Czech University of Agriculture in Prague

Faculty of Economics and Management
Department of Information Engineering

Operating Systems

Doc. Ing. Arnošt Veselý, CSc.

Prague 11.2.2005

Contents

CONTENTS..3

OPERATING SYSTEM (OS)...4

DESIGN OF THE UNIX OS (BASIC CONCEPTS)....................................11

UNIX FILESYSTEM..26

UNIX PROCESSES..42

UNIX SYSTEM CALLS FOR PROCESS CONTROL................................54

UNIX FILE MANAGER SYSTEM CALLS..69

COMMUNICATION SUPPORT..79

REFERENCES..97

Operating system (OS)

OS consists of:

1. kernel
2. system programs

Kernel tasks:

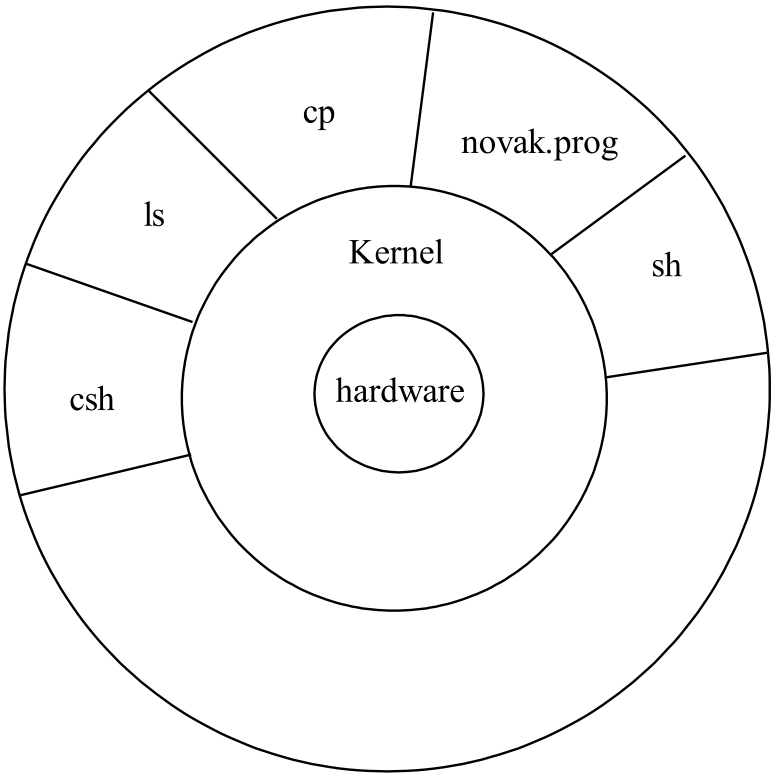
1. Creates processes and controls their execution
2. Facilitates communication among processes
3. Provides means for work with I/O devices
4. Part of the kernel is a file manager, which organizes disc data into files and directories
5. Overviews the system and records logs and different statistics

Unix consists of:

Kernel

System programs and libraries:

- shells(sh, bash, ksh, csh, tcsh)
- programs for file and directory manipulation (ls, cp, rm, tar, etc.)
- programming support (compilers, libraries)
- communication support (telnet, ftp, rlogin etc.)
- graphical interfaces (X-Windows, Open Windows)



Basic concepts

Program: executable file on disc

Process: running program

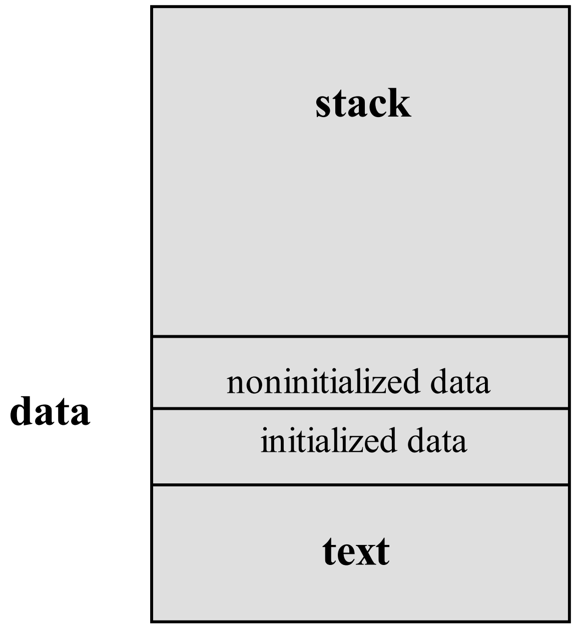


Fig. Process structure

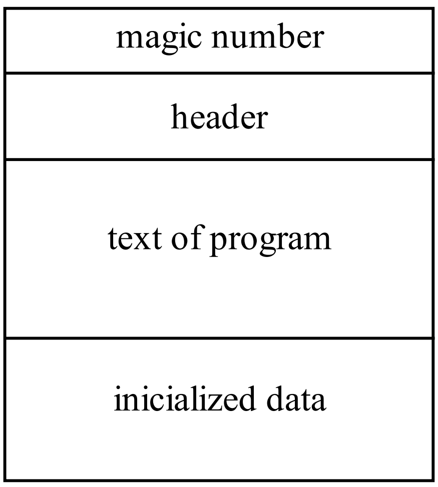


Fig. Structure of a target program on disc.

Basic process states

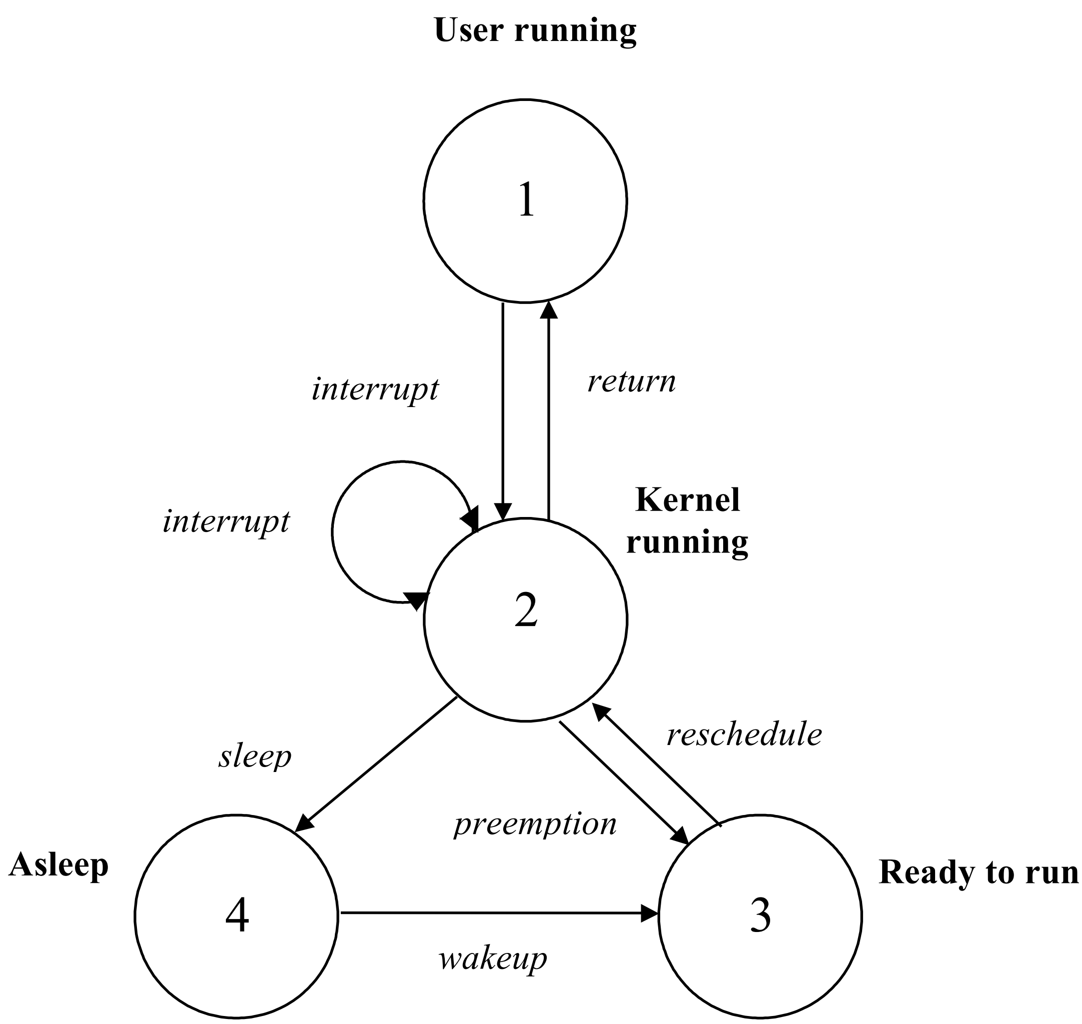


Fig. Basic process states

Process is asleep in memory (blocked) if it has not a resource, which is indispensable for its farther correct execution

OS activity after interrupt

external interrupt: I/O devices

inner interrupt:

- processor running errors (i.e. overflow, dividing by zero)
- running instruction of inner interrupt

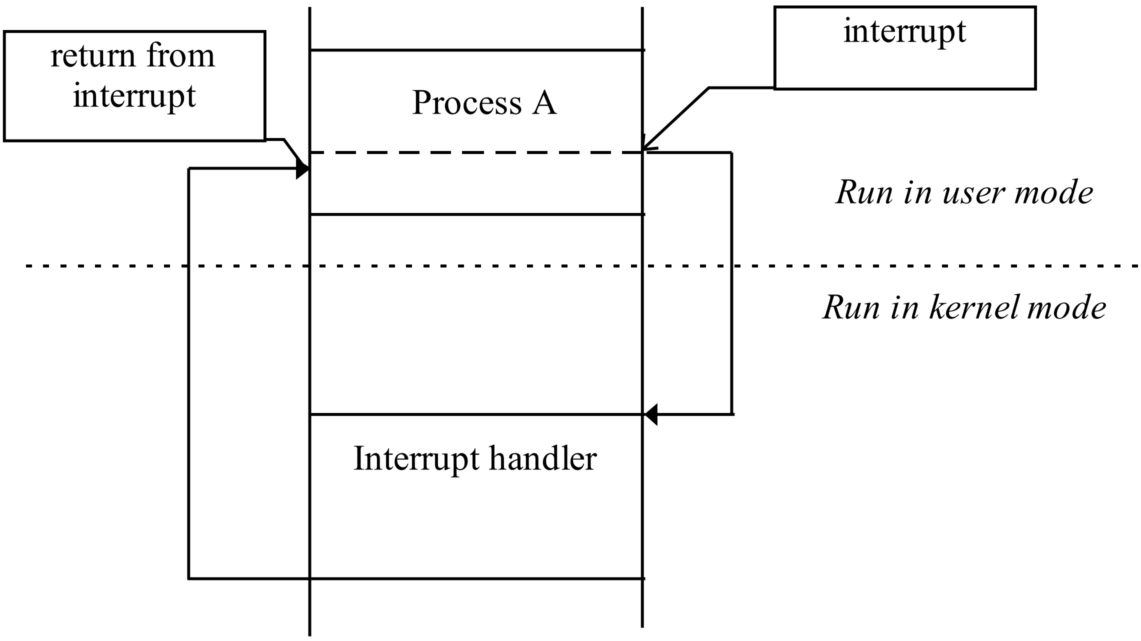


Fig. Process transition from user to kernel mode runing

System services (calls)

Every OS provides services to processes. System services can be realized by a subroutine call or internal interrupt.

1. System services in Unix are realized by means of internal interrupt.
2. System service is a function that contains instruction for internal interrupt. As soon as this instruction is executed the process changes the mode of running: it passes from user to kernel running. In kernel mode process executes code of the kernel (code of the system call).
3. System calls cannot be programmed in C language; they must be programmed in assembler.

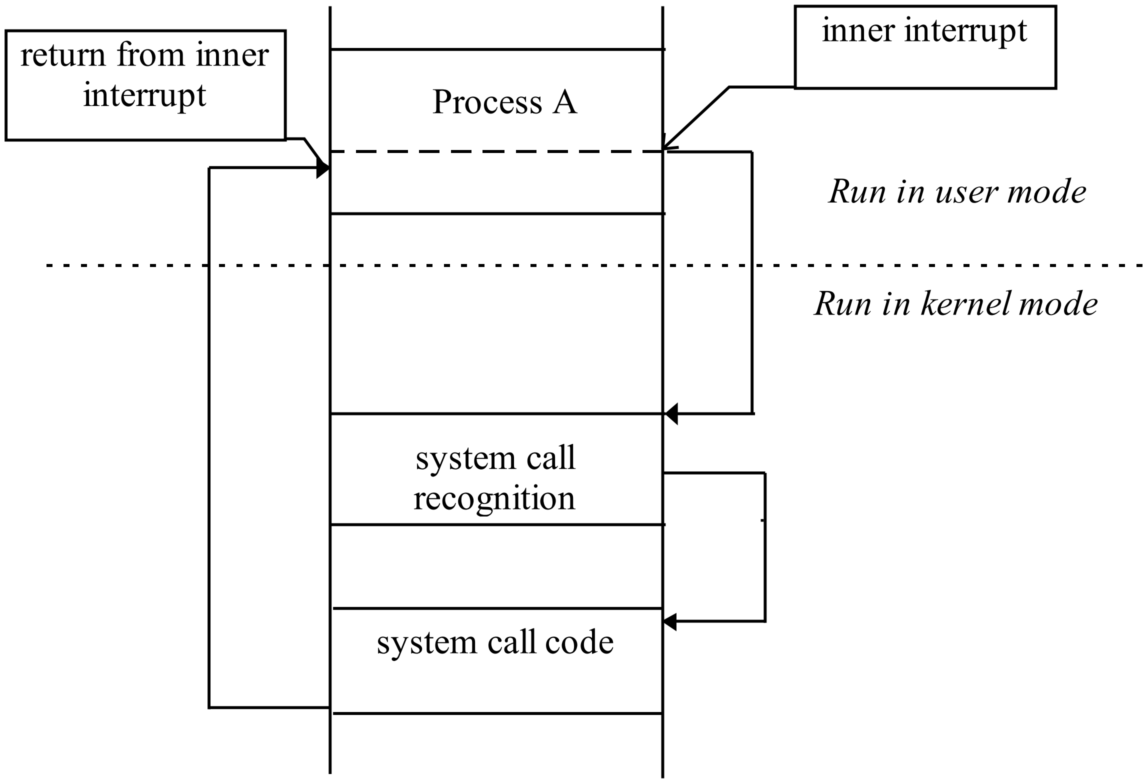


Fig. Process transition from user to kernel mode runnig

Preemption

Process can run continuously only for limited time interval (time quantum). After this interval is expired, kernel stops the process and puts it into ready to run state. Then scheduler chooses another process for running. The former process is said to be preempted.

The kernel is not preemptive: If a process is running in kernel mode, it cannot be preempted. (The reason is to exclude a possibility of time dependence of processes (race conditions)). Process running in kernel mode at first finishes kernel running (system service) and when it returns to user running, the control is given to scheduler and a new process could be scheduled.

Context switch: If another process is scheduled, kernel must swap user and system contexts of old process for user and system contexts of new one.

External interrupts

External interrupts are executed in the context of the interrupted process. Only system context of interrupted process must be saved and after return from interrupt it must be restored.

When process is running in kernel mode, some external interrupts have to be forbidden if race condition could occur.

Example (Unix)

Process A executes system service *read()* (reading from disc)

1. *read()* generates internal interrupt. Code of system call starts to be executed.
2. If asked data are not in disc cache in main memory, interrupt handler starts I/O transfer. Kernel puts process A into asleep state.
3. Process B is scheduled and begins to run.
4. Transfer from disc is finished. Disc module interrupts processor.
5. Interrupt is allowed. Disc interrupt handler is executed. Execution of the handler code is done in context of process B. At the end of its running interrupt handler puts process A into ready to run state.
6. Return from interrupt. Process B continues to run.
7. Time quantum for process B has expired and B is preempted. Process A is scheduled.

OS information about processes

OS stores information about processes to be able to control their run.

