenario:



Ciclo del productor:

```
for(;;){
    producir un item
    lock();
    while(buffer no está lleno) // REGION
        colocarlo en el buffer   // CRITICA
    unlock;
}
```

Si el buffer está lleno, el consumidor no puede entrar porque el mutex lo tiene el productor.

Posible Solución: si la condición llena/vacío no se cumple, soltar el mutex.
¿y en el interín que hacemos? Podemos bloquearlos y hacer que otro proceso los despierte de algún modo.

Imaginemos que podemos hacer algo asi:

```
queue q;
lock();
while (buffer lleno){
    (1) unlock();
    (3) wait(q);
}
```

El consumidor, cuando consume, nos despierta:

```
consumimos;
(2) signal(q);
```

ESTO TIENE UN ERROR muy sutil. Es la secuencia marcada. Esto se arregla si (1) y (3) son una secuencia ATOMICA.
POSIX llama a esto condición de variable de cola.
(obs: se usa while en vez de if para re-chequear la condicion por si hay varios threads corriendo)

```
#define N 10
int buffer[N];
int in,out, ctos;
pthread_mutex_t sem = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t lleno = PTHREAD_COND_INITIALIZER, vacio = PTHREAD_COND_INITIALIZER;
void *prod(void *arg)
{
    for(;;){
        sleep(1);
        pthread_mutex_lock(&sem);
        while(ctos >=N)
```

```
            pthread_cond_wait(&lleno, &sem);
        buffer[in]=44;
        in ++;
        if (in >=N)
            in=0;
        ctos++;
        pthread_cond_signal(&vacio); // siempre conviene
        pthread_mutex_unlock(&sem);  // en este orden
    }
}
void *cons(void*arg)
{
    for(;;){
        sleep(1);
        pthread_mutex_lock(&sem);
        while(ctos <=0)
            pthread_cond_wait(&vacio, &sem);
        printf("%d", buffer[out]);
        out++;
        if(out >=N)
            out=0;
            ctos--;
        pthread_cond_signal(&lleno);
        pthread_mutex_unlock(&sem);
    }
}
```

AGREGADO: Además de pthread_cond_signal está pthread_cond_broadcast

5.4 Semáforos de Dijkstra

- Para un MUTEX, ¿Puede un thread que ya lo tiene volver a tomarlo?

  Sí, puede.

  - En ese caso, ¿cuántos unlock's debe hacer para liberarlo?

    Si debe hacer tantos unlocks como locks, el mutex se llama RECUERSIVO. Además, se considera un error que el unlock lo haga un thread que no tengo el mutex.

Semáforos de Dijkstra

Escenario: tienen dos operaciones P y V

- P toma un semáforo y lo decrementa, si el valor resultante es negativo, se bloquea.
- V toma un semáforo y lo incrementa, si el valor es positivo despierta un thread bloqueado.

Implementación

```
#include <pthread.h>
typedef struct{
    int valor;
    pthread_mutex_t *sem;
    pthread_cond_t *q;
```

```
} *Dijkstra;
Dijkstra sem_init(int valor)
{
    Dijkstra d;
    d = malloc(sizeof(*d));
    d->valor = valor;
    d->sem = malloc(sizeof pthread_mutex_t);
    pthread_mutex_init(d->sem, 0);
    d->q = malloc(sizeof pthread_cond_t);
    pthread_cond_init(d->q, 0);
    return d;
}
void sem_free(Dijkstra d)
{
    free(d->sem);
    free(d->q);
    free(d);
}
void P(Dijkstra d)
{
    pthread_mutex_lock(d->sem);
    while(d->valor < 0)
        pthread_cond_wait(d->q,d->sem);
    d->valor--;
    pthread_mutex_unlock(d->sem);
}
void V(Dijkstra d)
{
    pthread_mutex_lock(d->sem);
    d->valor++;
    if(d->valor > 0)
        pthread_cond_signal(d->q);
    pthread_mutex_unlock(d->sem);
}
```

Recomendación: Leer The little book of semaphores - Downing

5.5 Funciones de pthread

Aquí se explican brevemente todas las funciones de la unidad

```
pthread_mutex_t sem = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t lleno = PTHREAD_COND_INITIALIZER, vacio = PTHREAD_COND_INITIALIZER;
pthread_mutex_t *sem;
pthread_cond_t *q;
pthread_mutex_init(d->sem, 0);
pthread_cond_init(d->q, 0);
pthread_mutex_lock(pthread_mutex_t *);
pthread_mutex_unlock(pthread_mutex_t *);
pthread_cond_signal(pthread_cond_t *);
pthread_cond_wait(pthread_cond_t *,pthread_mutex_t *);
```

6 Scheduling

6.1 Shortest Job First

Habrán notado que, en la discusión del threading, hay un concepto que no precisamos: en un cambio de threads, cómo se elige el thread a ejecutar (problema de scheduling)?

Normalmente el scheduling tiene dos aspectos:

1. Cómo implementarlo
2. Qué políticas usar para gestionar los procesos

Empecemos con un caso limite: no tener multitarea.

Escenario: tenemos 4 procesos que tardan 5,2,1,7 unidades de tiempo. Cuanto tardan efectivamente en salir de la cola los procesos?
1 tarda 5
2 tarda 5+2
3 tarda 5+2+1
4 tarda 5+2+1+7

Tiempo promedio 8,75.

Podemos achicar ese promedio? Si ejecutando en este orden 3,2,1,4 ahora tenemos
3: 1
2: 1+2
1: 1+2+5
4: 1+2+5+7

Ahora el promedio es 6,75.

A esta política se la conoce como Shortest Job First (sjf). Pero ahora cómo sabemos cuánto dura un proceso? Lo damos por sabido (de alguna manera).

Volvamos a la multitarea: sabemos que una posible variable a minimizar es el uso del quantum de tiempo.
El inconveniente es que es dificil hacer predicciones sobre todo del futuro. Pero podemos usar el pasado (que cuota del quantum usó) para estimar el futuro.
Por supuesto hay varias maneras de hacer eso:

- Promediar los n últimos usos
- Promediar con pesos. Esto atenua los efectos de cambio de conducta del proceso.
  Ej: $P = \frac{p_0 + \frac{p_1}{1!} + \frac{p_2}{2!} + \cdots + + \frac{p_n}{n!}}{n}$

Pero esto puede perjudicar a un grupo de procesos posponiendo indefinidamente su ejecución.

