WEEK 9

Linux: ext2fs & ext3fs, Windows NTFS Distributed Processing

Ch 12.8-9 [Stal 05]
Ch 10.6.4, 11.6-7 [Tane 01]
Ch 20.7 [DDC 04]
Ch 13 – 14.3 [Stal05]

1

Shared file

- Hard link
  - Direct link from several directories
  - Multiple owners, all with same rights
    - removed from one directory ⟶ available still from other locations
- Soft link (symbolic link)
  - New file, which has file type: symbolic link
  - Content of the new file: path to the actual file (string that contains directories and file name)
  - Original file exists only in one directory (-> just one owner)
    - owner removes ⟶ the others cannot access, the link is invalid

2

Hard link

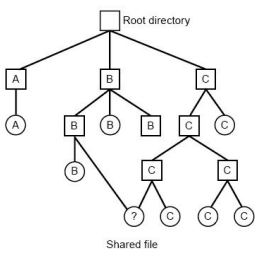


Fig. 6-18. File system containing a shared file.

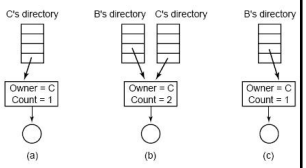


Fig. 6-19. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

[Tane 01]

3

LINUX Virtual File System

- Identical interface from the applications
- Supports several actual different file systems
- All requests via VFS

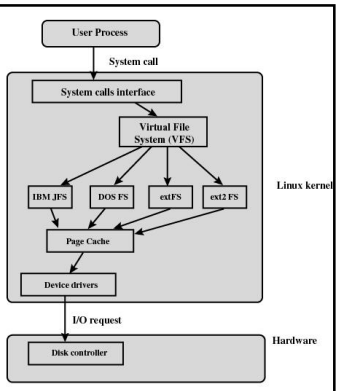


Figure 12.15 Linux Virtual File System Context

4

Disk layout – in general

- MBR (master boot record) Flash BIOS
- Partition table
  - information about the file system type on each partition
  - start and end points of each partition

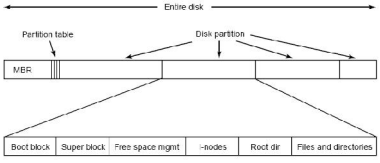


Fig. 6-11 [Tane 01]

Fig. 6-11. A possible file system layout.

5

Linux ext2fs

- block groups
  - Each group is one continuous area of the disk
  - i-nodes and datablock of one group cose to each other
  - Shorter track distances within one file
- All block sizes are identical(1 KB)
- All i-node sizes are 128B (trad. UNIX 64B)

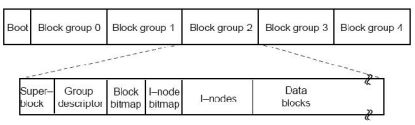


Fig. 10-35 [Tane 01]

Fig. 10-35. Layout of the Linux Ext2 file system.

6

ext2fs superblock

	0	1	2	3	4	5	6	7
0	Number of i-nodes				Number of blocks			
8	Number of reserved blocks				Number of free blocks			
16	Number of free i-nodes				First data block			
24	Block size				Fragment size			
32	Blocks per group				Fragments per group			
40	i-nodes per group				Time of mounting			
48	Time of last write				Status		Max. mnt cnt	
56	Ext2signat.		Status		Error behav.		Pad word	
64	Time of last test				Max test interval			
72	Operating system				File system revision			
80	RESUID		RESGID		Pad word			
blocksize	Pad words							

ext2fs group descriptor

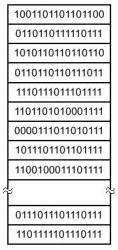
8192 blocks in one group

Each group has a copy of the super block and group descriptor

- Redundant information – the descriptor of group 0 is up-to-date

One item for one block in the group desciptor has 24 B

`bg_block_bitmap`	`location info`(4 B)
`bg_inode_bitmap`	`location info`(4 B)
`bg_inode_table`	`location info`(4 B)
`bg_free_blocks_count`	(2 B)
`bg_free_inodes_count`	(2 B)
`bg_used_dirs_count`	(2 B)
`bg_pad, bg_reserved`	(6 B)