Unix stores important information about processes into:

1. process table
2. u-area

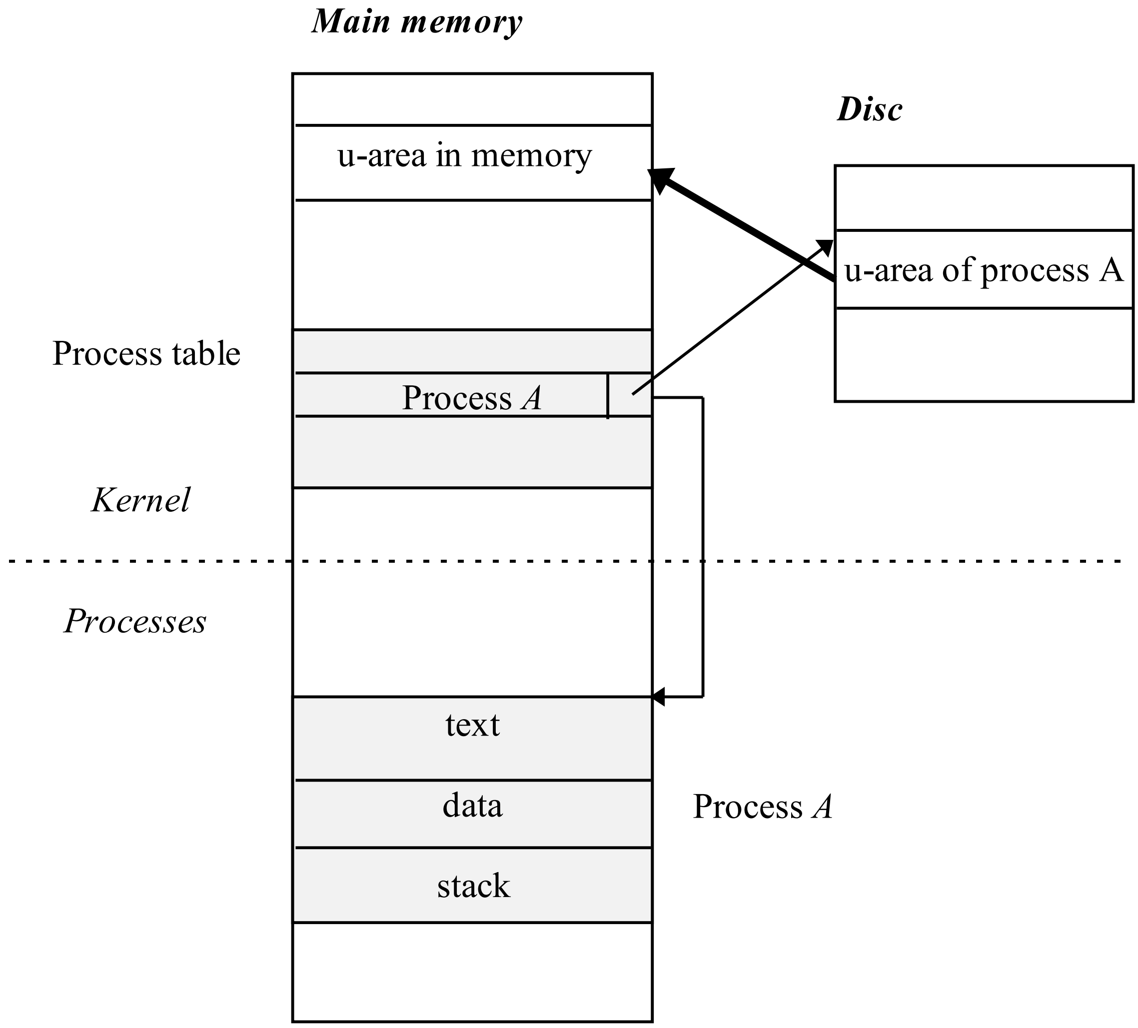


Fig. Process table and u-area. When a process is scheduled, its u-area is copied into main memory.

Context of a process

User context: text, data, stack, content of user registers

Systems context: process table, user’s area (u-area), content of system registers, content of system stack, page table

Design of the Unix OS (basic concepts)

Users

User has its username and *UID* (user identification)

System identifies users according their *UID*

Superuser is user who has *UID*=0. Superuser has all privileges to all files and all processes (with little exceptions)

Users are organized into groups:

Berkeley Unix: Each user can be at maximally member of 16 groups simultaneously

System V: Each user can be at certain moment only in one group. But he can change the group by command **newgrp**.

Basic information about users and groups OS maintains in two files:

`/etc/passwd` and `/etc/group`

username:encrypted password: UID:GID : Note : Home directory:Login shell

```
..............................................................................................................
novak: 1234567890ABC :100:10:Student:/home/novak:/bin/csh
..............................................................................................................
```

Fig. User table `/etc/passwd`

groupname : password : GID : list of users

```
..............................................................................................
 studenti:             : 10 :
ukol1   :123456789ABC: 21 :
ukol2   : *            : 22 :novak,novy,benes
```

Fig. Group table `/etc/group`

Files and directories

1. Files are organized into filesystems. Directories are files of special type. By means of directories files are organized into tree structure. Root of the tree structure is denoted by / .
2. All files and directories have their user owner and group owner. The first user owner of a file is the user who created it. Similarly the first group owner of a file is the group who created it. Further the ownership can be transferred to another user or group by a system call or command.
3. Access rights control the access to files.
4. Access rights are **r**, **w**, **x**. Their meaning is

regular files:

1. **r** reading
2. **w** writing
3. **x** executing

directories:

1. **r** reading content of the directory
2. **w** create and delete files and subdirectories in the directory
3. **x** enter the directory

Access rights are specified for: **individual owner**, **group owner**, **others**
Specification is written as a sequence of 9 characters:

individual owner

others

r w x r w x r w x

group owner

If some right is not given, instead of character **r**, **w**, **x** the character **-** is used

Example

r w - r - - r - -

Login into system

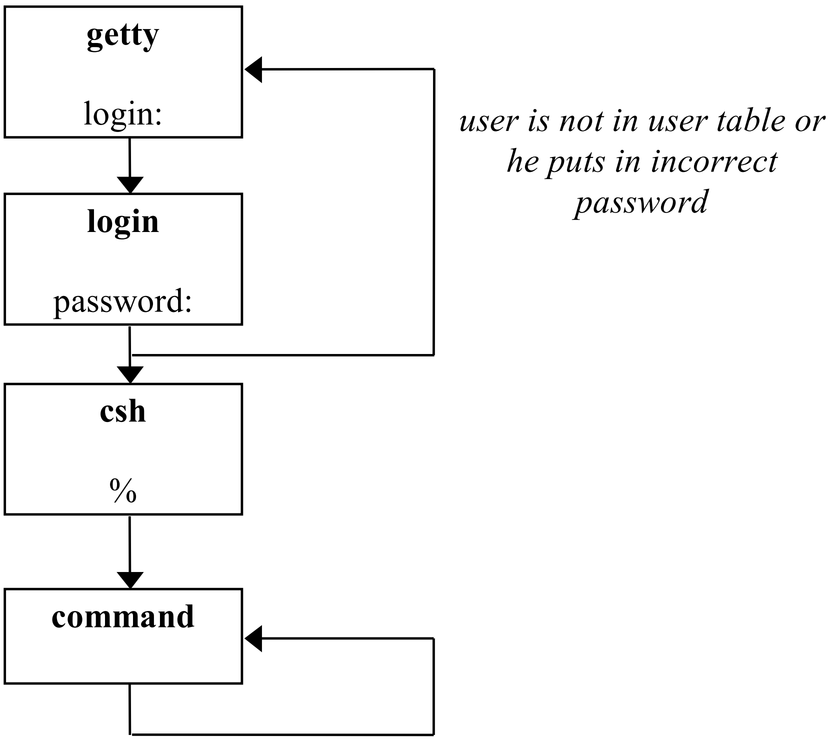


Fig. User login

As everybody can read file `/etc/passwd`, short passwords of other users can be cracked by a simple program.

Therefore encrypted user passwords some OS store in the file `/etc/shadow` that users cannot read

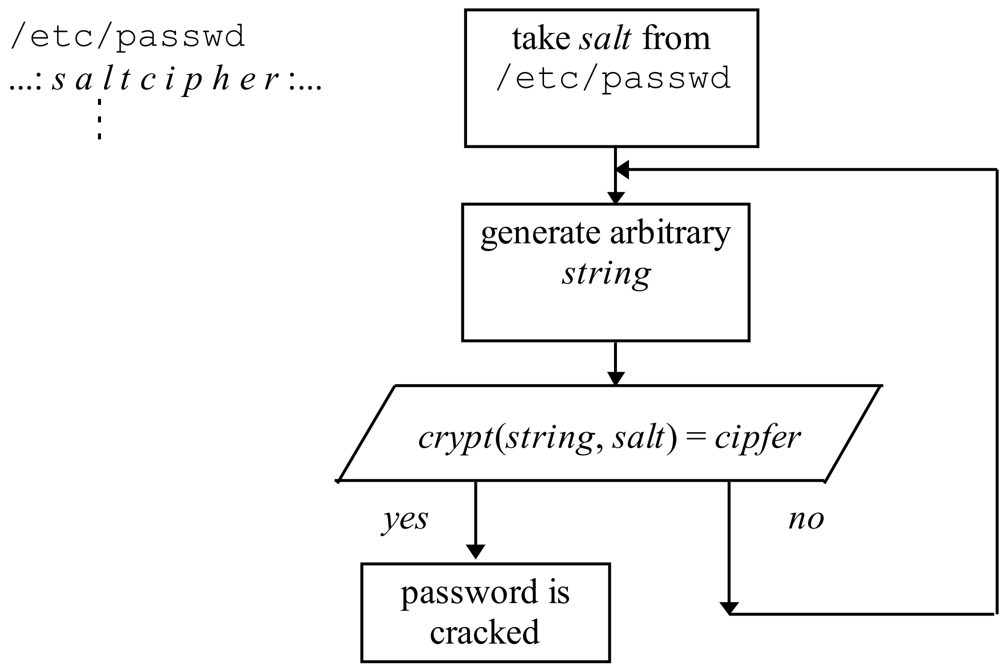


Fig. Principle of a program for password cracking

Compilation of programs

Compilation is done by program **cc**.

Program **cc** makes preprocessing and compilation. Then it calls linkage editor (loader) **ld**.

During linking process **ld** looks only through library **libc.a**. It is possible to compel **ld** to look also through other libraries (by setting running parameters **l** or **L**).

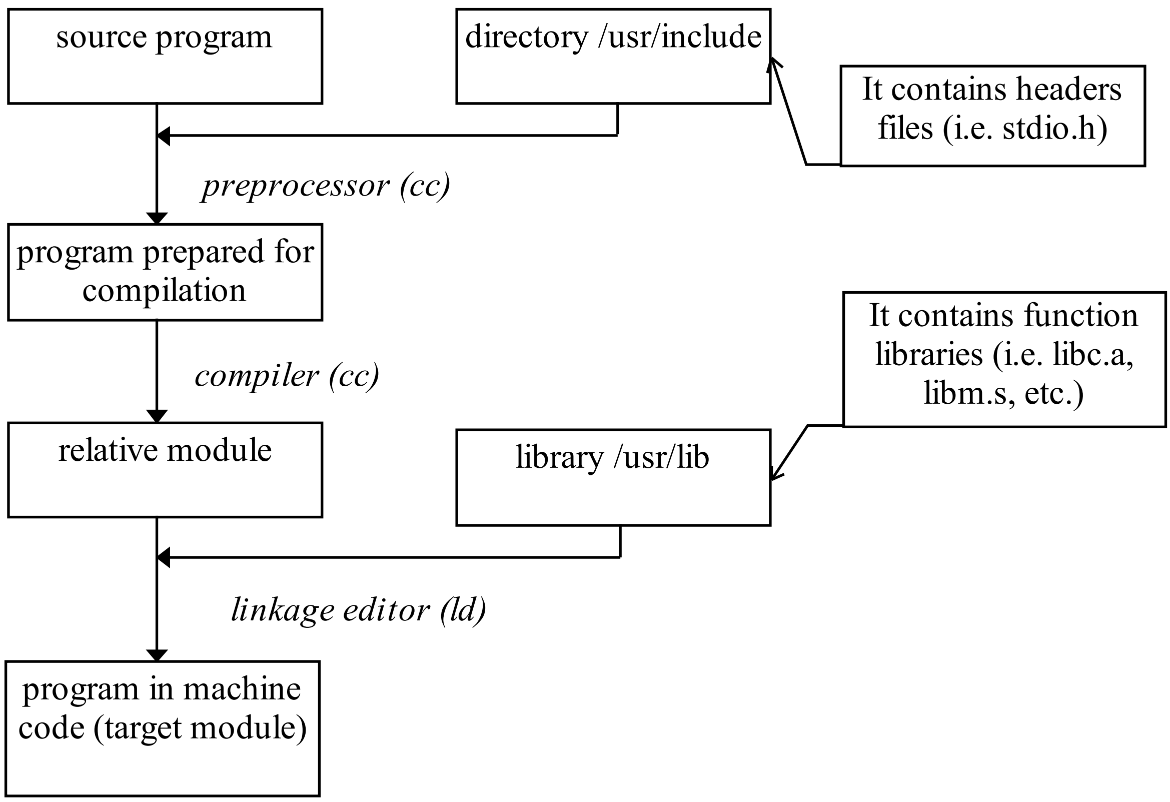


Fig. Compilation of a program

Compilation commands

```
cc prog.c -lm
cc prog.c  -L/usr/lib/libm.a
cc prog.c -lm -o prog.exe
cc prog.c funkce1.c funkce2.c error.c
cc prog.c funkce1.o funkce2.o error.o
cc prog.c -L/home/novak/lib/lib.a
```

Example of a program

```
/* cat: version 1 Print file on terminal */
#include <fcntl.h>
#include <stdio.h>
main(argc,argv)
     int argc;
     char *argv[];
{    int d,count;
     char buf[1024];
     if(argc != 2){
         printf("error: cat must have one
                      parametr\n");
         exit(1);
     }
     d = open(argv[1],O_RDONLY);
     if(d == -1){
         printf("cannot open file %s\n", argv[1]);
         exit(1);
     }
     while(( count = read(d,buf,sizeof(buf))) > 0)
         write(1,buf,count);
     return 0 ;
}
```

Translation and running the program

(program is in the file `mycat.c`)

```
$cc mycat.c –o mycat
$mycat /etc/passwd
```

Program execution

Before execution, memory area where noninitialized data reside is zeroed. The area where stack resides is not reset.
Static initialized variables are stored in initialized data area.
Static noninitialized variables are stored in noninitialized data area.
On stack there are stored:
automatic variables, call parameters and return addresses

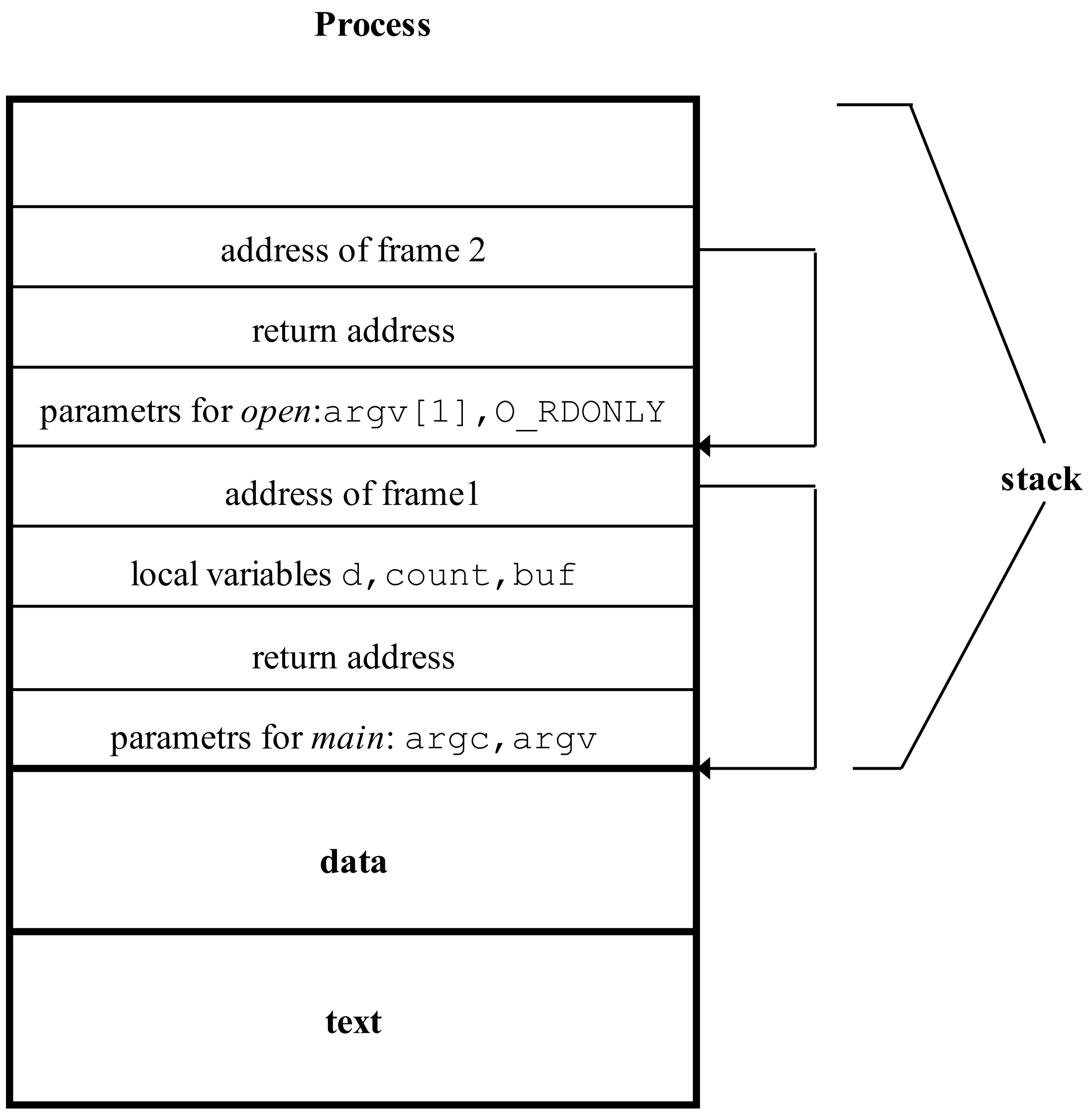


Fig. Execution of program: content of stack after calling *open()*

Process creation

System call *fork()*

1. OS creates a copy of a calling process.
2. Except *PID* and return value of the call, the child process inherits all user context of its parent.
3. Return value in parent process is child *PID* .
4. Return value in child process is 0.