6.2 Round Robin

- Una politica que ASEGURA que todos los procesos recibirán un quantum es armar una ronda. Esto se conoce como Round Robin.
  - Each process is assigned a time interval, called its quantum, which it is allowed to run.
  - If the process is still running at the end of the quantum, the CPU is preempted and given to another process.
  - If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course.
  - All the scheduler needs to do is maintain a list of runnable processes. When the process uses up its quantum, it is put on the end of the list.

  - setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests.

- Puede ocurrir que haya procesos más importantes que otros.
  - Cómo podemos favorecerlos? Necesitamos saber qué importancia tienen.
  - Podemos asignarles prioridades y con eso operar.
  - Para evitar desfavorecer algunos procesos (unfair) podemos combinar promedios y prioridades. Otra forma es que un quantum variable de acuerdo a la prioridad (el quantum debe tener cota superior).
- Además podemos tener el caso de procesos esporádicos pero que deben atender estallidos (busts) con mucha rapidez. Le damos el quantum que necesite (First Come First Served).
- También se pueden considear distintas políticas para distintos grupos. Otra politica es recalcular periódicamente las prioridades.
- Hay que evitar bandearse y evitar el trashing (se la pasa haciendo cosas propias como ponele se pone a hacer calculos de prioridades y onda asi).

7 Deadlock

Un deadlock es cuando los procesos involucrados no pueden progresar. Si uno de ellos no existiera los demás podrían pogresar. Son necesarios para esto:

1. Dos o más recursos compartidos con exclusión mutua
2. Que un proceso que tenga un recurso pueda retenerlo aún bloqueado
3. Que el recurso no pueda ser preemptable
4. Se forma un ciclo en el grafo de pedido-tomado. (los recursos y los procesos son nodos. Si hay una arista de un recurso a un proceso es porque el proceso quiere ese recurso y si un a flecha va de un proceso a un recurso entonces tiene ese recurso.)

Las tres primeras son consecuencia de diseño la cuarta es accidental.

7.1 Inversion de Prioridades

Veamos este escenario:

- un solo recurso R mutex (mutuamente exclusivo)
- dos procesos
- ambos hacen busy waiting (en vez de bloquear se quedan esperando)
- el scheduler usa prioridades
- un proceso de alta prioridad (P) y el otro de baja (p).

Traza

1. p toma a R
2. cambio de contexto P pide a R. Pasa a busy waiting
3. P y p estan en ready
4. el scheduler elige a P.
5. Pasa a 2). DEADLOCK!!!

- Cómo puede ser que esto pase si tenemos un sólo recurso y dos procesos?

  Esto se conoce como Inversion de Prioridades.

- Hay DOS recursos: R y el tiempo del procesador.

- Cómo se puede arreglar este problema?

  Intercambiando las prioridades hasta que p libere el recurso. Equivale a preemptar a P el tiempo del procesador.

7.2 Modos de evitar deadlocks

Esto equivale a poder preverlo o equivalentemente a que el sistema siempre opera en estados en que sean imposibles los deadlocks.

7.2.1 Deadlock Avoidance

La idea es intentar evitar alguna de las condiciones que provocan deadlocks

- Numerar los recursos y estipular que un proceso si los toma los debe tomar de menor a mayor (no puedo tomar el recurso 4 antes de tomar el 3).

  - Evita los ciclos.
  - Necesita saber qué recursos va a tomar antes de necesitarlos. En otras palabras la secuencia de toma de recursos debe ser posible de ordenacion (sort) topoligica.

- Hay casos más particulares, si llegamos a pretender que no haya recursos exclusivos y reemplazar el recurso por un proceso.

  - Los spoolers hacen eso. En ese caso la impresora solo puede ser accedida por el spooler para imprimir. Los procesos le envían todos los datos al spooler

- **Recursos preemtables:** necesito implementar algo de recovery

- **Dos o más recursos por proceso:** exigir que los procesos tomen todos los recursos simultaneamente

7.2.2 Banker's Algorithm

Es un algoritmo general desarrollado por Dijkstra.

- Evalúa los riesgos de dar creditos a sus clientes de acuerdo a los creditos que tiene tomados, que tiene pedidos.

- Sirve incluso si se tienen n recursos del mismo tipo

- Nadie realmente lo usa y la practica muestra que es más sencillo dejar que se produzca un deadlock y matar uno de los procesos.

7.3 Comentarios

- Un proceso sufre starvation si nunca tiene la posiblidad de acceder a su region critica. En el problema de los filosofos comensales se requiere que ningun proceso sufra starving.

- En sistemas donde el deadlock sea altamente probable diseñarlo con deteccion y roll-back. (sugerencia, mirar The Problem of Concurrency de Lee)

- Aún sin región crítica aparente, si se usa threading y se accede a variables globales hay que usar barreras. Pués si un thread escribe en RAM otro puede pisar el valor. Usar lock

8 Sistemas de archivos

Un sistema de archivos (File System) sirve para gestionar informacion persistente. Deber permitir:

- Guardar informacion
- Consultarla
- Actualizarla
- Eliminarla

Tipicamente se usan medios magnéticos. Pueden ser cintas, discos, etc.
Todo sistema de archivos contara con dos cosas centrales:

- La información que tiene que guardar (data)
- La información extra para la gestión (metadata)

Veamos algunos sistemas de archivos

8.1 TAR (UNIX)

Usa un sistema de archivos con esta estructura:

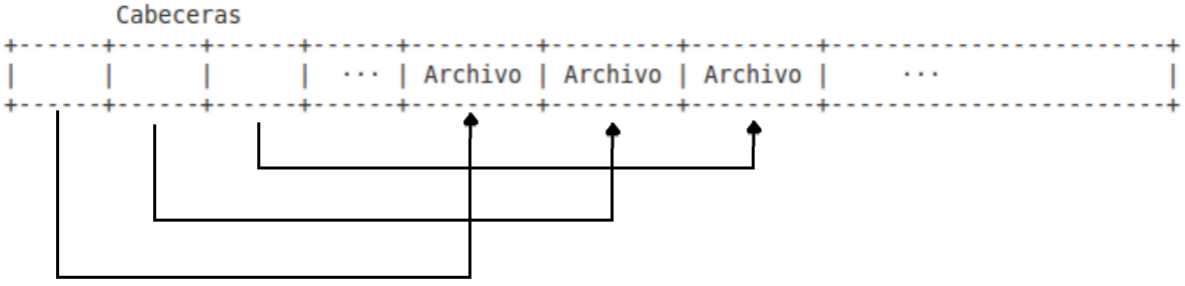
```
        cabecera                cabecera
        ------------------------------------------------------------------------------------
        |512 B  |   archivo   |*| 512 B   |   archivo   |*|            ...               |
        ------------------------------------------------------------------------------------
*= Relleno hasta un multiplo de 512
la cabecera contiene: Nombre, permisos, tamaño, etc
```

- Ventajas
  - Simple
- Desventajas
  - Acceso secuencial
  - Actualizacion compleja
  - Los archivos están en un sub bloque

Sirve para ocasinoes en que actualizacion, etc. no sea necesaria.