A bitmap

ext2fs i-node

	0	1	2	3	4	5	6	7
0	Mode		Uid		File size			
8	Access time				Time of creation			
16	Time of modification				Time of deletion			
24	Gid		Link counter		No. of blocks			
32	File attributes				Reserved (OS-dependent)			
40	12 direct blocks							
88	One-stage indirect block				Two-stage indirect block			
96	Three-stage indirect block				File version			
104	File ACL				Directory ACL			
112	Fragment address				Reserved (OS-dependent)			
120	Reserved (OS-dependent)							

Access Control List

9

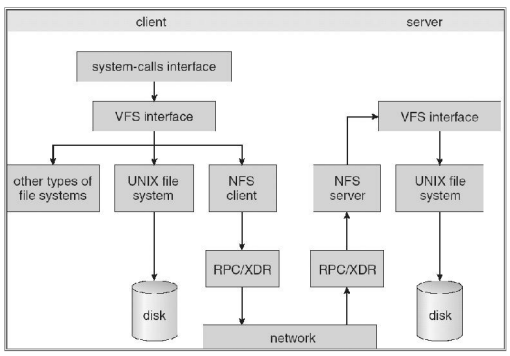
Linux procfs - process file system

- Not a true physical file system, just an interface to process control blocks
  - directory /proc
  - Each subdictory has its own read() ja write() operations
    - /proc/4321 is the directory for process 4321
  - read() and write() operations access a protected data structures using the control mechanisms of the file system (protection, concurrency)

10

NFS

See Ch 10.6.4 [Tane 01]



(Fig 11.15, [SGG07]

11

Linux ext3fs

[Tweedie talk, 20.7.2000]

(http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html)

- ext3fs = journaling ext2fs
  - Journaling is only an additional layer o top of ext2fs
  - Journaling layer is undependent of the file system under it -> ext2fs does not need to know
  - Operations follow transactional semantics
- Main goal: shorten a recovery duration after an unplanned shutdown
  - Fsck (e2fsck) takes hours in a big file system
  - There is a need for better availability
- Compatible with ext2fs
  - Clean, correctly unmounted ext3fs does not contain any journal information, it can be mounted also as ext2fs file system

TKTL: uses ext3fs

12

Windows NTFS: Master File Table

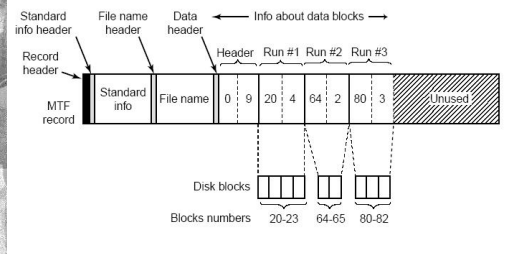


Fig. 11-36. An MFT record for a three-run, nine-block file.

[Tane 01]

13

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

Fig. 11-35. The attributes used in MFT records.

[Tane 01]

14

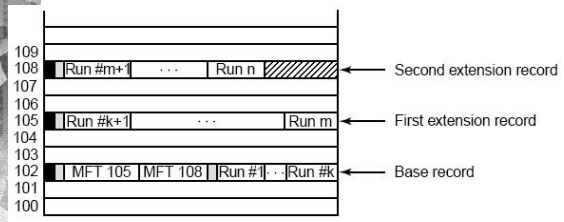
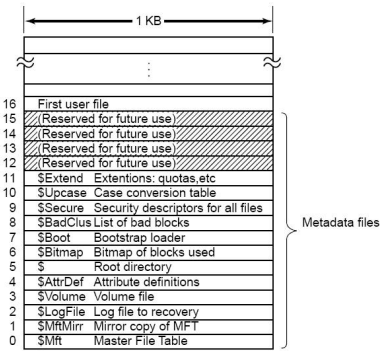


Fig. 11-37. A file that requires three MFT records to store all its runs.

[Tane 01]

15



[Tane 01] Fig. 11-34. The NTFS master file table.