Example

```
main()
        {
            int r;
            if((r=fork())== 0)
                printf("I am child");
                            /* child */
            else
                printf("I am parent");
                            /* parent */
        }
```

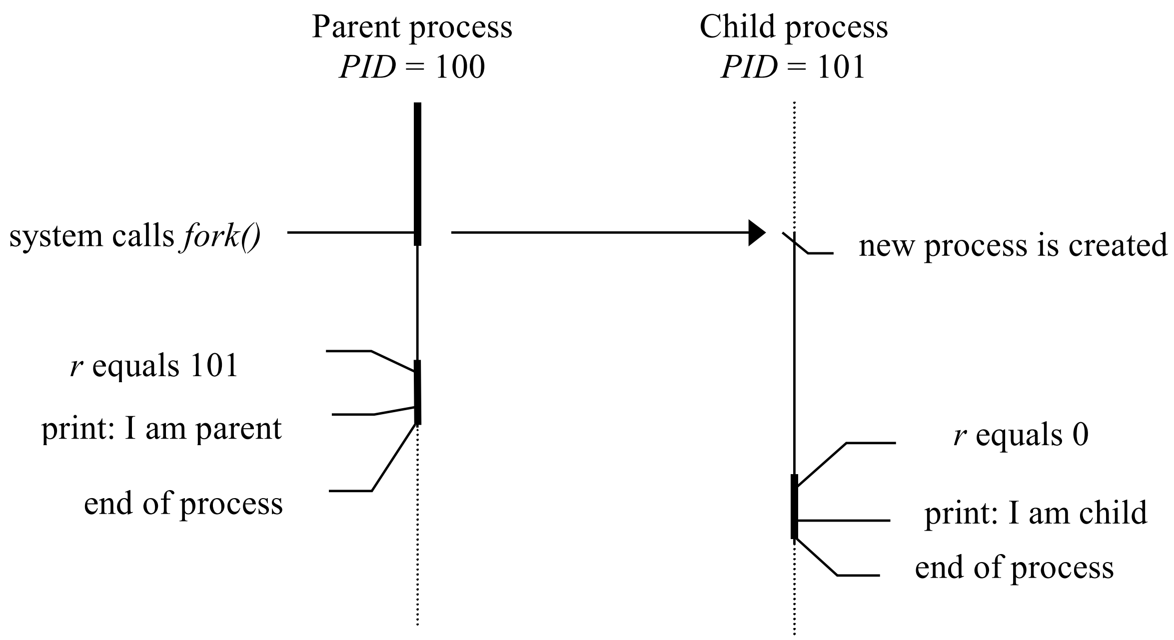


Fig. After system call *fork()* a new process is created

Example

The following program was executed. How many processes remained in system in asleep state? (3)

```
main()
{    (if fork() == 0)
          pause();
     fork();
     pause();
}
```

Process text change

System call *execl()* invokes another program.

OS changes text of the calling process.

Example

```
#include <stdio.h>
main()
{
        if(fork()==0)
            execl("/bin/date", "date",NULL);
                        /*child*/
        wait(NULL);
                        /*parent*/
        printf("child process finished\n");
}
```

1. Child process will be further controlled by the text stored in the file `/bin/date`
2. Parent process is waiting for child process exit (it is in asleep state). After the child exit OS will awake the parent. If in *wait()* call is supplied a pointer to a variable, OS will put exit status of the child into this variable.

Owner of a process

Processes are not anonymous. With each process the following ownership parameters are associated:

- real user (*UID*)
- effective user (*EUID*)
- real group (*GID*)
- effective group (*EGID*)

Real and effective ownership

- After login *UID* and *GID* of login shell are set according to the user table.
- User processes inherit their *UID* and *GID* from login shell.
- *UID* and *GID* are all the time the same.
- At the beginning of a process *EUID* and *EGID* are equal to *UID* and *GID*.
- During process run and under special circumstances they may be changed (e.g. if the process executes system call `execl(/path/file,”file”,NULL)` and `file` has set s-bit).

Access rights are checked against effective owners.

Signals

There are 32 signals (in older Unix systems only 16) that one process can send to another process.

system calls

kill()	for sending a signal
signal()	for catching a signal

command:

kill [-*signal*] *PID* . . . .

sends a signal number *signal* to process *PID*

OS activity during system call *kill()*:

- It writes down into process table the number of the sent signal
- If the process to which the signal has been sent is in asleep state, OS will awake it

Activity of a process which the signal has been sent to:

- Signal is processed before process run
- Processing of signal depends on whether the reaction on coming signal has been specified before (by a system call *signal()*)

If the reaction has not been specified, default action is taken.

For most signals default action means abortion of the process (exception: for signal SIGCHLD(17) and SIGCONT (18) default action means to do nothing).

If the reaction has been specified, the action is taken according to this specification. Specification could be:

- ignore the sent signal
- run code of signal handler

Example

Initial state:

Process *A* is running

Process *C* is asleep

Process B sends:

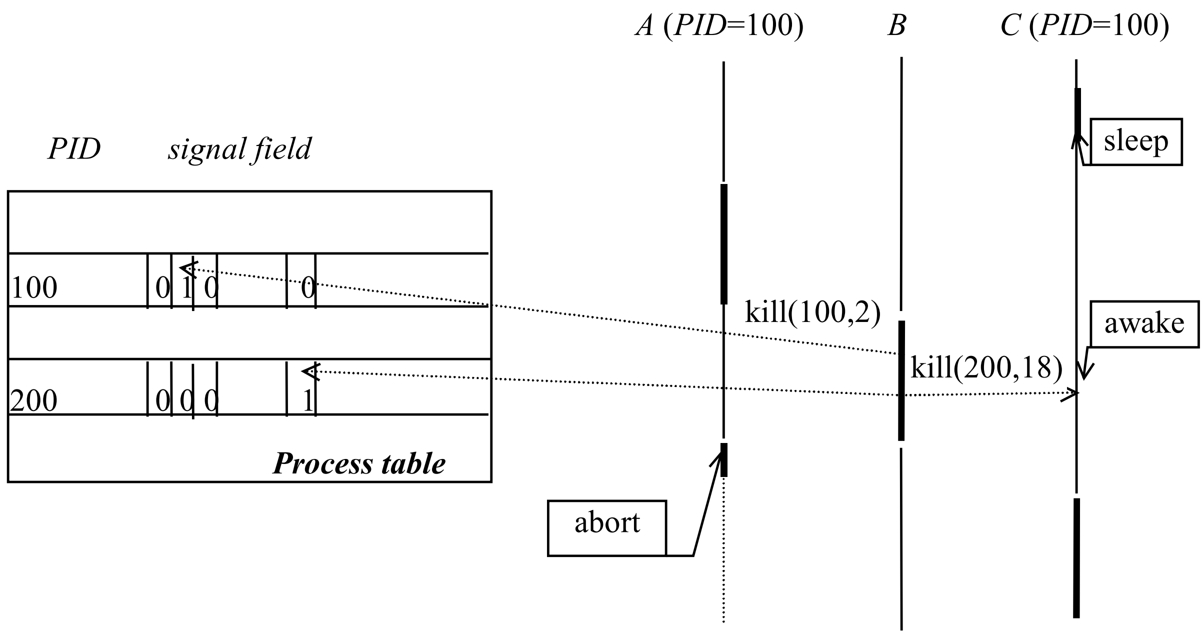
signal 2 to process *A*

signal 18 (`SIGCONT`) to process *C*

Result:

Process *A* is aborted

Process *C* is awaken and it will be scheduled and run



Inputs and outputs

Data input into a process and data output from a process are sequences of bytes (called streams).

Before working with a stream:

- process must open it using system call ***open()***
- if opening is successful, ***open()*** will return small positive number called descriptor
- further the opened stream is identified by the returned descriptor

Process can read opened file by system call

read(descriptor, read_buffer, number_of_bytes)

Process can write to opened file by system call

write(descriptor, write_buffer, number_of_bytes))

Reading and writing a file

- The reading and writing start from a current position.
- The current position is during reading and writing updated.
- It can be set by system call *lseek()* to arbitrary value, and that means, that random access to the bytes in streams is possible.

Control files of I/O devices

Physical devices are represented by their control files, stored in `/dev`

```
ls -al  /dev | more
```

```
brw-rw-rw- 1 root   floppy 2,0 Oct 20 12:20 fd0H1440
brw-rw---- 1 root   disk   3,0 Oct 20 12:20 hda
brw-rw---- 1 root   disk   3,1 Oct 20 12:20 hda1
brw-rw---- 1 root   disk   3,2 Oct 20 12:20 hda2
crw-rw---- 1 daemon daemon 6,0 Oct 20 12:20 lp0
crw-rw-rw- 1 root   disk   9,0 Oct 20 12:20 st0
crw-rw-rw- 1 novak  users  5,0 Oct 20 12:20 tty
crw--w--w- 1 novak  users  4,1 Oct 20 12:20 tty1
```

Major number of a device: identification of handler

Major number is defined during configuration of OS (before compilation)

Minor number of a device: serial number of device

By minor numbers OS discriminate among devices controlled by the same handler

Handlers: with blocked data transfer (disc)

with character transfer (printer, terminal)

Direct manipulation with I/O device

It is possible to work with devices by means of their control files

```
cp  /etc/hosts   /dev/tty1
cat /etc/group   >/dev/tty
cp /etc/hosts   /dev/fd0H1440
```

Reading and writing control file means jump to device handler, identified with major number.

Control file does not contain any data. But it has its i-node.

Start and shutdown of Unix system

OS can run in one of these ***regime levels:***

0 System halt
1 Single user
2-5 Multiuser
6 Restart

Before OS begins to run at certain level, the process **init** will run level specific start scripts.

Which start scripts will run at particular level is defined by the content of `/etc/inittab` file

identificator: level: action : command

```
rc : 2345 : wait : /etc/rc.d/rc.2
net : 345 : wait : /etc/rc.d/rc.net
..........................................
c1 : 2345 : respawn : /sbin/getty 9600 tty1
```

Fig. File `/etc/inittab`.

line rc:

before run at levels 2, 3, 4 or 5 OS will start script `/etc/rc.d/rc.2`

`wait` means that OS will wait till the script finishes

line c1:

before run at levels 2, 3, 4 or 5 OS will start program `/sbin/getty` with run parameters 9600 and tty1

`respawn` means that if getty finishes, OS will start it again

One line of `/etc/inittab` defines implicit run level. Superuser can change run level by the command.

Shutdown commands:

shutdown -h +*time message*

shutdown -h now

Unix filesystem

Files are organized into filesystems. Several filesystems can reside on one disc. The whole filesystem must reside on the same disc.

File types:

regular files	`-`
directories	`d`
control files of block I/O devices	`b`
control files of character I/O devices	`c`
link files	`l`

Directories: special files that enable to organize files into tree structure

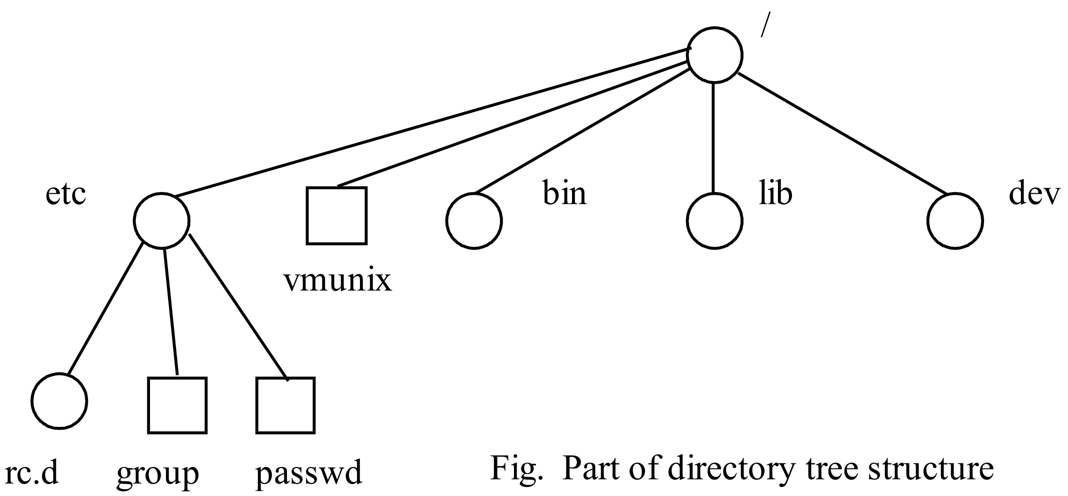


Fig. Part of directory tree structure

Denotation conventions:

- parent directory `..`
- root directory `/`
- current (working) directory `.`
- home directory `~`

Identification of files:

complete path `/usr/bin/ls`

incomplete path `bin/ls` (starts in current directory)

File system organization (Unix SYSTEM V)

Bootblock contains boot program
Superblock contains system information about the filesystem:

- size of filesystem
- size of i-node area
- number of free blocks
- number of free i-nodes
- first block of free blocks list
- first block of i-nodes free list

i-node area:

- all system information about file is stored in i-node
- in a directory there are stored only pairs (*filename*, *i-node number*)
- root directory has i-node 2

Data area contains:

- data of files and directories
- address blocks
- free block list
- free i-nodes list

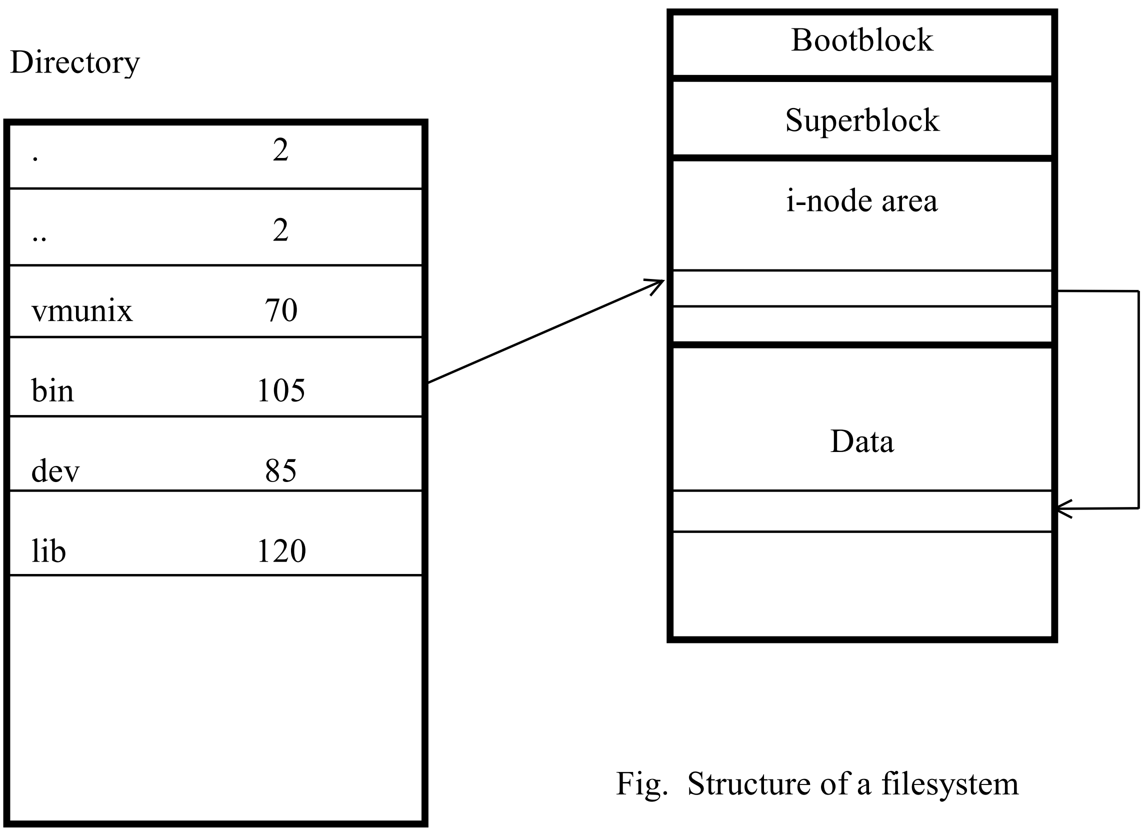


Fig. Structure of a filesystem

i-node items

- *File type*
- *Access rights*
- *User owner* (*UID*)
- *Group owner* (*GUI*)
- *Time of last write into file*
- *Time of last access to file (reading or executing)*
- *Time of last modification of i-node*
- *Size of file in bytes*
- *Number of disc blocks necessary to store file on disc*
- *Number of references (hard links, references) to file*
- *Addresses of file data blocks*