8.2 Concentrar la metadata (Commodore 64)

Podemos saber qué cabeceras están libres con un valor en particular.



La 1° cabecera apunta al 1° archivo, la 2° al segundo, etc

- Ventajas
  - Simple
  - Es más rápido saber si un archivo está o no
- Desventajas
  - Actualizacion compleja
  - Tamaño fijo del sector de cabeceras.
  - Los archivos están en un sub bloque

8.3 Questar/MFS

Trató de permitir la fragmentación de archivos

```
                                                Disco
          ------------------------------------------------------------------------------------
          |      *      | | | | | | | ... | | | | | | | |
          ------------------------------------------------------------------------------------
           mapa de bits | catalogo (con n dir entries**)|

 *= 1 bit por sector de 512 e indicaba si estaba ocupado
 **= cada Dir Entry tenia
 +------+-----------------+-------+------------+----------------+------------+------------+---+
 |nombre|     en ASCI     | largo | 1er sector | nro de sectores| 2er sector | 3er sector |...|
 +------+-----------------+-------+------------+----------------+------------+------------+---+
 |  7+3 | clave de acceso |       |                      Fragmentos                            |
```

Problema: Fragmentacion acotada.

NOTA: Un subdirectorio es simplemente un archivo solo que sus fragmentos son dir entries.

8.4 MSDOS FS

A principio de los 80 MICROSOFT comienza a usar este File System. Se basa en unificar dos aspectos a manejar:

- Gestion del espacio libre
- Permitir y gestionar la fragmentacion arbitraria.

Hicieron esto con una estructura llamada FAT (File Allocation Table). El medio magnético aparece asi:
(MSDOS en lugar de sectores de 512 bytes usa clusters de 4 sectores fisicos contiguos)

```
arranque        1era FAT        2da FAT         directorio raiz (catalogo raiz)         resto del disco
=============================================================================================================
|       |                 |           |      |       |      | ... | |         |                              |
=============================================================================================================
                                                dir. entries

cada dir. entry tiene
=============================================================================================
| nombre (2 B) | ext (3 B) | atributos (oculto, etc) | creacion | modificacion | nro de cluster |
=============================================================================================
                                                  |          fechas         |     inicio     |
```

La FAT es un arreglo de enteros de 24 bits para los disquettes; luego se estira a 32 para los primeros rígidos. A cada cluster le corresponde un entero, en el mismo orden.

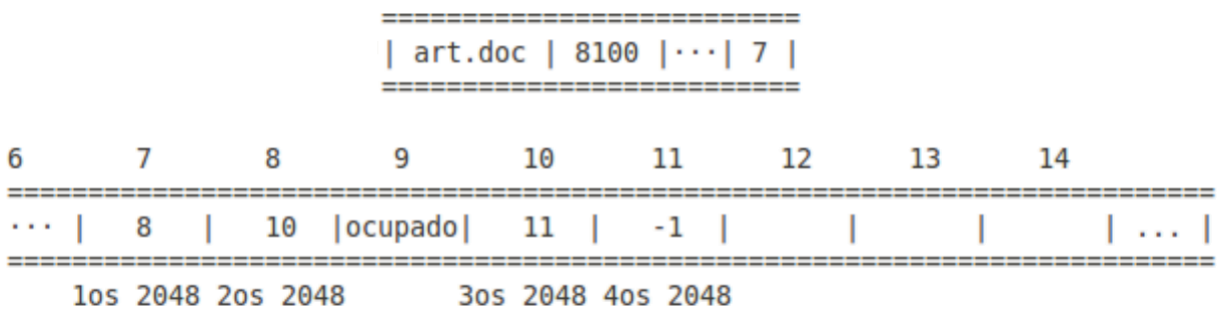
```
cluster 0 <---> entero 0

clsuter 1 <---> entero 1
```

...

El valor del entero determina el estado del cluster correspondiente:

- 0: cluster libre
- fffffe: cluster dañado (seteado durante el formateo)
- El valor de un cluster usado por un archivo es ffffff si es el último; caso contrario el número de cluster siguiente.

8.4.1 Ejemplo - Un archivo con 8100 btes de largo

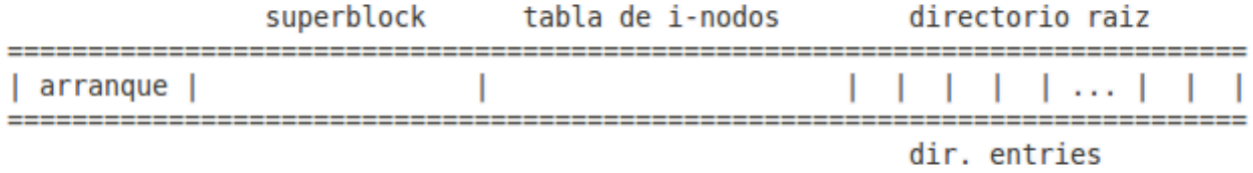


La FAT permite representar la ubicación de los clusters de un archivo como una lista enlazada.

- Ventajas
  - Es un buen FS
- Desventajas
  - Es un buen FS para discos chicos. Si la FAT no cabe en memoria hay varias lecturas extras para acceder a un archivo
  - La politica de actualizacion sincrona de la FAT lentificaba la escritura
  - The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time. Conceivably the table could be put in pageable memory, but it would still occupy a great deal of virtual memory and disk space as well as generating extra paging traffic.