16

TCP/IP concepts

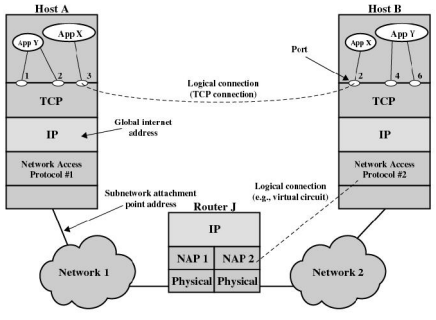


Figure 13.4 TCP/IP Concepts

17

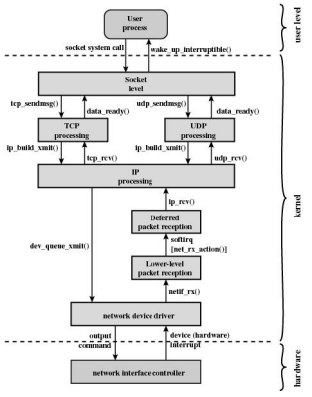
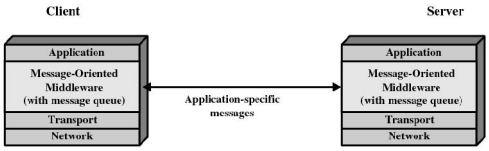


Figure 13.7 Linux Kernel Components for TCP/IP Processing

18

Process communication : message passing — Ch 14

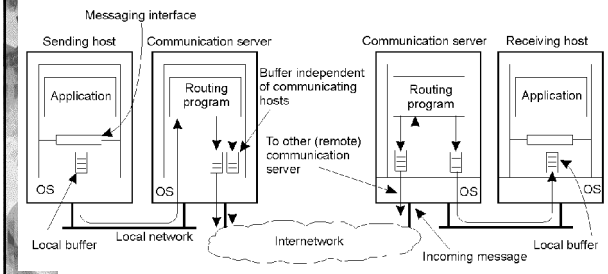
- Reliable / unreliable
- Synchronous / asynchronous
- Persistent / transient
- Binding
  - Based on network topology or message content
  - Static / Dynamic



(a) Message-Oriented Middleware

19

Message is buffered on the way



20

Esimerkki: header.h

```
/* Definitions needed by clients and servers.                */
#define TRUE             1
#define MAX_PATH       255   /* maximum length of file name  */
#define BUF_SIZE      1024   /* how much data to transfer at once */
#define FILE_SERVER    243   /* file server's network address */

/* Definitions of the allowed operations */
#define CREATE           1   /* create a new file            */
#define READ             2   /* read data from a file and return it */
#define WRITE            3   /* write data to a file         */
#define DELETE           4   /* delete an existing file      */

/* Error codes. */
#define OK               0   /* operation performed correctly */
#define E_BAD_OPCODE    -1   /* unknown operation requested  */
#define E_BAD_PARAM     -2   /* error in a parameter         */
#define E_IO            -3   /* disk error or other I/O error */

/* Definition of the message format. */
struct message {
     long source;            /* sender's identity            */
     long dest;              /* receiver's identity          */
     long opcode;            /* requested operation          */
     long count;             /* number of bytes to transfer  */
     long offset;            /* position in file to start I/O */
     long result;            /* result of the operation      */
     char name[MAX_PATH];    /* name of file being operated on */
     char data[BUF_SIZE];    /* data to be read or written   */
};
```

21

Esimerkki: server

```
#include <header.h>
void main(void) {
     struct message ml, m2;          /* incoming and outgoing messages */
     int r;                          /* result code                    */

     while(TRUE) {                   /* server runs forever            */
          receive(FILE_SERVER, &ml); /* block waiting for a message    */
          switch(ml.opcode) {        /* dispatch on type of request    */
               case CREATE:   r = do_create(&ml, &m2); break;
               case READ:     r = do_read(&ml, &m2); break;
               case WRITE:    r = do_write(&ml, &m2); break;
               case DELETE:   r = do_delete(&ml, &m2); break;
               default:       r = E_BAD_OPCODE;
          }
          m2.result = r;             /* return result to client        */
          send(ml.source, &m2);      /* send reply                     */
     }
}
```