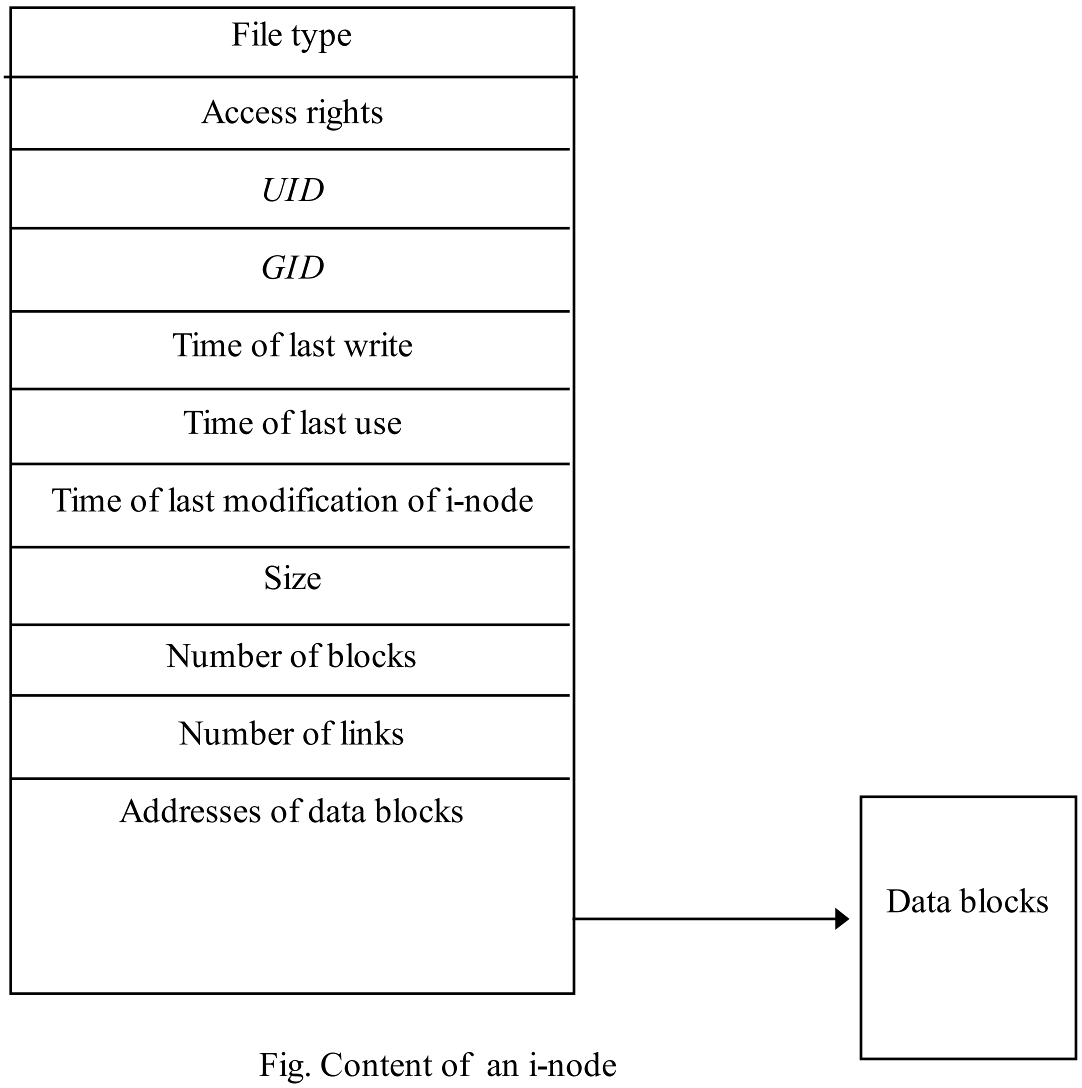
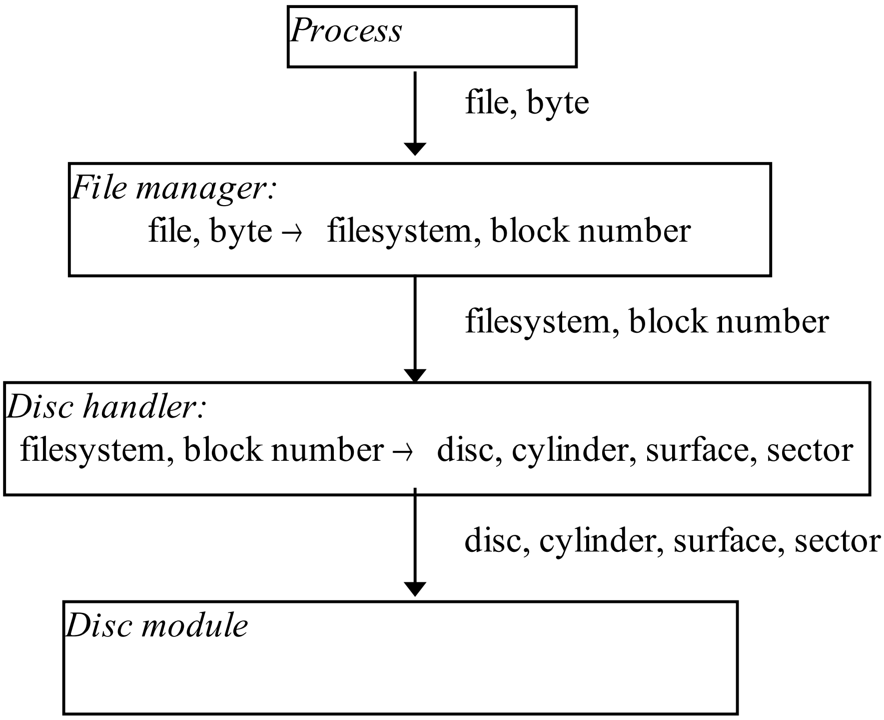


Fig. Content of an i-node

Allocation of file data on disc

- Block (or cluster) is one or several sectors on disc.
- Numbers of blocks are local in a filesystem.



- File data are stored in blocks.
- Files must begin at the beginnings of blocks.
- A choice of big blocks may cause wasting of disc space, especially if there are many small files.

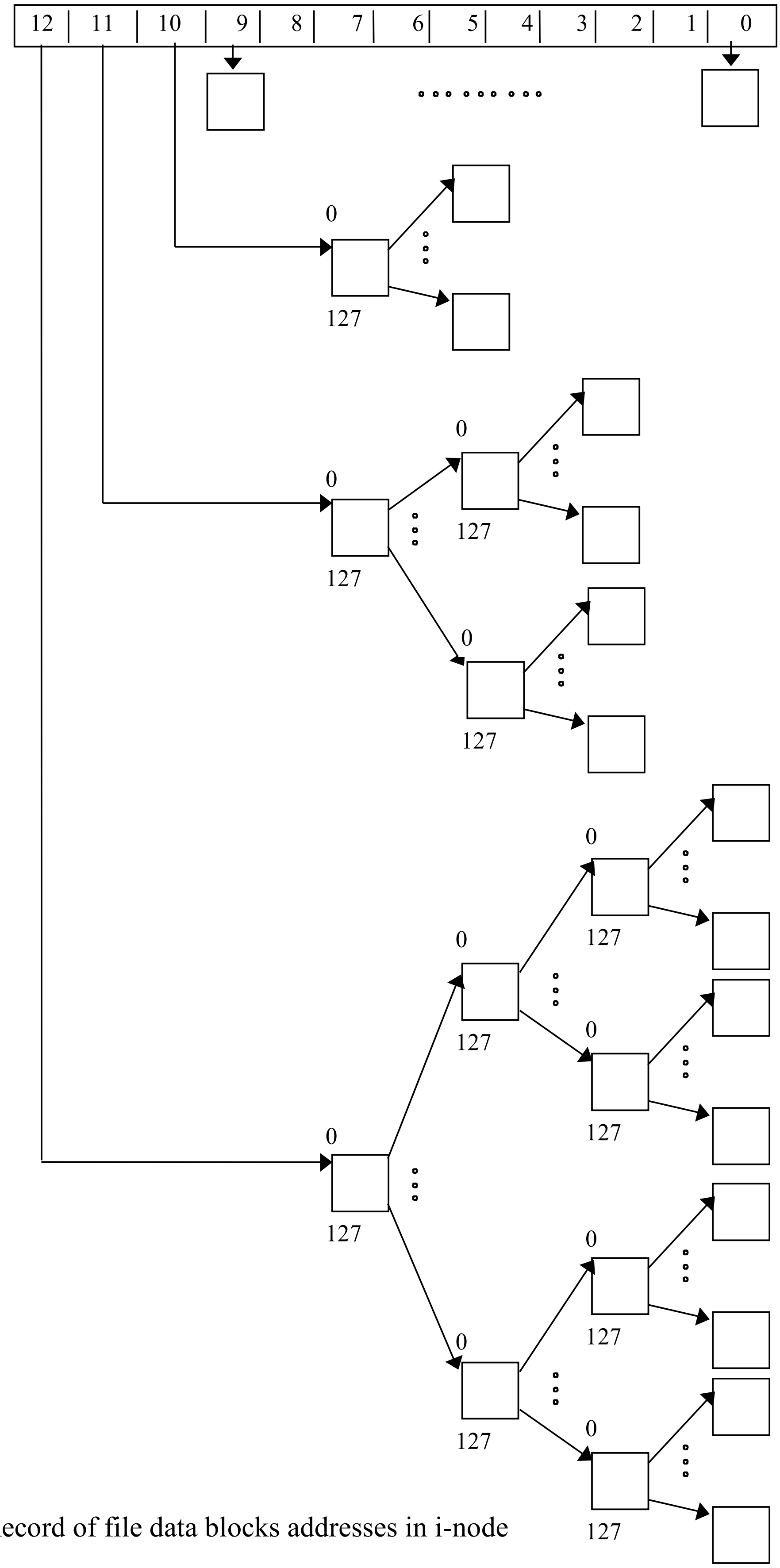


Fig. Record of file data blocks addresses in i-node

Maximal size of a file

block is 512 B, block number is in 4B:

$max\_size = 10\times512 + 128\times512 + 128\times128\times512 + 128\times128\times128\times512 \cong 1GB$

Change of ownership:

chown *new_username*[.*new_groupname*] *filename . . .*

chgrp *new_group filename . . .*

Hard links (references to a file)

system calls:

link()	creates a hard link
unlink()	deletes a hard link

OS activity during *link()* system call

1. It creates new directory item of a file (hard link to a file)
2. Number of references to file is increased by 1

OS activity during *unlink()* system call

1. Directory item of a file is removed
2. The number of hard links to file i-node is decreased by 1. If the new number of hard links is less than 1, OS puts i-node on i-node free list and data blocks of the file on data block free list (the file is deleted).

command

ln *filename link*	creates a hard link *link* to a file identified by hardlink *filename*
rm *filename . .*	removes hardlinks *filename . .*
mv *from to*	moves and possibly renames hardlink *from* to hardlink *to*

Program rm

Program **rm** uses unlink system call for deleting files.

Because numbers of i-nodes are in each filesystem local, it is not possible to create a hardlink that points outside the filesystem.

It is not possible to create a hard link to a directory because of danger of looping.

Example

```
cd /usr/john
ln /etc/passwd data
ln /etc/passwd my.data
cp /etc/passwd data.copy
```

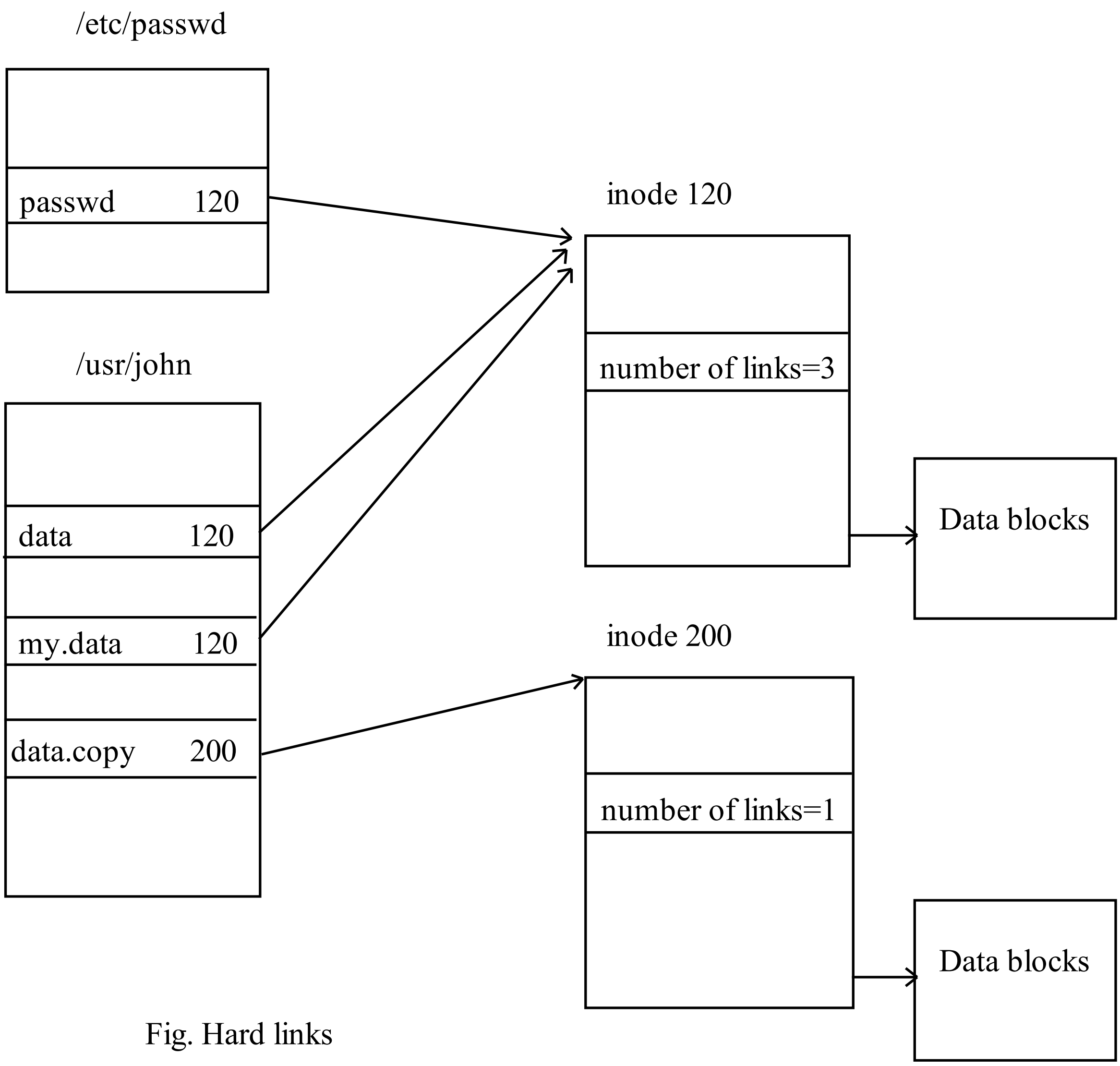


Fig. Hard links

Soft Links

Soft link is a special file (type *l*) containing a path.

command for creating a link file (or link):

ln -s *path link*

link is a path that determines where the link file will be created

path identifies file which the link will refer to

The content of link file will be *path*

How OS works with soft links during file search

Link file contains complete path:

Example:

- suppose OS is looking for the file `/path1/link/path2`
- the content of link file `link` is complete path `/path3`

As soon as OS comes to `link`, it starts to search file `/path3/path2`

Link file contains incomplete path:

Example:

- suppose OS is looking for the file `/path1/link/path2`
- the content of link file `link` is incomplete path `path3`

As soon as OS comes to `link`, it starts to search file `./path3/path2`

Creation of a directory

Command

mkdir *directory . . .*

System call

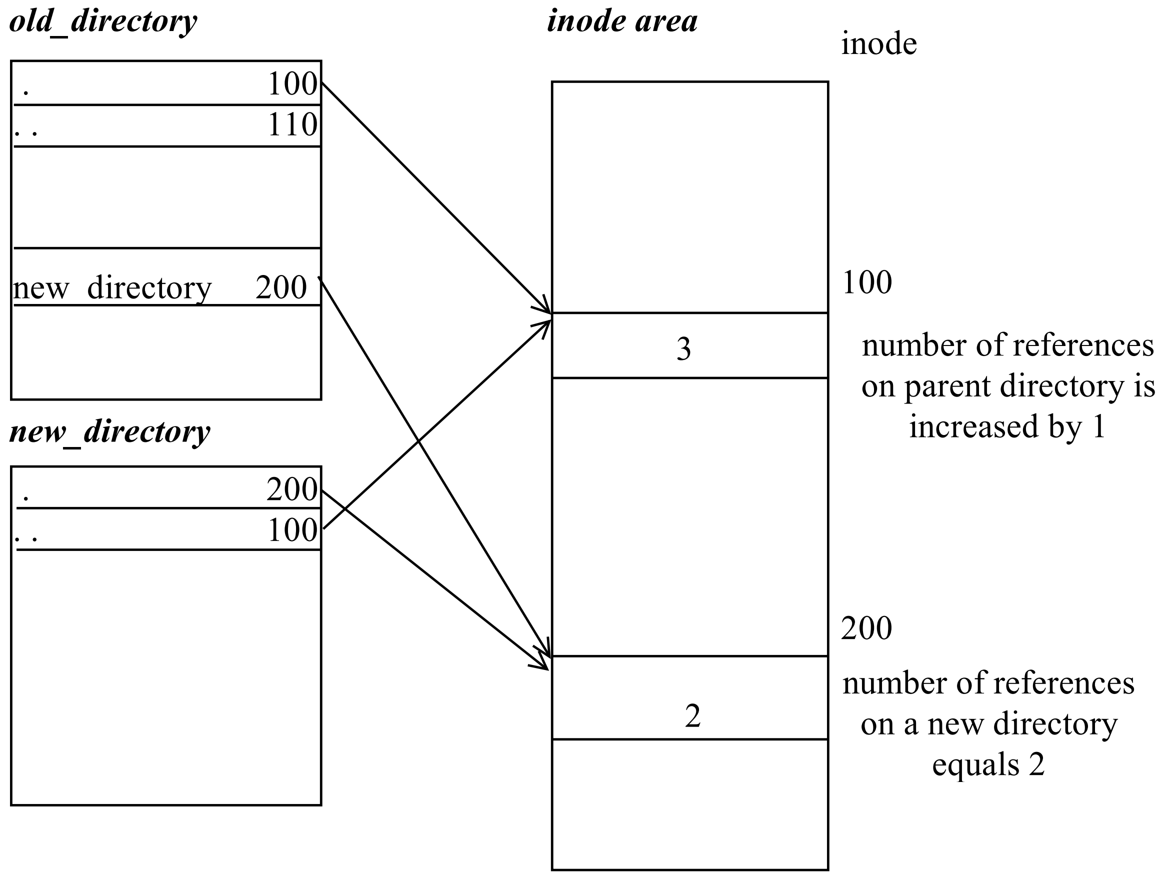
mknod()

OS activity during system call *mknod()*

- New item in parent directory is created
- Free i-node is allocated for the new directory
- New directory data structure is created

Example

A new directory *new_directory* is created in a directory *old_directory* (i-node=100)



- At the beginning there are 2 references on each newly created directory.
- If a new subdirectory is created, number of references to its parent directory is increased by 1.