8.5 UNIX FS

Hecho a principios de los 70. Es resumen del System7. El disco se estructura asi:

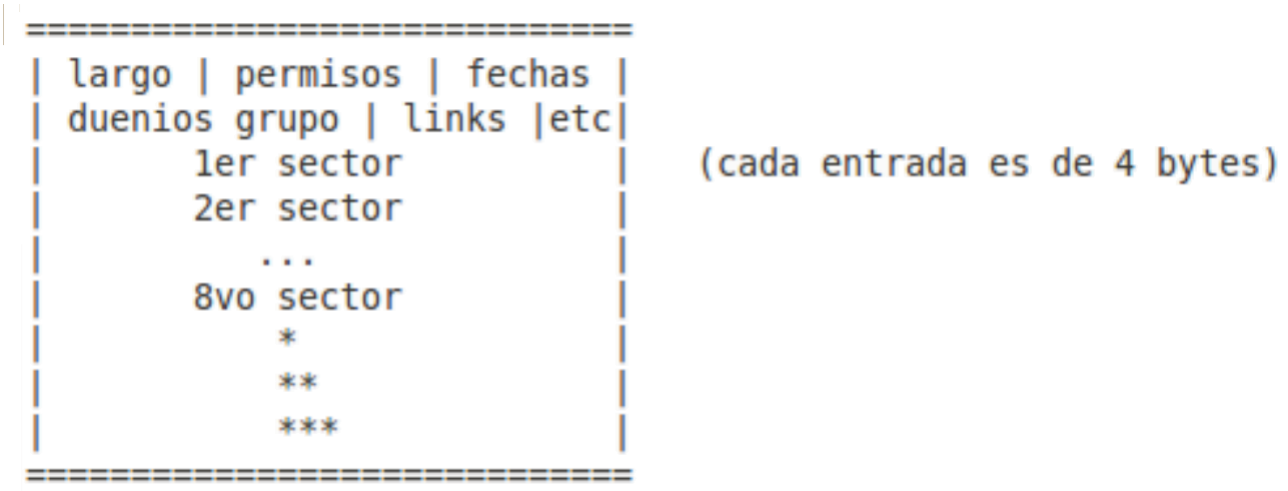


El superblock tiene:

- Un mapa de bits para sectores libres y ocupados
- Una lista enlazada con los i-nodes libres.

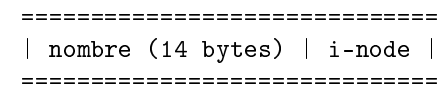
- The first one is the superblock. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file system type, the number of blocks in the file system, and other key administrative information.

Un i-nodo (information node)
Sector logico (mínimo tamaño que toma el sistema operativo): 512 bytes (coincide con el físico) tiene



en * habia una dirección a una tabla con 128 entradas de 4 bytes que indicaban dónde estaban los sectores 9, 10...etc.
** se conocia como 2 indirecciones y basicamente apuntaba a una tabla como a la que apuntaba * pero en vez de que sus entradas apuntaran a sectores, sus entradas apuntaban a una tabla del tipo de *. Es decir que tenia 128^2 sectores de informacion.
Te animas a buscar como seria el *** (3 indirecciones)

Cada dir. entry era



Un archivo puede tener varios nombres (links). Solo necesitan estar apuntando al mismo i-node. De este modo para borrar un archivo es borrar un link y decrementar el numero de links. Cuando este llega a 0 puedo borrar el archivo

- Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an i-node (index-node), which lists the attributes and disk addresses of the files blocks. A simple example is depicted in Fig. 6-15. Given the i-node, it is then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only kn bytes. Only this much space need be reserved in advance.

- One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of a block containing more disk block addresses, as shown in Fig. 6-15. Even more advanced would be two or more such blocks containing disk addresses or even disk blocks pointing to other disk blocks full of addresses. We will come back to i-nodes when studying UNIX later.

8.6 Mejorando el FS

Queremos mejorar el Unix FS. Con mejorar nos referminos a varias posibles lineas: seguridad, rapidez, etc.
Probemos con rapidez.

Para eso iremos al estado de California, más exactamente a la universidad de Berkeley, Oakland. BSD modifica el UNIX File System consiguiendo incrementos de velocidad de aproximadamente 80%

Como lo lograron? Hicieron dos innovaciones: aumentaron el tamaño del sector lógico, desarrollaron el algoritmo del ascensor

8.6.1 Hacer crecer el sector lógico

Un sector lógico distinto del físico. El sector logico de BSD era de 4096 bytes en lugar de 512 bytes. Se leían 4096 bytes (8 sectores) porque aunque los programas leían de a 512, el sistema operativo pedía de a 8 de estos sectores. Entonces cuando el programa pedía los proximos 512, el sistema operativo ya los tenía en cache. Esto acelero un 20%

8.6.2 Algoritmo del ascensor

Teniendo en cuenta que un disco tiene dos tipos de movimientos

- Rotación de platos (angular)
- Cabezal (radial)

Puedo optimizarlos. Se puede acelerar el acceso a un disco no poniendo dos sectores seguidos. Esto mejora el rendimiento angular [interleaping de sectores fisicos para minimizar la cantidad de revoluciones para leer una pista]

Si quiero leer 1 y 2 y están pegados quizás termino de leer 1 y quiero leer 2 pero me pase entonces tengo que esperar que el disco de toda la vuelta. En cambio si 2 no está pegado a 1 no tengo que esperar eso.

Para mejorar el rendimiento del movimiento radial se usa el ALGORITMO DEL ASCENSOR: colectar pedidos de lectura/escritura y reordenarlos de modo de no hacer mas de una o dos pasadas. Es óptimo excepto que requiere que los pedidos de lectura/escritura sean independientes.

Es decir, al inicio el cabezal está detenido. Al llegar la primer petición va una dirección. Cada pedido que llega se va cumpliendo en el orden en que aparecen en el recorrido del cabezal hasta el fnal del recorrido, luego se empieza el recorrido inverso y se responden los de esa dirección y así sucesivamente.

8.6.3 Distribución del Superblock

Qué ocurre si el sistema se cae en el medio de la creación de un directorio?

Queda un FS inconsistente que hay que devolver a un estado consistente.

Esto lo hace el fsck (File System Checker). Para esto es que las operaciones de creación deben ordenarse. Lo siguiente a mejorar consiste en distribuir el superblock. En vez de tenerlo al principio del disco lo distribuyo en partes en el disco.