22

Esimerkki: client

```
                                                    (a)
#include <header.h>
int copy(char *src, char *dst){                     /* procedure to copy file using the server */
     struct message ml;                             /* message buffer                          */
     long position;                                 /* current file position                   */
     long client = 110;                             /* client's address                        */

     initialize( );                                 /* prepare for execution                   */
     position = 0;
     do {
          ml.opcode = READ;                         /* operation is a read                     */
          ml.offset = position;                     /* current position in the file            */
          ml.count  = BUF_SIZE;                                                                 /* how many bytes to read*/
          strcpy(&ml.name, src);                    /* copy name of file to be read to message */
          send(FILESERVER, &ml);                    /* send the message to the file server     */
          receive(client, &ml);                     /* block waiting for the reply             */

          /* Write the data just received to the destination file.                             */
          ml.opcode = WRITE;                        /* operation is a write                    */
          ml.offset = position;                     /* current position in the file            */
          ml.count = ml.result;                     /* how many bytes to write                 */
          strcpy(&ml.name, dst);                    /* copy name of file to be written to buf  */
          send(FILE_SERVER, &ml);                   /* send the message to the file server     */
          receive(client, &ml);                     /* block waiting for the reply             */
          position += ml.result;                    /* ml.result is number of bytes written    */
     } while( ml.result > 0 );                      /* iterate until done                      */
     return(ml.result >= 0 ? OK : ml result);       /* return OK or error code                 */
}
```

Process communication: RPC

- Remote procedure call looks to the calling client as any other local procedure (or function or method) call.

(b) Remote Procedure Calls

24

RPC – Remote Procedure Call

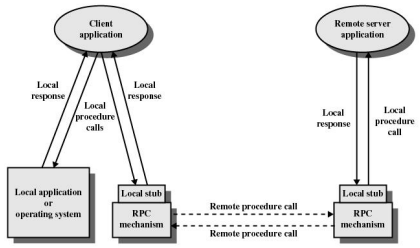
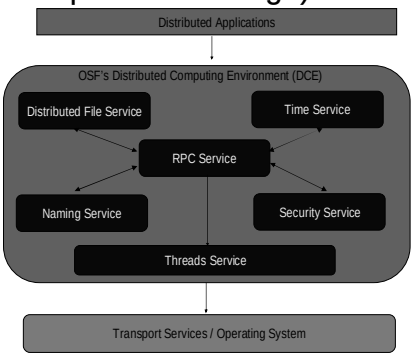


Figure 14.12 Remote Procedure Call Mechanism

25

DCE architecture (example of RPC usage)



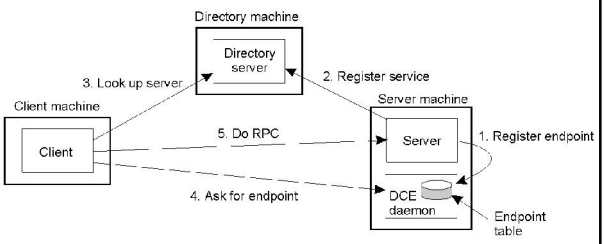
26

RPC elements

- RPC *stubs*
  - Hide the actual communication from the calling program
  - Marshal and unmarshal the parameters
- Transportation service
  - Needed to delivered the message containing the marshalled call
  - Can be provided by OS or middleware
- Name service: look up, binding
  - Contains interface definitions

27

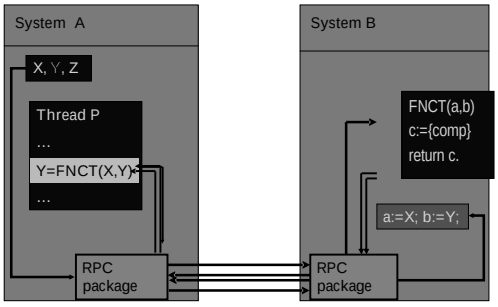
Binding a Client to a Server



Client-to-server binding in DCE.