Access rights

right	*file*	*directory*
r	read from file	read content of a directory
w	write into file	create, delete or rename files in a directory
x	execute file	enter into a directory

rights are specified for:

individual owner			***group owner***			***others***		
r	**w**	**x**	**r**	**w**	**x**	**r**	**w**	**x**

notation:

rwxrwxrwx or **777**

Example:

rw-r--r-- or **644**

Evaluation of access rights

access rights are evaluated according to the effective owner of a process (*EUID* or *EGID*)

UIDO=individual owner of file

GIDO= group owner of file

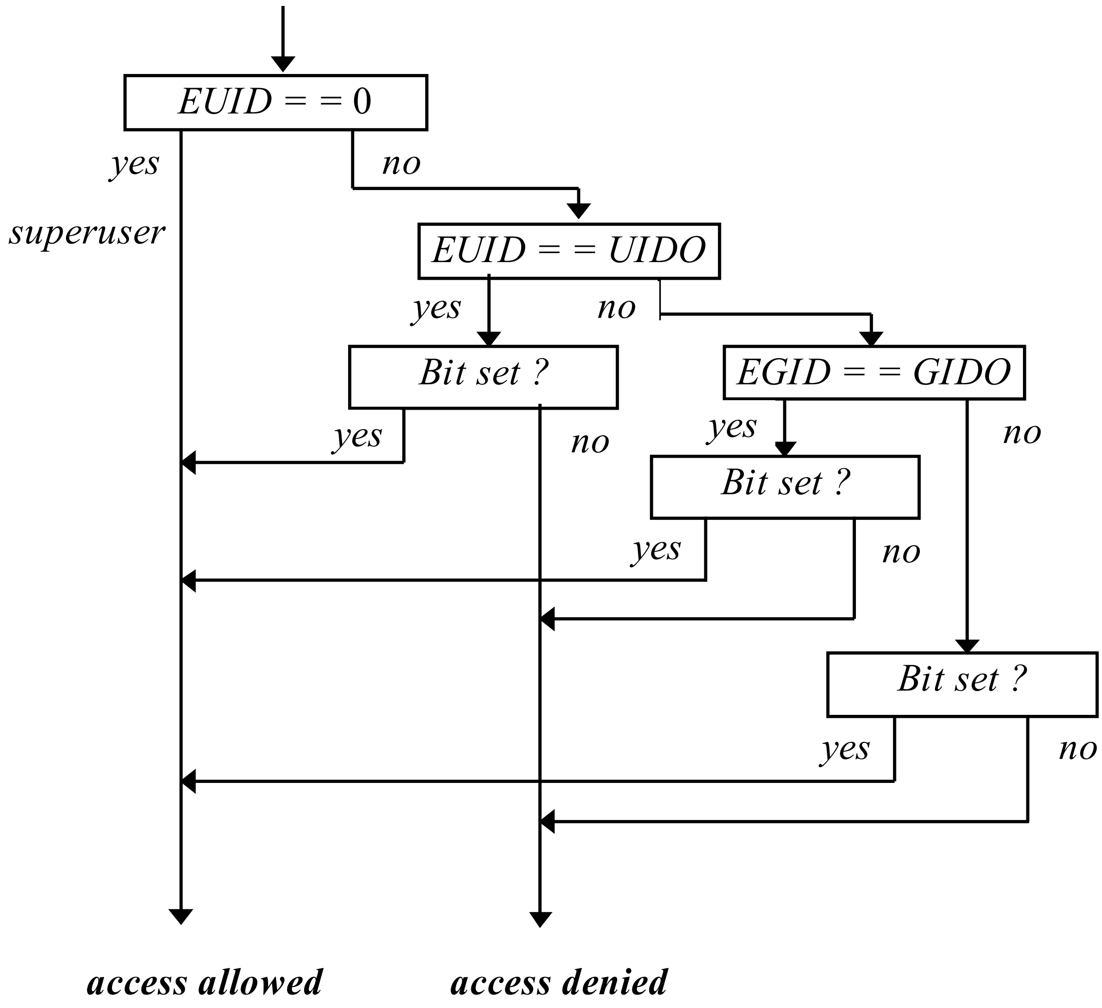


Fig. Access right check

Example

If access rights to a file are set

---rwxrwx

the owner of the file can't read , write or execute the file, but the others can.

Change of effective user by user s-bit:

If a file has user s-bit set, then before start of its execution, the *EUID* of process is set to *UIDO* of the file.

Change of effective group by group s-bit:

If a file has group s-bit set, then before start of its execution, the *EGID* of process is set to *GIDO* of the file.

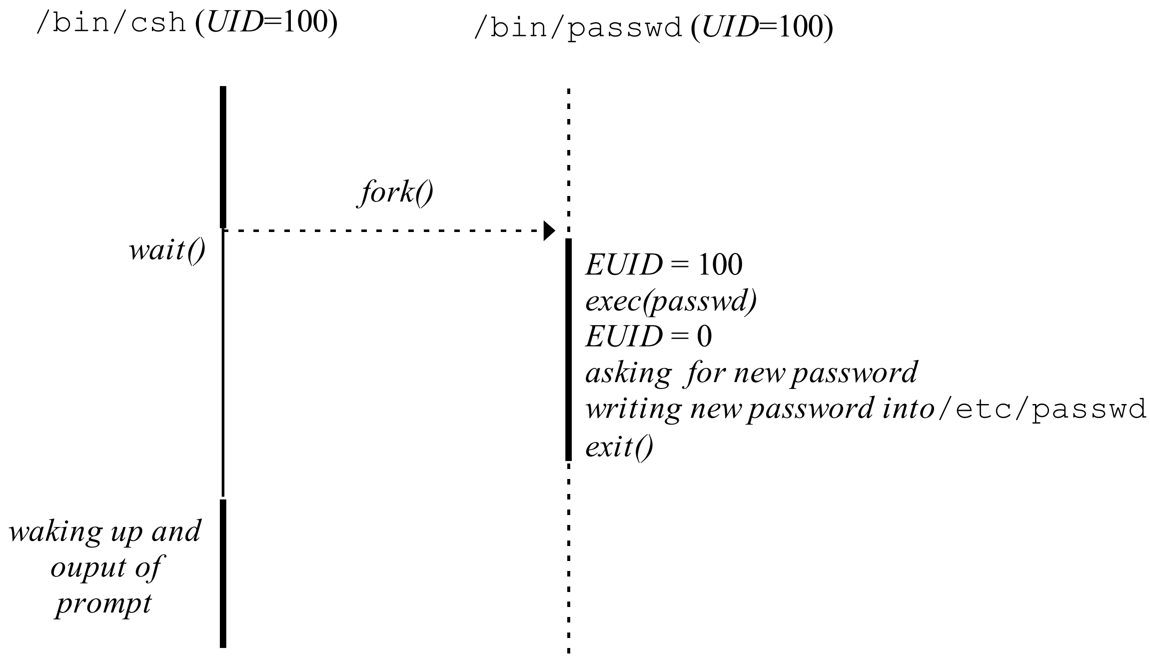


Fig.Writing into user table

Sticky bit: t-bit

is set together with access rights

Its meaning:

regular executable files: page table is maintained after exit of program

directories:

no meaning (older systems)

or possibility to write into directory but not to delete files of other owners

Setting access rights:

group s-bit

sticky bit

user s-bit

```
s s t  r w x  r w x  r w x
1 0 0  1 1 1  1 0 1  1 0 1
  4      7      5      5
```

system call:

`chmod("/home/novak/prog",04755);`

number must be octal

command:

chmod *access_rights file*

Example

`$`**`chmod`** `644 s1` *sets access rights of* `s1` *to* **rw-r--r--**

`$`**`chmod`** `4755 s2` *sets access rights of* `s2` *to* **rwxr-xr-x** *and sets s-bit*

Print of access rights

command:	***print:***
ls -l *file*	*file type, access rights, number of links, individual owner, group owner, size, time of last modification*
ls -il *file*	*as* ***ls -l*** *and number of i-node*
ls -lu *file*	*as* ***ls -l*** *, instead of time of last modification the time of last use is printed*
ls -lc *file*	*as* ***ls -l*** *, instead of time of last modification the time of last modification of i-node is printed*

individual owner	***group owner***	***others***
user s-bit	group s-bit	t-bit
r|- w|- x|-|s|S	**r|- w|- x|-|s|S**	**r|- w|- x|-|t|T**

r|- :
r **r** set
- **r** unset

w|- :
w **w** set
- **w** unset

x -| s | S :
- **x** unset, s-bit unset
x **x** set , s-bit unset
S s-bit set, **x** unset
s **x** set, s-bit set

x -| t | T :
- **x** unset, t-bit unset
x **x** set , t-bit unset
T t-bit set, **x** unset
t **x** set, t-bit set

Creating and mounting filesystems

Creating a control file of a filesystem

```
#/etc/mknod  /dev/c0d2s1  b  6  10
#/etc/mknod  /dev/c0d2s2  b  6  11
```

major

minor

Creating a filesystem

mkfs *control_file number_of_blocks* [: *number_of_i-nodes*]

Example

```
#/etc/mkfs  /dev/c0d2s1  800000
#/etc/mkfs  /dev/c0d2s2  1200000
```

Mounting a filesystem:

mount [**-r**] [-o *options*] *control_file directory_of_mounting*

-r only reading of mounted files is allowed

options:

nosuid	effect of s-bits is blocked for mounted files
noexec	no mounted file can be executed
nodev	no mounted control file can be used as control file

umount *control_file* | *directory_of_mounting*

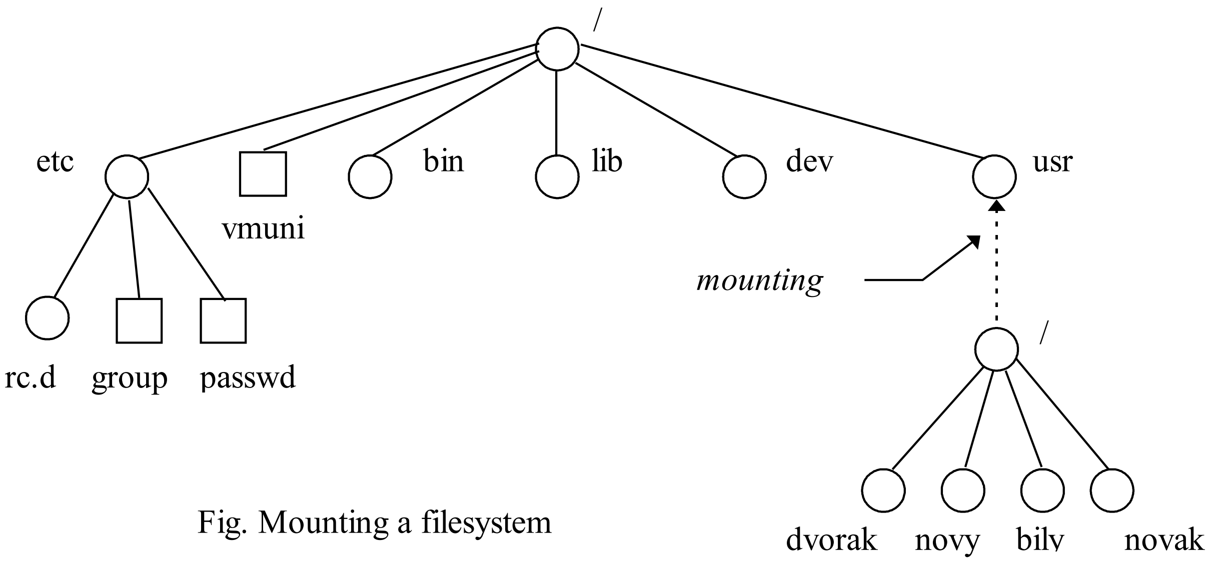


Fig. Mounting a filesystem

Security risks of s-bits

- If a user once breaks superuser password, he can hide a shell with set s-bit and superuser ownership under unsuspicious name somewhere in the file system for future intrusion.
- Therefore new files with set s-bit and superuser ownership are potentially dangerous and system administrator should look for them regularly.

For searching the system program find can be used

find / *-user* root -perm -4000 -print

System call chown()

- in some OS users can't use ***chown()*** at all
- in the other OS s-bit is during ***chown()*** system call reset

Security risks concerning mounting filesystems

- control I/O file with UID=0 might be mounted
- shell with s-bit and superuser ownership might be mounted

Security risk prevention

- in older OS only superuser can mount filesystems
- in newer OS user can carry out only restricted mount

    mount *control_file* *directory_of_mounting*

A user can't specify options of mounting.

The options are taken from file `/etc/fstab` whose content can change only superuser.

Unix processes

User context: text, data, stack, content of user registers

Systems context: ***process table, user area*** (u-area), content of system registers, content of system stack, page table

Process table contains:

- PID (process identification)
- Process state (for example Ready to Run, Asleep in memory etc.)
- Event descriptor when the process is in sleep state
- Field of signals (for each signal is reserved one bit)
- Pointer to page table (page table defines allocation of process in main memory)
- Pointer to u-area on disc
- Process identifiers, which specify the relationship of processes to each other (i.e. identifiers of parent process, child processes etc.)
- Various timers which count process execution time and utilization of system resources
- Scheduling parameters for evaluation of priority of the process

u-area contains:

- Real and effective user ID and group ID (*UID, EUID, GID, EGID*)
- Current directory and current root directory
- Descriptor table
- I/O parameters
- Signal handlers
- Control terminal
- Error field (records errors encountered during a system call)
- Return value field (contains the result of a system call)

Process states

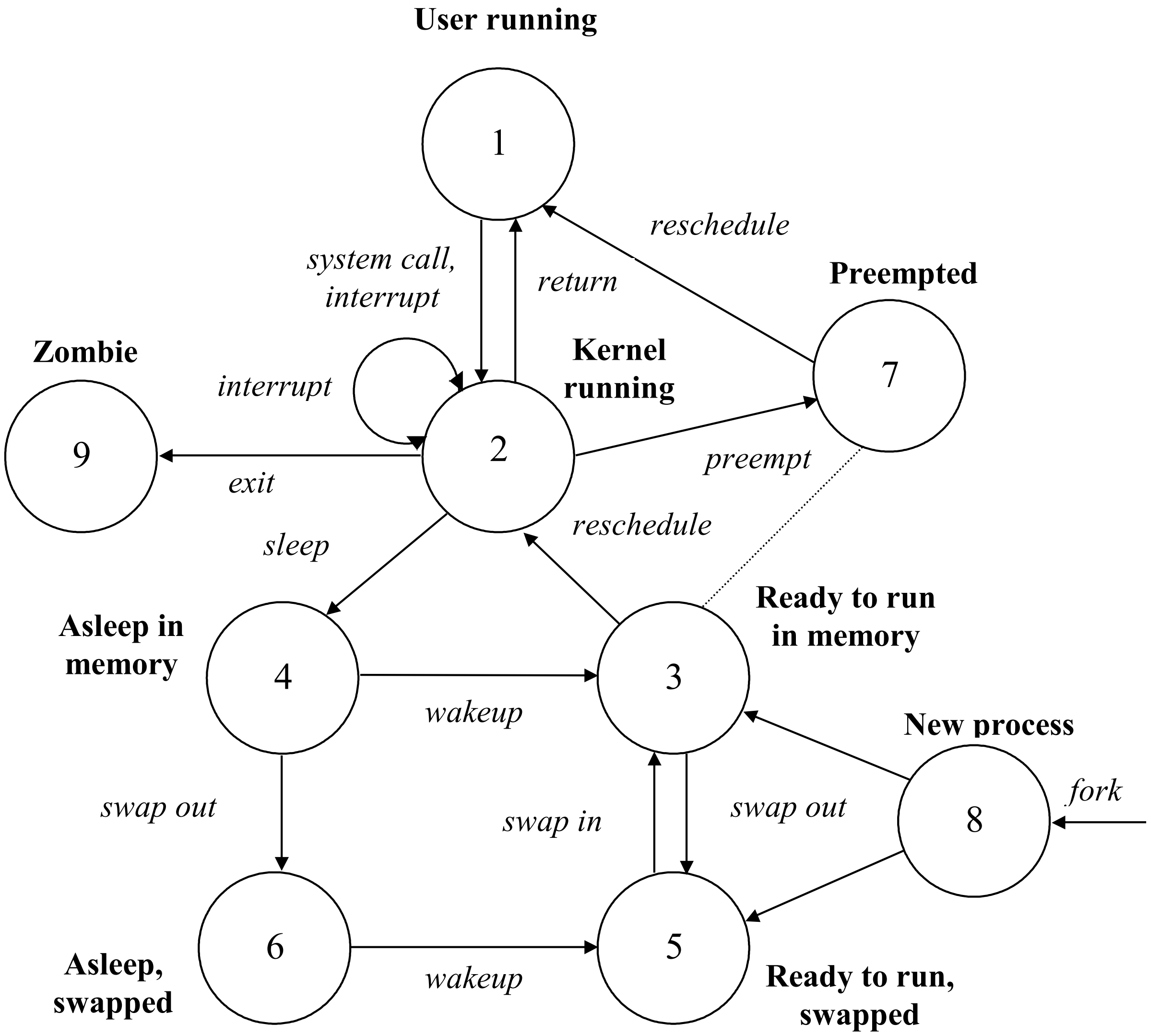


Fig. Process state transition

Process in asleep state

- **When process runs in kernel mode and some condition necessary for its further execution is not fulfilled, the process is blocked (put into asleep state)**
- **Process will wait in asleep state till the event "condition is fulfilled" occurs.**

Process for example needs

- data from disc
- some buffer which is locked
- exit of some another process etc.