Entonces una operación en vez de tener que ir al superblock al principio del disco solo tiene que ir al más cercano. Es además más seguro porque si se me caga en el que tiene el superblock todo al principio cago todo, pero en el otro si cago alguno se me caga un sector pero no todo el disco. Este sistema dio un 60% más de coso.

```
================================= ================================
|SB|             | |SB0|   |SB1|   |SB2|   | ... | ================================
==============================
Unix File System           FFS (Fast File System)
```

Esto aún se puede mejorar! En Paris ubicamos a Gelinas que hizo unos FS llamados ext y ext2. Saco las secuencias dependites (no las saca...las IGNORA!). Ahroa el fsck de los ext* es mucho más complicado.

8.7 Más mejoras

Tomemos la apertura de un archivo. El nucleo debe:

- buscar en el directorio correspondiente /* un directorio es un arreglo de dir entries O(n) */
- sacar su i-nodo de informacion.
- chequear los accesos.
- construir el descriptor en el System Segment

Podemos acelerar esto cambiando el arreglo por otra estructura. Por ejemplo, un árbol! Si está balanceado sale con O(ln N). Casos de estos son Red-Black, AVL y BTREE.

Los BTREE (Balanced Trees) fueron elaborados en 1971. "En vez de hacer crecer un arbol desde la raíz, lo hacemos crecer desde las hojas".

Podemos usar un BTREE para los directorios, idem para los sectores libes. (ordenado: por tamaño y ubicación)

El pionero en esto fue IRIX (el UNIX de SGI). Tener dos BTREES para los bloques libres es complicado y eso llevó a que IRIX tuviera que incluir journaling.

Qué es un journaling? Son acciones que se toman para, frente a un cambio de metadata permitir volver atrás cambios parciales y alcanzar un estado consistente. Es muy usado, por ejemplo en bases de datos. En esencia se guardan los valores intermedios de la matadata con un timestamp. Por supuesto si se hace 'commit' (se completa la transacción) se marca también.

Journaling baja la performance pero aumenta la seguridad.

FS con journaling: IRIX (xfs), IBM (jfs), ext3, ext4, ResierFS, etc.

Qué podemos mejorar? Arreglo-O(n) BTREE—O(log N) Hash—-O(1)

Para buscar un ejecutable tener una base de datos con las ubicaciones de los ejecutables y traerlos con una busqueda con hash. El que comenzo a usar esto con atributos de archivos fue MacOS (en Linux lo hace el shell). Esto tambien fue usado por BeFS (practical implementation of file system...me la corto si lo consigo) de BeOS usado por BeBox.

Podemos mejorar esto!? (campo experimental).

Podemos pedir:

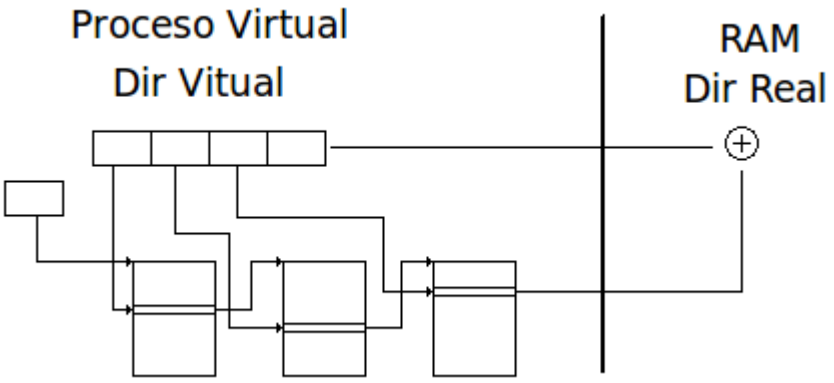
- Máxima velocidad
- Máxima seguridad

Son antagónicos pués mantener el log de transacciones para el journaling consume mucho tiempo. Para mejorar esto ponemos el FS en el LOG ahora cada vez que escribimos en el FS estamos escribiendo en el LOG también. Cada cambio se agrega en lugar libre con un timestamp.

Este FS se conoce como LogFS y fue una tesis de Ph.D. de Margo Seltzer en Berkeley. Es muy rápido pero tiene el inconveniente de derrochar mucho espacio (nunca pisa nada sino que va agregando) así que periódicamente hay que correr una aspiradora

9 Manejo de memoria

Usaremos sistemas con paginacion. (Sistemas Operativos Modernos Tanenbaum o Silberchautz)



- Todo proceso tiene sus tablas y su conjunto de páginas.

  Cuando referenciamos una página tenemos dos posibilidades

  - La página está en RAM (es un hit)
  - La página no está en RAM (es un miss)

- Para poder usar una página, esta debe estar en RAM.

- Qué ocurre cuando la RAM no tiene espacio libre y es necesario cargar una pagina?
  - Hay que hacer lugar desalojando una pagina presente y guardarla en otro lugar, tipicamente en disco (al area de swap).
- Ahora bien, qué página elegimos para desalojar (page eviction)?
  - Este problema es el de las políticas de desalojo. Debemos elegir la página más conveniente (la que no vamos a usar). Este problema equivale a predecir el futuro.

9.1 Política de FIFO

Las últimas páginas en entrar tienen una buena probabilidad de ser usadas. Entonces las primeras páginas que entraron a memoria serían buenas candidatas a ser desalojadas.

Esto es un FIFO. Es sencillo pero sufre lo que se conoce como anomalía de Belady.

9.1.1 Anomalía de Belady

Ante la política de FIFO, Belady encontró una secuencia de desalojo tal que al tener más recursos (más página físicas) había más misses.

Supongamos tener 3 páginas de RAM y necesitamos alojar 5 páginas.