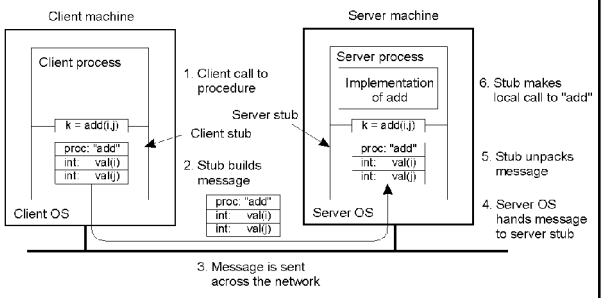
28

RPC: a Schematic View



29

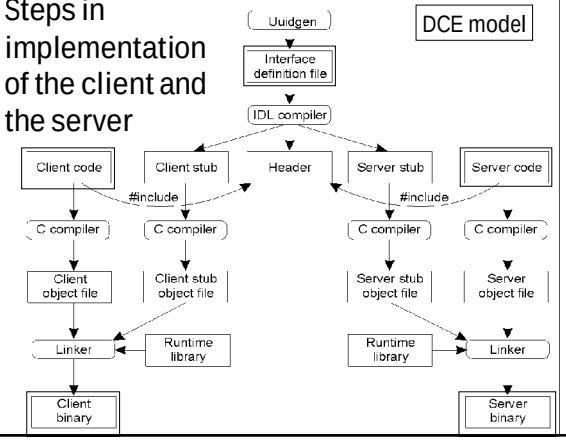
Passing value parameters in RPC



Steps involved in doing remote computation through RPC

30

Steps in implementation of the client and the server



Esimerkki: simple.idl

- IDL – Interface Description Language to describe the signature of the remote procedure
- IDL is used to automatically generate stubs
- Linux *rpcgen* uses RPC language instead of idl

```
/* SIMPLE.IDL */
[
uuid(004C4B40-E7C5-1CB9-94E7-0000C07C3610),
version(1.0)
]
interface simple
{
   void simple_operation(

      [in]  long x,
      [out] long *y
   );
}
```

32

Esimerkki: simple-client.c

```
/* SIMPLE_CLIENT.C */
#include <stdio.h>
#include "simple.h"

main(int argc, char *argv[])
{
idl_long_int x;
idl_long_int y;

if  (argc < 2 ) {   x = 1; }
    else {   x = atol(argv[1]); }

simple_operation(x, &y);        /* This is the Remote Procedure Call */

printf("The answer is: %ld.\n");
return(1);
}
```

simple.h is automatically generated By idl compiler from simple.idl

33

Esimerkki: simple-server.c 1/3

```
/* SIMPLE_SERVER.C */

#include <stdio.h>
#include <dce/rpc.h>
#include "simple.h"

#define ERR_CHK(stat, msg) if(stat != rpc_s_ok)\
   { fprintf(stderr,"Error: %s in file: %s at line %d.\n",msg,__FILE__, __LINE__);\
     exit(1); }

/***** Server Control *****/

main()
{
error_status_t       status;
rpc_binding_vector_t  *bindings;
unsigned_char_t       *name = "/.:/applications/simple";
```

34

Esimerkki: simple-server.c 2/3

```
rpc_server_register_if(simple_v1_0_s_ifspec, NULL, NULL, &status);
   ERR_CHK(status,"Could not register interface");
rpc_server_use_all_protseqs(rpc_c_protseq_max_regs_default, &status);
   ERR_CHK(status,"Could not use all protocols");
rpc_server_inq_bindings(&bindings, &status);
   ERR_CHK(status,"Could not get binding vector");
rpc_ns_binding_export(rpc_c_ns_syntax_default, name, simple_v1_0_s_ifspec, bindings,
                NULL, &status);
ERR_CHK(status,"Could not export bindings");

rpc_ep_register(simple_v1_0_s_ifspec, bindings, NULL, NULL, &status);
   ERR_CHK(status,"Could not register endpoint");

printf("Listening for requests\n");

rpc_server_listen(rpc_c_listen_max_calls_default, &status);
}
```

Esimerkki: simple-server.c 3/3 - the actual procedure to be called

```
/***** Server Operation *****/

void simple_operation(idl_long_int x, idl_long_int *y)
{
*y = ++x;
}
```

36