Process asleep

Algorithm sleep_on(event)

- Change of process state is written into process table.
- Process is removed from Scheduler table.
- Process is put into a cue in Asleep process table. Asleep process table is organized according to the events which processes are waiting on

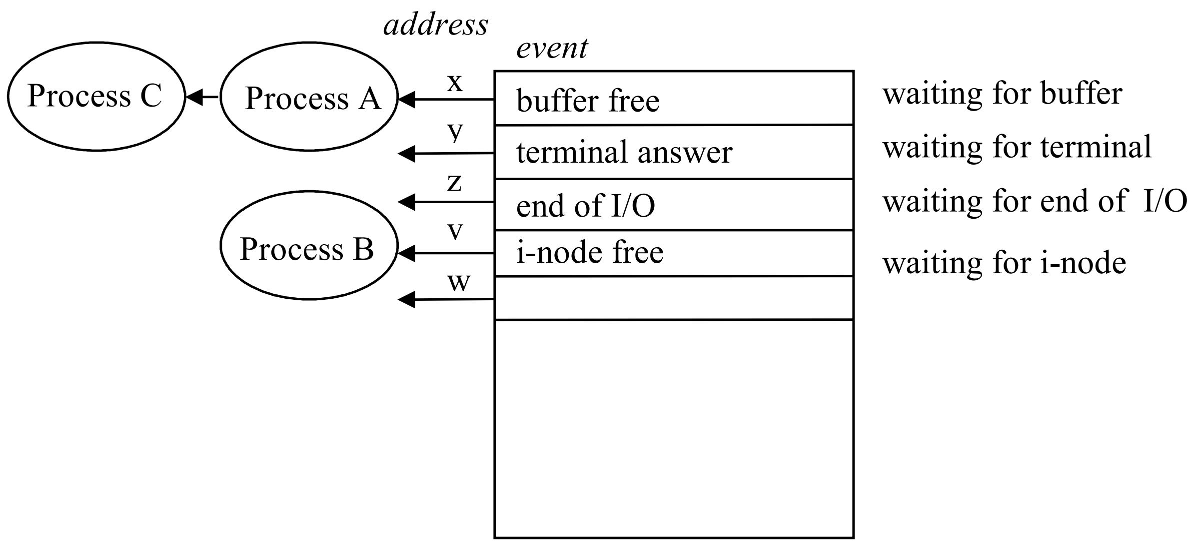


Fig. Asleep process table

Process wake up

As soon as an event of Asleep process table occurs, OS will wake up all processes of its cue.

Algorithm wake_up(event)

1. Change of process state is written into process table
2. Process is removed from Asleep process table
3. Process is put into a cue in Scheduler table

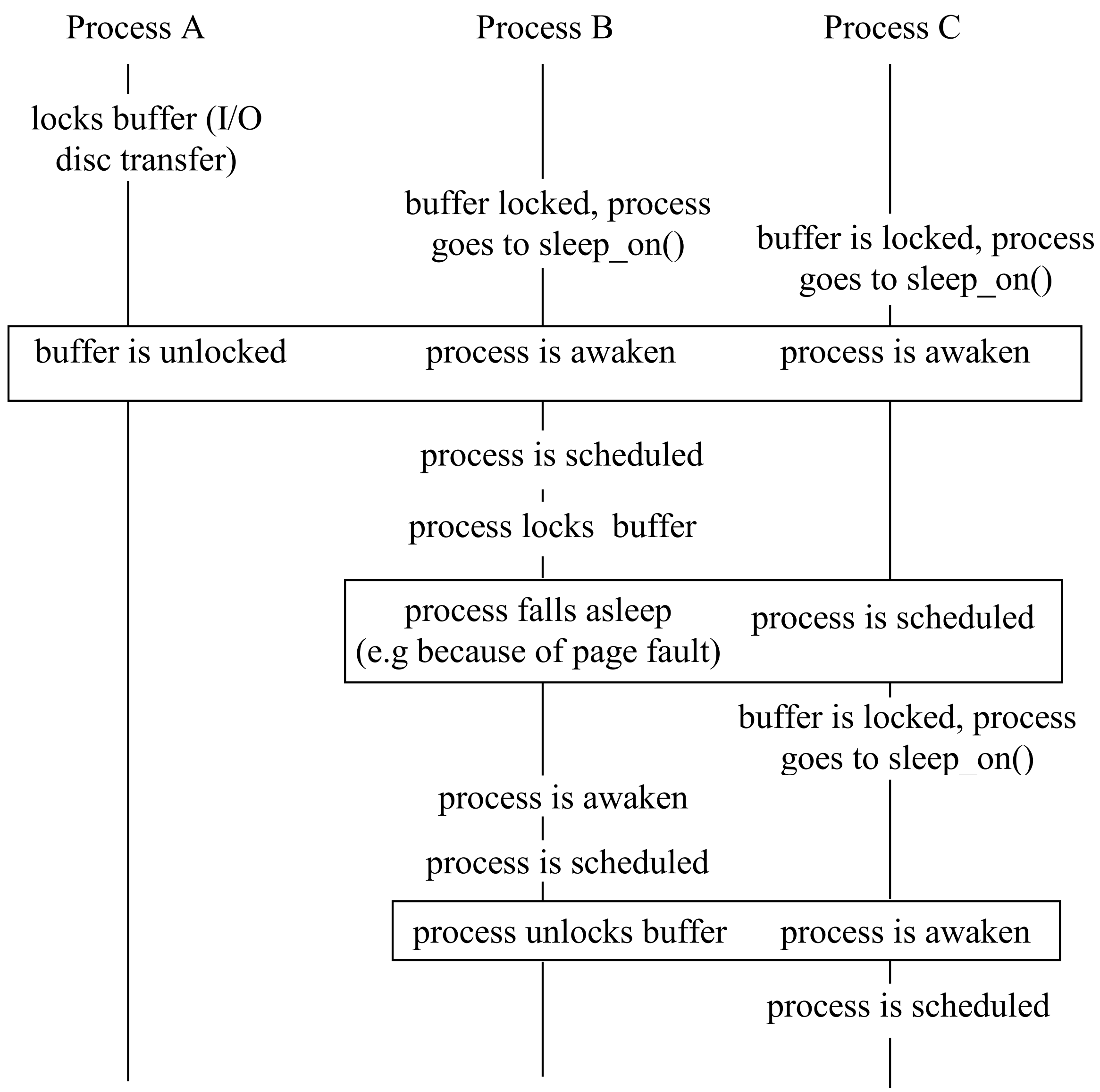


Fig. Sleeping processes

Scheduling

- Processes are scheduled according to their **priority**
- Priority is a positive or negative number or zero
- The process with smaller priority has **higher scheduling priority**

Evaluation of priority

Processes, which will run in user mode:

$$priority = base + CPU\_usage$$

user can set value of *base* by system call *nice()* to some positive value

CPU_usage is a parameter, that value is stored in process table:

1. When process is scheduled $CPU\_usage=0$
2. At every time tic

$$CPU\_usage = CPU\_usage+1$$

3. Before each evaluation of *priority*

$$CPU\_usage=CPU\_usage/2$$

When priority is evaluated?

- Priority is evaluated at the moment of process registration in the scheduler table.
- Then the priority is updated every 1sec.
- Priority is also evaluated if process returns from kernel running state into user running state.

Priority of a process that will run in user mode is positive

Processes that will run in kernel mode:

- These processes had to run in kernel mode and had to fall in asleep state (The reason: process running in kernel mode can't be preempted!)
- Priority is a negative number that corresponds to the event that caused the process to fall asleep

Priority of a process that will run in kernel mode is negative

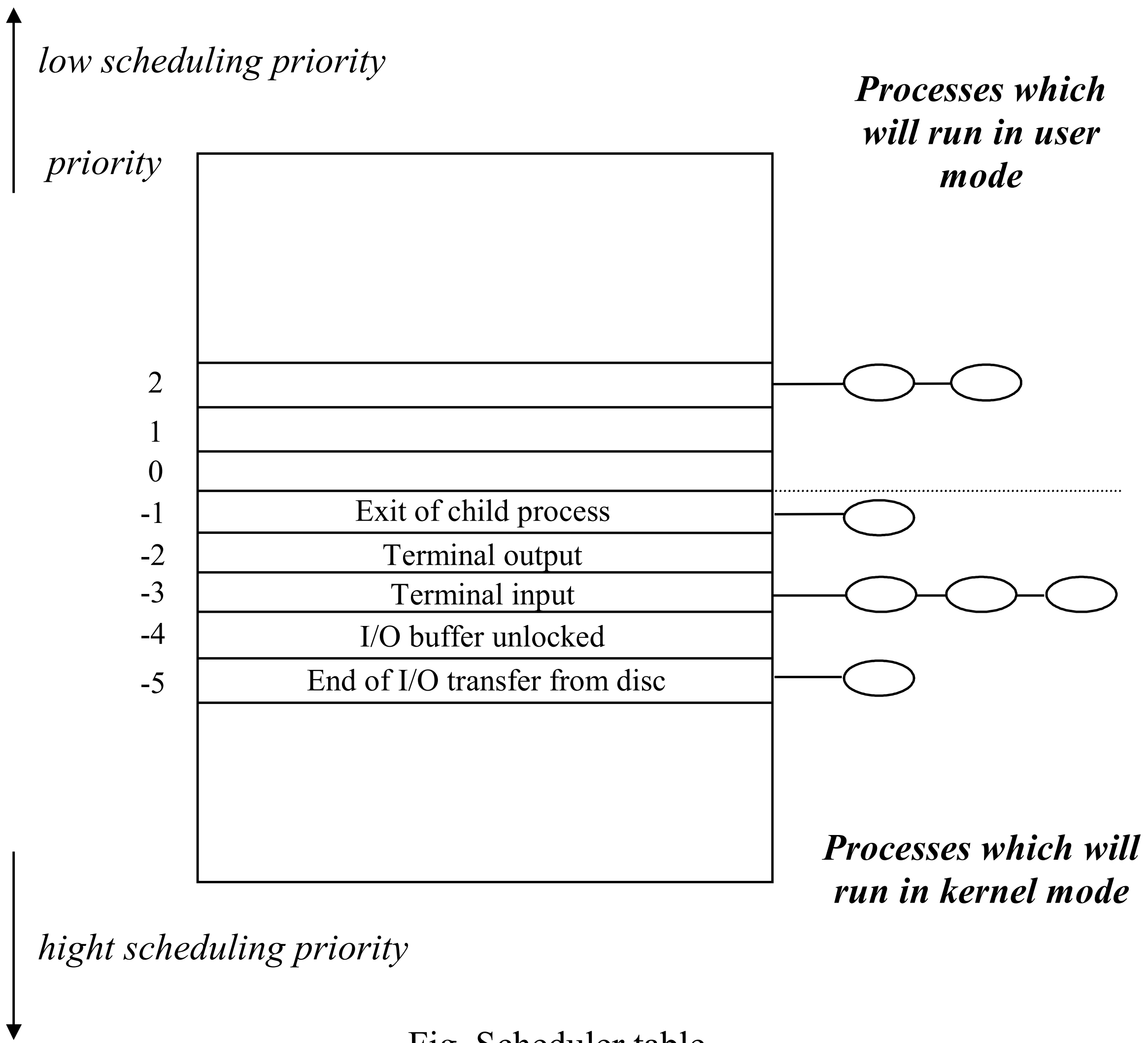


Fig. Scheduler table

Allocation of processes in memory

Unix uses the technique known as **paging**

(historically Berkeley based OS used paging and SYSTEM V based OS used paging combined with segmentation)

Necessary technical support of processor:

Processor must have address unit with **dynamic address translation (DAT),** controlled by page table.

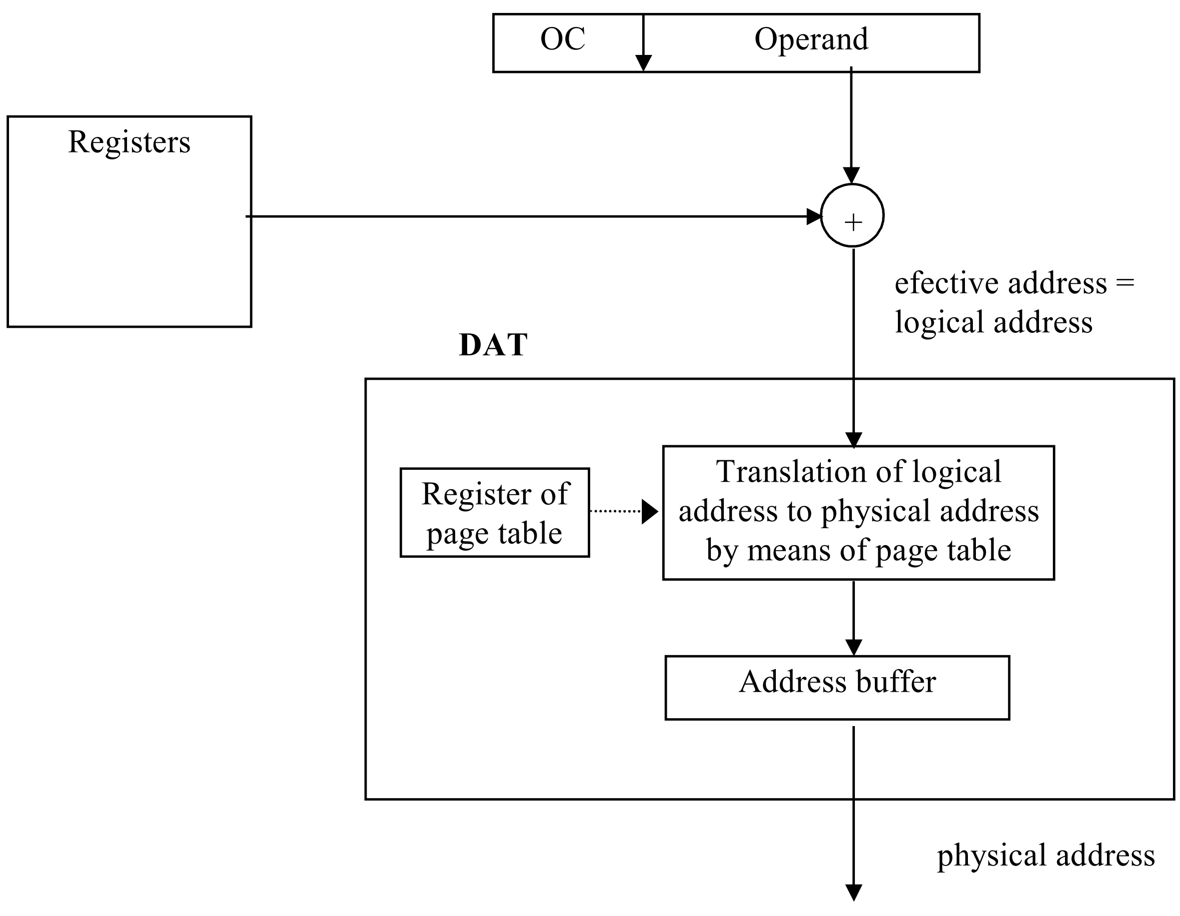


Fig. Address unit with DAT

Activities of address unit:

- If the row of page table is valid (P=1) address unit translates logical address to physical address
- If the row is not valid (P=0) address unit generates **page fault interrupt**
- If the process writes into a page, address unit sets the change bit (Z=1)

Address unit has associative cache memory for caching used rows of page table (so called TLB, Translation Lookaside Buffer)