Supongamos que la secuencia de uso (referencias) es

`1,2,3,4,1,2,5,1,2,3,4,5`

Veamos cuantos hits y cuantos missed hay en esta secuencia

- Pido página 1: missed | 1 | | |
- Pido página 2: missed | 2 | 1 | |
- Pido página 3: missed | 3 | 2 | 1 |
- Pido página 4: missed | 4 | 3 | 2 |
- Pido página 1: missed | 1 | 4 | 3 |
- Pido página 2: missed | 2 | 1 | 4 |
- Pido página 5: missed | 5 | 2 | 1 |
- Pido página 1: hit | 5 | 2 | 1 |
- Pido página 2: hit | 5 | 2 | 1 |
- Pido página 3: missed | 3 | 5 | 2 |
- Pido página 4: missed | 4 | 3 | 5 |
- Pido página 5: hit | 4 | 3 | 5 |

Resultado:

- missed: 9
- hits: 3

Esto es un desastre compremos más memoria. Ahora 5 páginas lágicas, 4 físicas

- Pido página 1: missed | 1 | | | |
- Pido página 2: missed | 2 | 1 | | |

- Pido página 3: missed | 3 | 2 | 1 | |
- Pido página 4: missed | 4 | 3 | 2 | 1 |
- Pido página 1: hit | 4 | 3 | 2 | 1 |
- Pido página 2: hit | 4 | 3 | 2 | 1 |
- Pido página 5: missed | 5 | 4 | 3 | 2 |
- Pido página 1: missed | 1 | 5 | 4 | 3 |
- Pido página 2: missed | 2 | 1 | 5 | 4 |
- Pido página 3: missed | 3 | 2 | 1 | 5 |
- Pido página 4: missed | 4 | 3 | 2 | 1 |
- Pido página 5: mussed | 5 | 4 | 3 | 2 |

Resultado

- missed: 10
- hits 2

Con mas paginas de memoria fisica tengo mas misses!!!

9.2 Stack Property

Debemos deseñar politicas que no sufran anomalís como la de Belady.

Mas precisamente, Si R:recursos, P:performance (por ejemplo hits), requerimos que

```
si R1≤R2 entonces P1≤P2
```

Esto se conoce como STACK PROPERTY.

9.3 Least Recently Used (LRU) page replacement algorithm

Consideremos:

1. El espejo temporal. Convierto el pasado en futuro y el futuro en pasado
2. Este limite $\lim_{t \leftarrow \inf}$ (* limite de t viniendo de infinito *)

Ya sabemos que la página óptima para desalojo es la que no se va a usar (o lo que es lo mismo, se usará en t=inf).

Usando 1, esto equivale a elegir una página que se referencio por última vez en t=-inf.

Usando 2, la página óptima es la que más tarde se va a usar. Por el espejo esto equivale a la página que se usó menos ultimamente. Esto se conoce como LRU (Least Recently Used).

Esta política no sufre la anomalia de Belady. Pero, cómo la implementamos?? Necesitamos un timestamp y un mecanismo que lo actualice por cada acceso. Podemos hacerlo por hardware. Pero tenemos:

- Diseño no estandard
- Desborde con el tiempo.

9.4 Second Chance Page Replacement Algorithm

Podemos arreglar el último problema mencionado en LRU disminuyendo los timestamps regularmente (restándoles algo o dividiéndolos) pero esto es uan operación costosa.

Ej: restándoles el menor timestamp.

Debemos utilizar otra cosa (aproximar el LRU).

Podemos probar con acotar el timestamp. De hecho, el timestamp más barato es el que usa 1 bit (1 si fue referenciada desde el último reset, 0 si no lo fue). Por supesto periodicamente se resetean. Esto no me da un LRU óptimo pero tengo un subconjunto de paginas que desde el último reseteo hasta ahora no fue referenciado.

9.5 Algoritmo del reloj con segunda oportunidad

Esto se puede mejorar con el algoritmo del reloj con segunda oportunidad.

Las páginas alojadas se disponen en anillo.

Cuando hay que desalojar una página, se recorre el anillo y se resetean.

Se deja ejecutar el sistema un poco más y elige alguna en cero. Esto obedece el principio de localidad (principio empírico que establece que si un proceso normal ha utilizado un subconjunto de páginas tiene un tiempo prolongado para seguir usando esas páginas).

Podemos mejorar esto agregando un bit extra a cada página marcándola como "limpia" si no tuvo cambios desde que vino del disco (de la swap) o si lo tuvo (dirty bit).

El algoritmo del reloj trata ahora de elegir (si las hay) páginas no referenciadas (LRU) y limpias (no hay que copiarlas). Si no las hay elige no referenciadas y las copia al swap. Estos bits estan en los descriptores de las tablas de paginacion. Por supuesto el kernel lleva aparte el conjunto de páginas que está usando cada proceso llamado working set, que es de tamaño variable.

Algunos procesadores (tipicamente el PC RT de IBM) usaron una tabla hash y otra invertida para saber la direccion virtual sabiendo la real.

En general por simplicidad el nucleo no es desalogado aunque hay excepciones como Solaris.

10 Seguridad (no entra)

Qué aspectos son importantes?

Entre otros:

- Confidencialidad
- Integridad

- Lo segundo se puede arreglar con backups, transacciones, códigos correctores, accesos controlados, etc.
- La confidencialidad con accesos controlados, autenticaciones, cifrado, etc.
- Las autenticaciones se pueden hacer con passwords.

  Posibles ataques para la contrasenia:

  - Fuerza bruta. Esto lo podemos arreglar temporalizando los intentos o dando accesos muy restringidos a las passwords.
  - Intercepción de comunicaciones (modo promiscuo). Esto se puede evitar encriptando, o con anonimato, o incluso con respuesta a broadcasts.
  - Posible intrusión. Cómo lo vemos!? Con logs, cambio de CRCs en programas, uso de sockets inusuales o tráfico inusual. Los últimos dos se pueden ver con lsof que da información sobre los sockets y recursos que usa cierto proceso

10.1 Cómo evitar intrusiones?

Principios usados:

- Siempre usar el mínimo privilegio necesario. Consiste en que un proceso solo acceda a los mínimos recursos y con las mínimas acciones permitidas.

  - Ventajas de este principio: limita mucho el danio que puede hacer un priceso malicioso.
  - Desventajas: mucha metadata extra (debemos tener una lista de derechos para toda combinacion proceso/recurso). La administracion es un infierno.

    NOTA: Las listas de procesos/recursos y sus derechos y permisos se conocen como Access Control Lists (ACL).

    Podemos relajar esto y en lugar de fijar permisos por proceso establecer DOMINIOS de permisos. Un proceso se ejecuta dentro de un cierto dominio y esto determina sus permisos. En UNIX estos dominios se asocian con USUARIOS. Esto puede ser un poco generoso...hay que tener cuidado que el proceso que se corre no sea malicioso.

    Un proceso es malicioso si lleva a cabo acciones maliciosas tales como sacar claves, alterar cosas que no debe, etc.

    Un proceso puede ser malicioso 'ex-profeso' (ej: Interbase de Borland, que tenía back doors).

    Otra forma es un proceso normal, al que se le cambia la conducta:

    * Modificando el text segment de un programa lo cual es particularmente malo si el blanco es un compilador
    * Modificando su ejecución trantando de invocarlo con argumentos, entornos, datos, etc., que lo haga fallar. Normalmente para esto se usan buffer overflow, que permite sobreescribir otras variables. Cuando se sobreescriben direcciones de retorno, se comoce como STACK SWASHING. Esto se ha usado y se usa mucho.
      - Caso concreto: eva, 1997. Se rompio la seguridad de la máquina para extraer cierta información.
      - mount "integra" file systems en otros dispositivos como directorios. Para esto, necesita privilegios grandes.
        Cómo hacer que un usuario cualquiera corra esto!?
        Con setuid que permite que un programa aún ejecutándose inicialmente en un dominio restringido puede tener los privilegios de otro dominio. En particular mount tenia setuid root. Un ataque usando mount uso un esquema como el siguiente:

        ```
        invocava a mount hexas  +-----------------------------------------------+
        mount /dev/hd0   |+128 carateres| ret'|    X  |     |/bin/sh.
        |  +-----------------------------------------------+     string lo suficientemente larga
           ret' era un jump a donde empezaban los hexas
           X era int $80 pidiendo exec de /bin/sh
        Al ejecutarse una de las funciones copiaba el puntero de mount a un arreglo de
        128 chars automático (alojada en el marco de activación)
        |       |
        +----------+
        | ret   |
        +----------+
        |       | activacion del frame
        +----------+
        |       |
        | ...   | buffer
        |       |
        +----------+
        |       |
        ```

    Y ejecutaba un shell como root. Como subsanar esto!? Ponemos un proceso 'canario'. Es como un proceso que va primero para asegurarse de que este pasando lo que corresponde que pase.

    ```
    |       |
    +----------+
    | ret   |
    ```

```
+----------+
|  canario | activacion del frame
+----------+
|        |
antes del ret verificamos el canario
```

11 STM: Software Transactional Memory

Se basa en el concepto de transacción utilizada en bases de datos: una transacción es una secuencia de operaciones que se realiza en forma atómica. Por lo tanto, ningun proceso debe poder observar un estado intermedio.

La transaccion se realiza completamente o se deshace. Si la transacción es exitosa se hace un commit (se confirma, se comprometen los datos) y ahora sí, el resto de los procesos puede observar esos cambios. Pero si la transaccion falla el sistema deshace los cambios (rollback).

Las transacciones de base de datos tienen además la obligación de mantener la memoria consistente pero además tienen que ocuparse de la persistencia de los datos en algún medio no volatil.

Nosotros utilizaremos un enfoque basado en transacciones que permite trabajar con concurrencia sin utilizar locks.

En 1986 Tom Knight tuvo la idea de hacer soporte de hardware para memoria transaccional.

En 1995 Nir Shauit y Dan Toutou hicieron STM con un enfoque optimista: los procesos pueden escribir y leer la memoria sin utilizar los locks pero se debe llevar un registro de los accesos a memoria compartida de cada transacción. Al momento de intentar hacer un commit se debe verificar que no se haya utilizado un valor que otra transacción ya modificó.

Se puede decir que penaliza a los lectores en vez de los escritores.

Qué pasa si al finalizar una transacción los datos no se pueden confirmar? Se aborta (abort) la transacción y se deshace (rollback).

Posteriormente se ejecuta (reexecute) nuevamente toda la transacción.

- Ventajas:
  - Se incrementa la concurrencia (imaginemos en un árbol dos procesos que quieran trabajar con ramas diferentes pueden hacerlo sin molestarse mutuamente)
  - Simplifica la programación: ahora no hay que estar pendientes de cuando liberar locks, ni el orden de operaciones ni el manejo de excepciones

    Nota: una transacción puede fallar en cualquier momento, obligando a un rollback.
  - Es muy escalable y mas fácil de integrar a objetos o modulos
  - Deadlock/Livelock: desde el punto de vista del usiario no puede haber deadlock ni livelock. El "gestor de transacciones" se encargara de hacer las transacciones que sean necesarias.
  - Inversión de prioridades: podría ocurrir que haya transacciones de baja prioridad que me impidan que otra de alta prioridad haga commit. Si la transacción de baja prioridad no hizo commit puedo deshacerla permitiendo el exito de la primera.
- Desvenajas:
  - Algunas operaciones (sobre todo E/S) no pueden deshacerse. En la práctica se utilizan buffers para demorar la modificación hasta el momento del commit.
  - El trabajo necesario para llevar registros y deshacer transacciones. En la práctica los sistemas de memoria transaccional son el doble de lentos que los sistemas basados en locks.

11.1 CMT: Composable Memory Transactions

La idea es permitir componer acciones atómicas dentro de una acción atómica que las englobe. Esa operación no se podía hacer atómicamente hasta el momento en sistemas basados en locks (ej: quitar un elementos de una estructura e insertarlo en otra visto como transacción. Se pueden hacer atómicas el borrado de una e inserción en la otra, pero no se puede hacer

inaccesible el estado intermedio en que el elemento movido no está en ninguna de las dos estructuras), la composición de acciones atómicas tampoco es trivial.

Qué pasa si falla algo o si hay que reahacer la transacción? El comando 'retry' permite reintentar una transacción desde el comienzo.

Cuándo se produce el retry? Cuando alguna porción de memoria accedida por la transacción hasta ese momento se haya modificado.

La primitiva orElse permite combinar dos transacciones resultando en una transacción que ejecuta la primera, si esta falla intenta ejecutar la segunda y si vuelve a fallar se repite toda la transacción. La primitiva orElse es el sueño del select componible.

Todo esto se implementó en GHC (Glasgow Haskell Compiler). Hay implementaciones para Haskell, Scheme y C (y sus derivados onda C++ y toda la banda...).

11.1.1 Propuesta de soporte en el lenguaje - Harris-Fraser

```
atomic{ //insert en lista doblemente enlazada

    newNode->prev=node;
    newNode->next=node->next;
    node->next->prev=newNode;
    node->next=newNode;

}
```

Soporte para 'variables de condicion' Conditional Critical Region (CCR)

```
atomic(queueSize>0){

    //remove from queue and use

} //si no se cumple la condición espera hasta una nueva transaccion antes de reintentar.
```

Con CMT

```
atomic{

    if(queueSize>0){
        //work
    } else{

        //retry. Ahora solo reintentara la transacción si cambió algún valor utilizado por esta
    }

}
```

11.1.2 Implementando CMT

Se requiere un considerable esfuerzo en las lógicas de gestión de las transacciones.