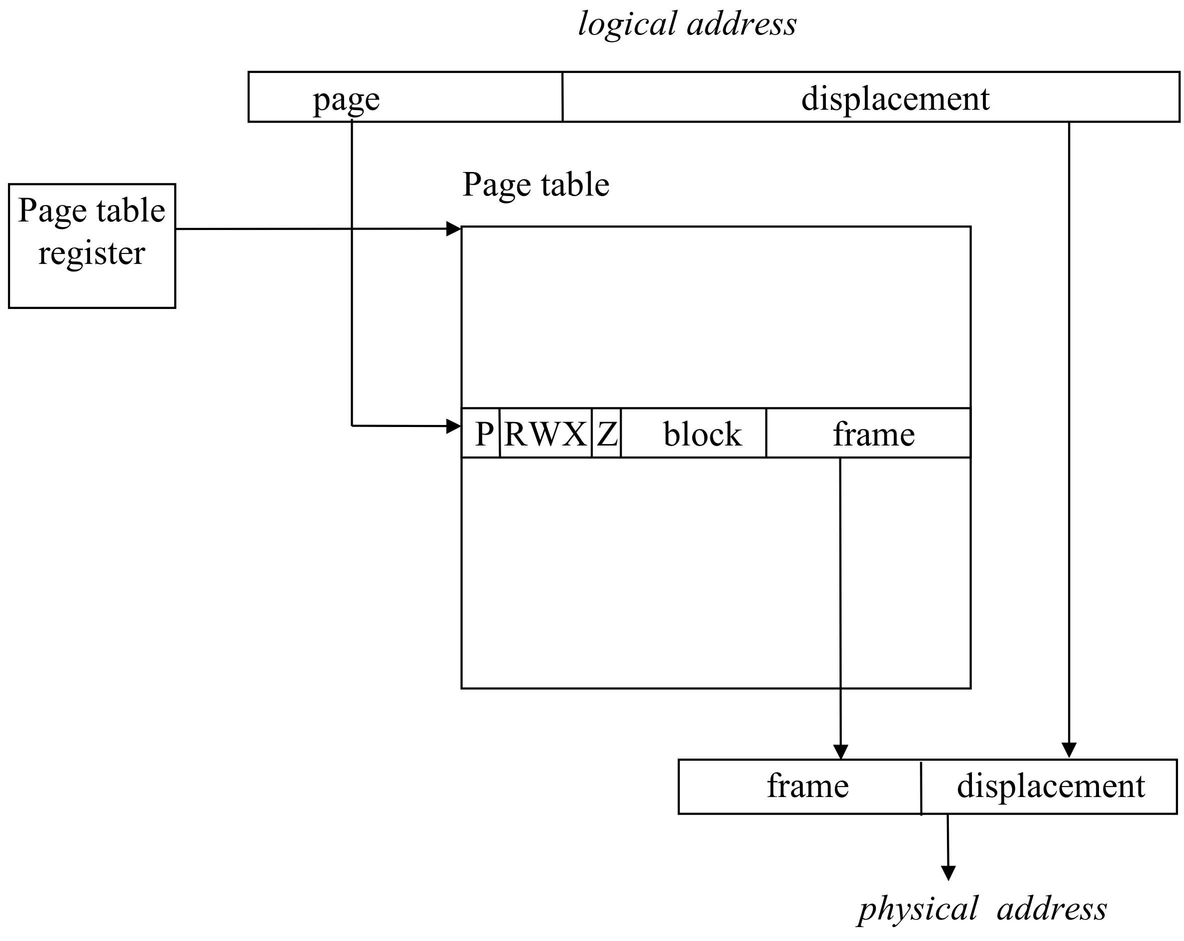
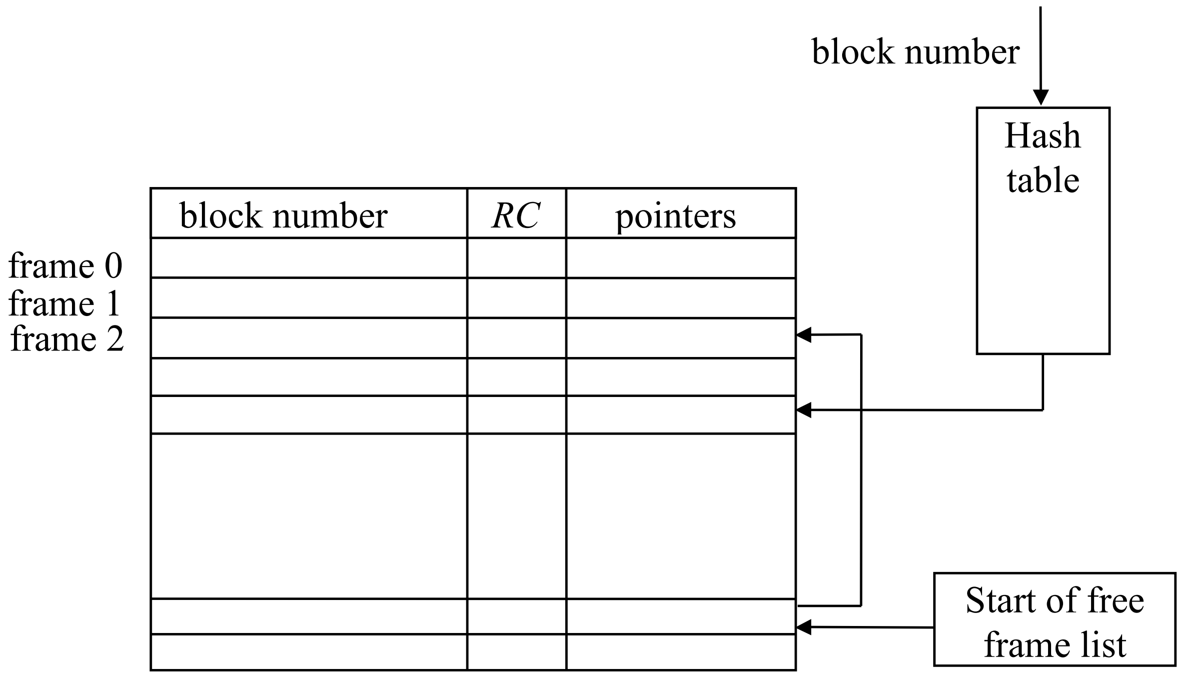


Fig. Translation of logical address to physical address

Allocation of memory:



RC number of pointers from page table

Fig. Memory map

1. When process is created, OS builds up data structure of its page table and initializes it. For text pages and initialized data pages OS writes down corresponding numbers of blocks. OS also set bits P, R, W, X, Z.
2. If process uses address inside a page that row in page table is not valid, address unit generates fault page interrupt (When process starts running, it has no valid row of page table).
3. After interrupt, fault page handler allocates for the page a frame in memory.
4. If it is necessary, the fault page handler initiates reading of the page content into allocated frame from disc. (Processes share text pages and initialized data. Therefore some pages can be already in memory and their reading is not necessary).
5. When the process is scheduled next time, address, which had caused interrupt, is translated and process continues running

Page fault handler activity

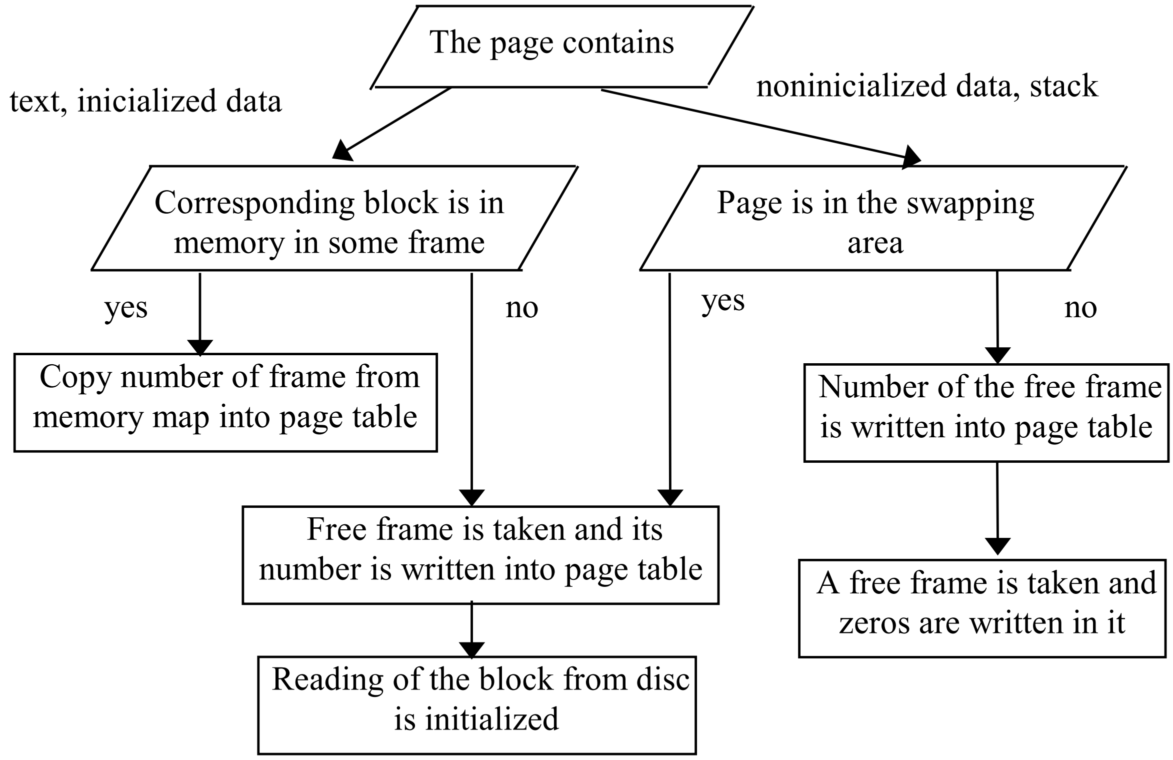


Fig. Page fault handler activity

Deallocation of frames:

- it is done by process **page_daemon** (*PID*=2)
- page_daemon is running in regular intervals (i.e. 200 ms)
- page_daemon keeps number of free frames between low and high limits
- page_daemon is also activated if there are no free frames available

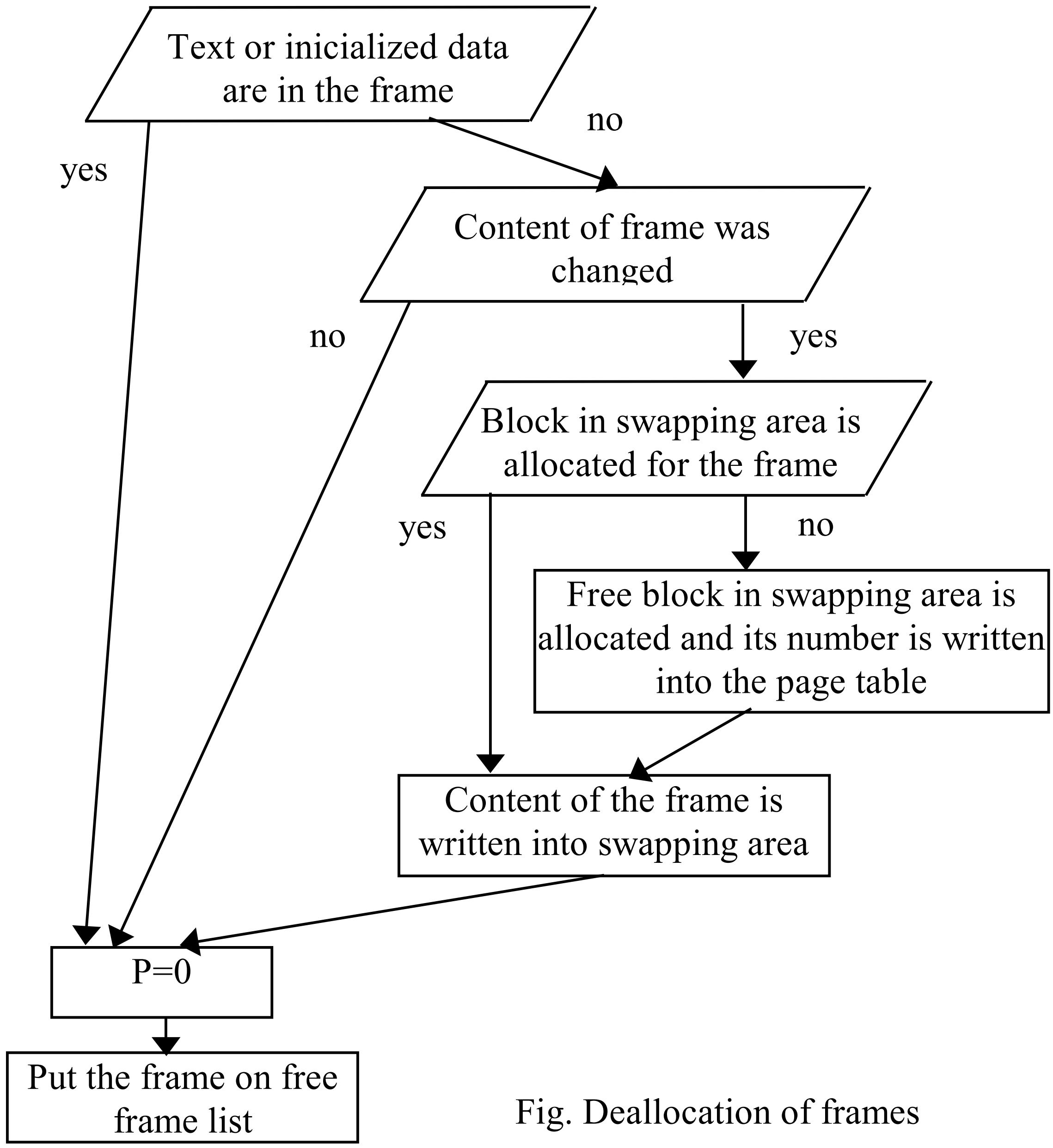


Fig. Deallocation of frames

Allocation of processes in main memory

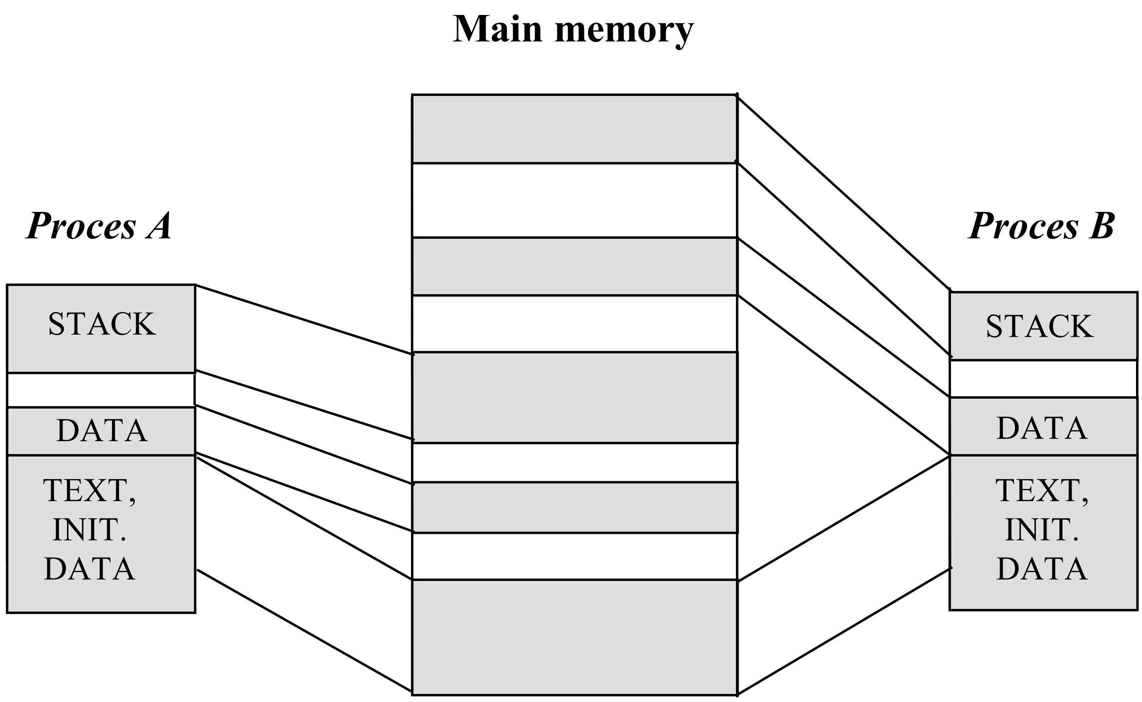


Fig. Processes *A* a *B* are controled by the same program

Unix system calls for process control

Process creation

- The first process after system load is created by the kernel (usually swapper (*PID*=0))
- All other processes are created by the same way: one of the existing processes calls the system service *fork()*
- Therefore each process has its parent process (swapper is the only exception)
- History of process creation could be described by a graph

OS can reconstruct this graph from information maintained in the process table

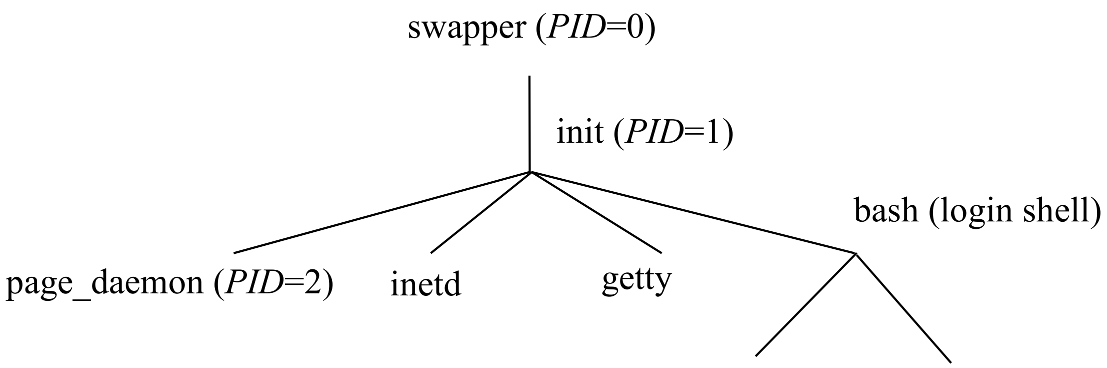


Fig. Graph of history of process creation

Process group (*PGRP*)

- Every process is a member of one process group (OS maintain process *PGRP* in the process table)
- Every process group has its process group leader
- Leader of a group is a process for which is *PID*=*PGRP*

How a process can become a process leader?