Dos maneras posibles:

- lock free
- locking (lo que más se usa)
  - encounter time locking: se bloquea la memoria en cada caso y se registra el cambio
  - commit time locking: cada hilo utiliza una copia de los valores en memoria compartida, al momento de escribir se bloquea la memoria y se verifican las actualizaciones.
    - Transactional locking II (Dice, Shaleu, Shauit):
      1. Lectura de un reloj al comienzo de la transacción

2. read/write -> se verifica la hora de modificación de esa porción (si es mayor a la de "begin" se hace un abort)
3. commit (locking)

– commited ordering (co-commit order): evalúan la mejor forma de organizar las transacciones basándose en el concepto de serializabilidad. Estos sistemas proporcionan por definición el mayor grado de concurrencia posible en el sentido de que estoy seguro de no obligar a realizar bloqueos innecesarios

Un problema: si se utiliza optmistic reading (leo y al final me fijo si alguien lo cambio)

```
proceso1 proceso2
atomic{ | atomic{
    if(x!=y)    | x++;
        while(true){}   | y++;
}       |    }
```

Si el proceso llegara al final de la transacción detectaria que leyó valores inconsistentes. Sin embargo si en el medio se leyó el valor incrementado de x (pero el de y todavia no se modificó) la transacción nunca termina...jamas!!!!

11.1.3 LibCMT - Duilio Protti

Es una implementacion de CMT para C
Tiene dos conceptos fundamentales:

- GTVar: representa una variable compartida
- GTTransaction: transaccion creada por un hilo

Ejemplo: incremento de un entero

```
void f(GTransaction *tr,gpointer user_data)
{
    int *i;
    i=g_transaction_read_tvar(tr,tvar_i);
    *i=++(*i);
}
void walker_thread(void *data)
{
    tr = g_transaction_new("inc",f,NULL);
    g_transaction_do(tr,NULL);
}
```

Cómo hacemos un doble incrememnto!?

```
tr = g_transaction_new("inc",f,NULL);
tr2 = g_transaction_sequence(tr,tr);
g_transaction_do(tr2,NULL)
void h(GTransaction *tr, gpointer user_data)
{
    int *i;
    i=S_transaction_read_tvar(tr,tvar_i);
    if(*i<N)
        S_transaction_retry(tr);
```

```
⋮
}
```

Ahora vemos solución para el problema de los filosofos comensales

```
void take_par(GTransaction *tr,gpointer user_data)
{
    fork1= get_transactional_read_tvar(tr,tforks[index]);
    fork2= get_transactional_read_tvar(tr,tforks[(index+1)%NP]);
    //NP number of philosopher
    if(fork1->in_use || fork2->in_use)
        g_transaction_retry(tr);
    fork1->in_use=fork2->in_use=TRUE;
}
void worker_thread(void *data)
{
    take_forks = take_pair_tr[0];
    for(i=1;i<NP;i++)
        take_forks = g_transation_or_else (take_forks, take_pair_tr[i]);
}
No esta g_transaction_do(take_forks)
```

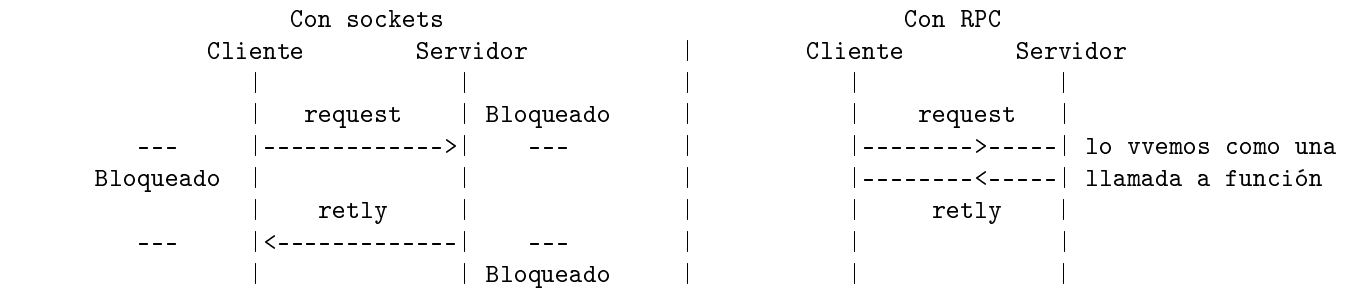
12 Remote Procedure Call (RPC)

La programación distribuída aparece cuando un sistema se ejecuta en procesos separados. Tenemos casos desde multicores que comparte RAM hasta computadoras separadas.

Existen diversos enfoques para este tipo de programación, veremos ahora el RPC (Remote Procedure Call)

- Nace con MACH en 1979 que lo implementa en MIG (MACH Interface Generator)
- En 1982 SUN toma la idea y desarrolla el llamado SUN RPC usando C (también conocido como ONC RPC de Open Network Computing)
- RPC es un protocolo que permite a un programa de ordenador ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos.
- El protocolo es un gran avance sobre los sockets usados hasta el momento. De esta manera el programador no tenía que estar pendiente de las comunicaciones, estando éstas encapsuladas dentro de las RPC.
- Las RPC son muy utilizadas dentro del paradigma cliente-servidor. Siendo el cliente el que inicia el proceso solicitando al servidor que ejecute cierto procedimiento o función y enviando éste de vuelta el resultado de dicha operación al cliente.
- Cuál es la idea de RPC? Estructurar los servicios distribuídos de acuerdo al siguiente esquema (modelo request/retry)

  En un modelo síncrono



Los servicios se pueden describir como funciones. La idea es ocultar los detalles de la comunicación al cliente y que pida un servicio a un servidor como si fuera un llamado a una función propia

- La secuencia de eventos durante un RPC

  1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
     - A stub is a piece of code used for converting parameters passed during a Remote Procedure Call.
  2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
  3. The client's local operating system sends the message from the client machine to the server machine.
  4. The local operating system on the server machine passes the incoming packets to the server stub.
  5. The server stub unpacks the parameters from the message . Unpacking the parameters is called unmarshalling.
  6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.