- The first created process has *PID*=*PGRP*
- After *fork()* the child process inherits context of parent process and therefore also its process group
- Each process can ask OS for process leadership by system call *setpgrp()*. After this call OS sets *PGRP* of calling process to its *PID*. Therefore after this call the calling process will become a leader of a new process group

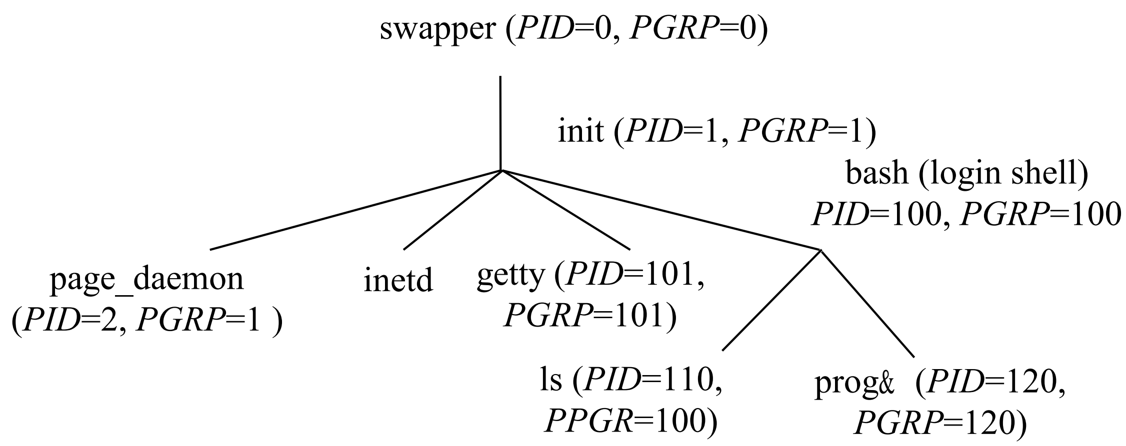


Fig. Process groups

Control terminal

- User processes usually have control terminals
- Control terminal is always accessible by special driver `/dev/tty`
- System processes that are running in the system, fulfill some system tasks and have no control terminal, are called daemons

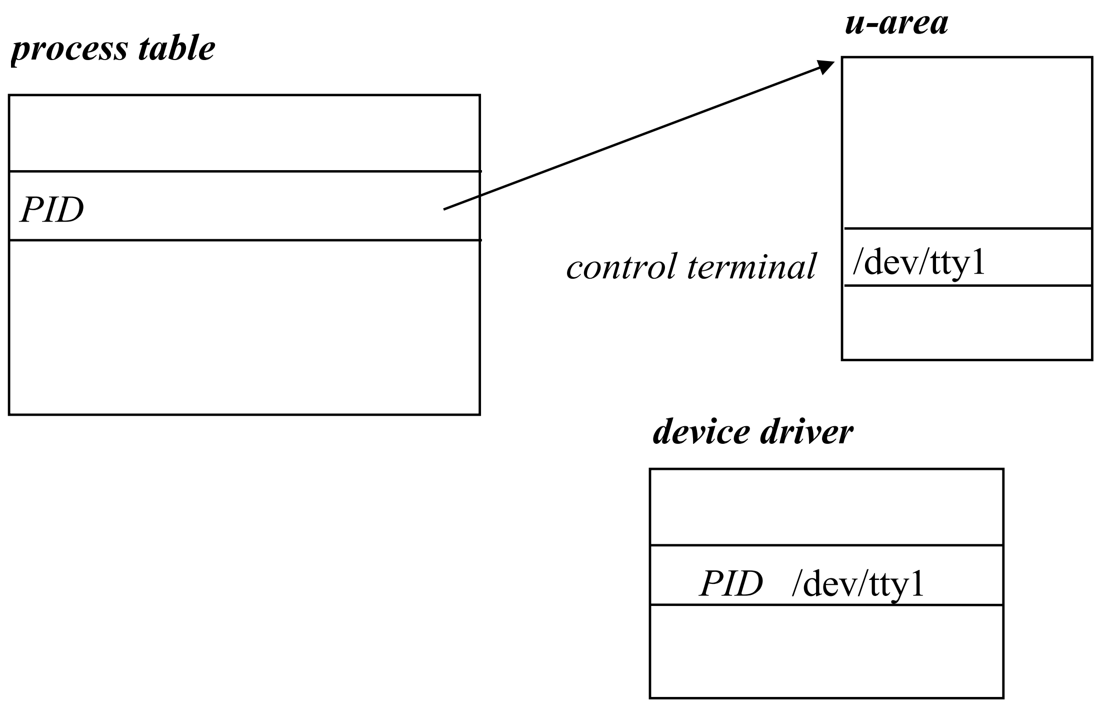
How a process can gain a control terminal:

If:

- Process is a process group leader
- Process has no associated control terminal yet
- Process has opened control file of a terminal (i.e. it calls system service `open(/dev/tty1, . . . ))`
- The opened terminal has not been associated to any other process as a control terminal

Then:

1. The opened terminal will become a control terminal of the process (control file of the terminal is written into u-area of the process, into the item *control terminal*)
2. Process will become control process of the terminal (the couple *PID* of the process and control file of the terminal is written into terminal driver data structure)



The mechanism of control terminal enables these actions of OS:

1. After exit of a process, that is process group leader and has control terminal, OS sends signal SIGHUP to all processes of its group
2. After switching off the control terminal, OS sends signal SIGHUP to all processes, that are members of process group of its control process including control process (control process must be process group leader).

Thus processes running on the background are not canceled neither after logout nor after the terminal is switched off.

System call *fork()*

Creation of a new process (***child process***)

```
int fork(void);
pid=fork();
```

OS creates a copy of the process that called *fork()*.

OS activity during *fork()* system call

1. OS allocates a free row in process table for the child process. It copies items from parent process table row into child process table row with these exceptions:

    child process has its own PID,

    own pointer to u-area,

    own pointer to page table

2. OS allocates a u-area on disc.

3. OS creates new page table for the child process. At the beginning the contents of data and stack areas are in the both processes the same. During further run they may differ.

4. OS writes into item *return system call value* in u-area:

    child PID in parent process

    0 in child process

I.e. system call in parent process returns *PID* of the child process and in the child process it returns 0. Therefore although parent and child processes are controlled by the same program, they could be controlled by different parts of it.

Example

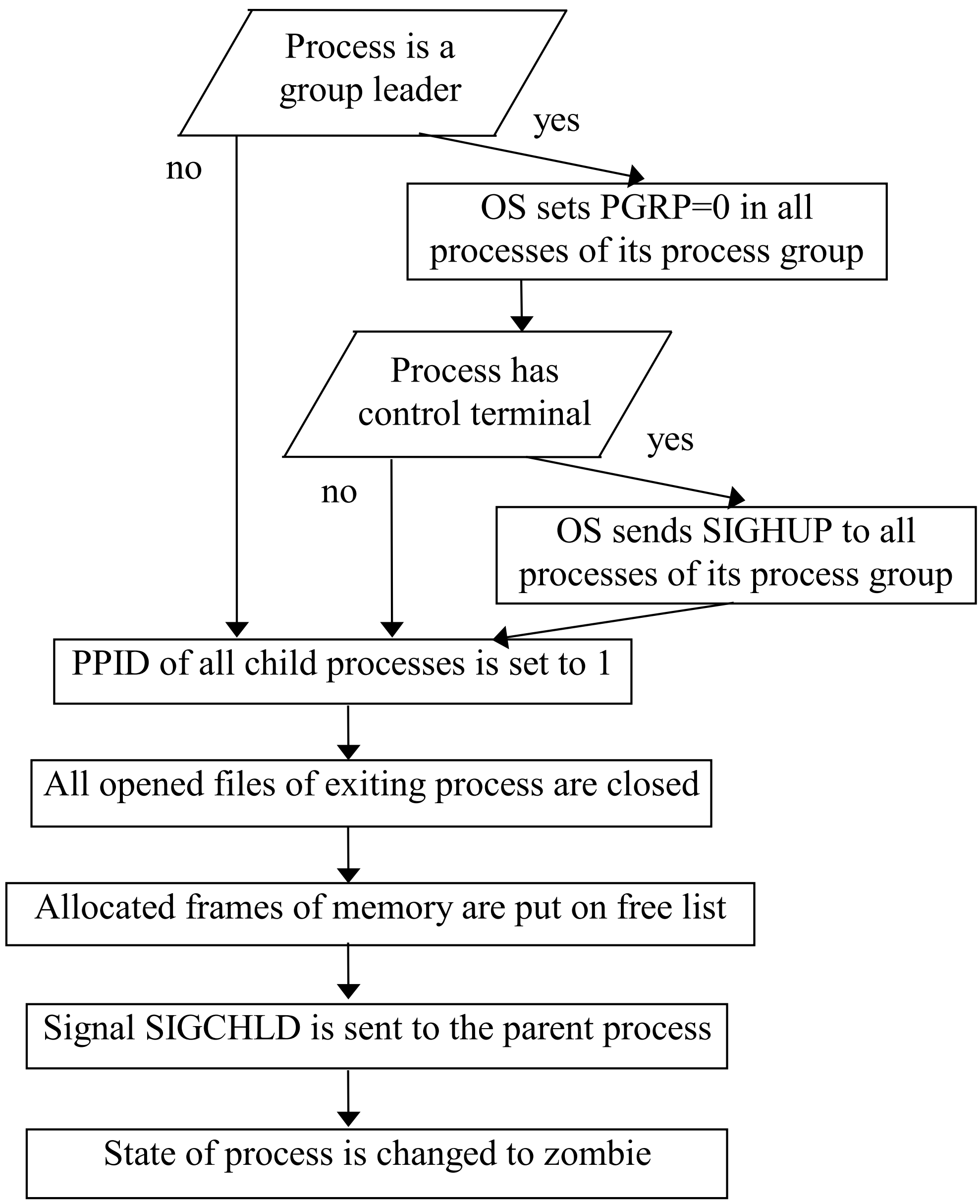
```
#include <stdio.h>
main(){
int pid;
if  ((pid = fork())==0){
    printf("\nChild process:PID=%d Parent PID=%d\n",\
                getpid(),getppid());
        pause();
}
printf("\nParent process:PID=%d Child PID=%d\n",\
                getpid(),pid);
exit(0);
}
```

System call *exit()*

Termination of a process. After *exit()* call the process will be in zombie state.

```
void exit(int status);
exit(status);
```

OS activity during *exit()* system call:



System call *wait()*

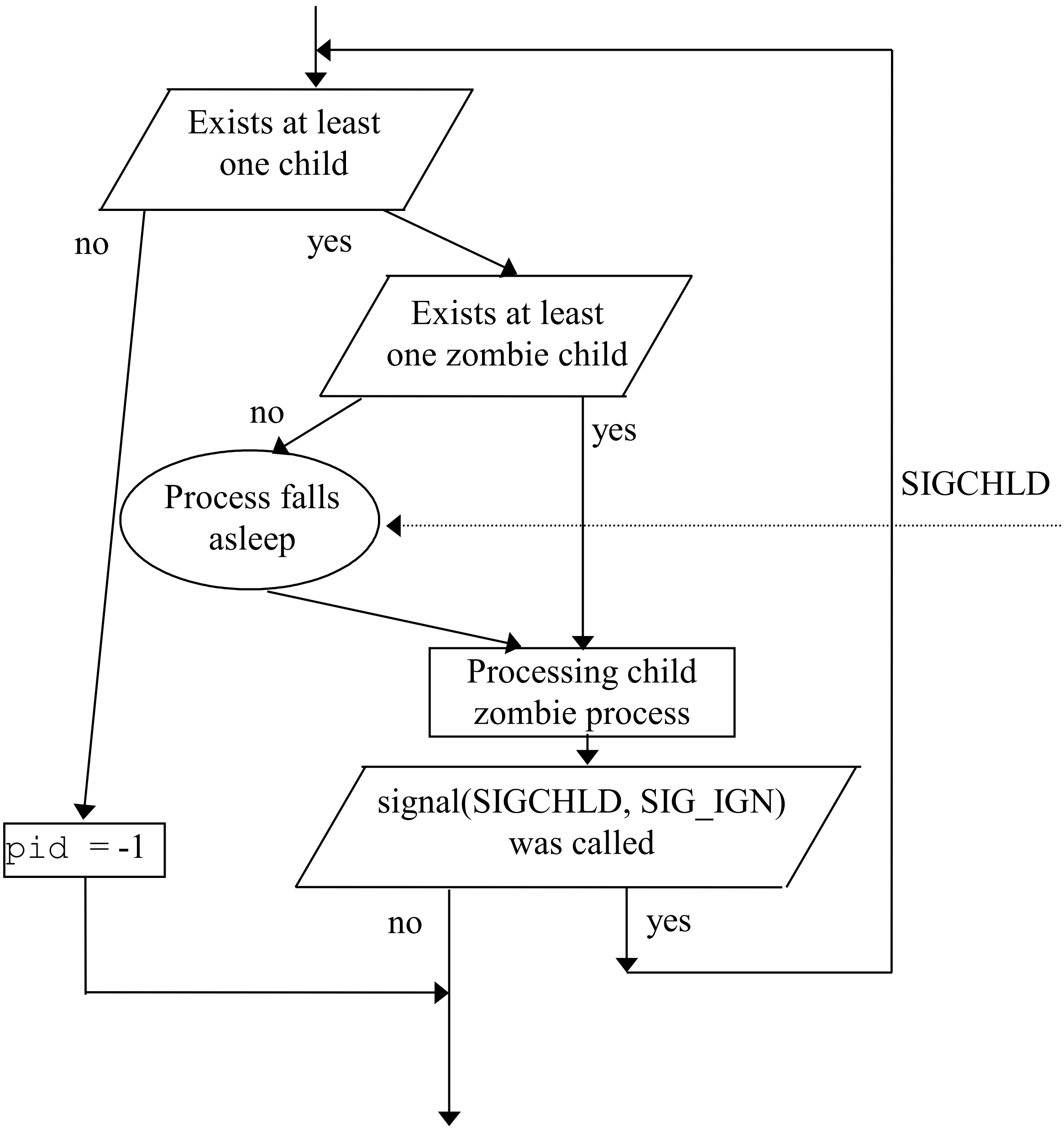
Waiting for exiting of child processes and their processing.

```
int  wait(int *stat_adr);
pid=wait(stat_adr);
```

Processing of a child zombie process means:

- *PID* of the zombie child is put into return variable `pid`
- least significant bits (bits 0-7) of `status` (`status` is the parameter of *exit()* call of the zombie child) are put into 8-15 bits of `stat_adr`
- Counts of utilization of system resources of the zombie child process are added to those of the parent process
- Zombie child process is deleted from the process table

OS activity during *wait()* system call



System call *execve()*

Change of text of a running process (start of a new program)

```
int execve(char *path,char *argv[],char *envp[]);
ret=execve(path,argv,envp);
```

`path` is a pointer to disc file with new program

`argv` is a pointer to a field of pointers, which point to character strings. The strings will be accessible after text change. First string must be the name of the file containing new text. The last pointer in the field must be NULL pointer.

`envp` is a pointer to a field of pointers that points to character strings. The string are usually of the form X=Y and will be accessible after text change. The last pointer in the field must be NULL pointer.

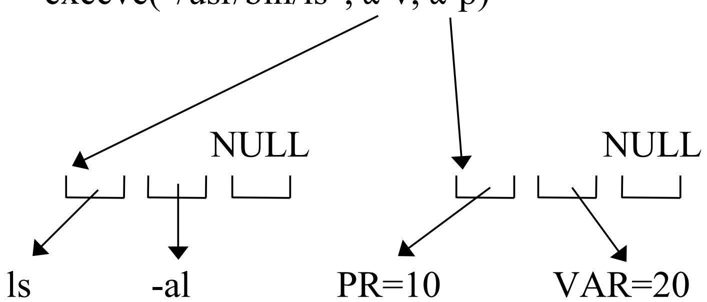
New program can access in *execve()* call specified strings only if it is written as:

```
main(argc,argv,envp)
int argc;
char *argv[];
char *envp[];
{
. . . .
}
```

OS will put into `argc` the number of pointers in `argv` field (not counting NULL pointer)

Example

execve(“/usr/bin/ls“, & v, & p)



OS actions during system call *execve()*

- OS checks if `path` is executable program and if process has access rights for its execution
- If file `path` has s-bit set, OS will change process *EUID* or *EGID* to the owner of the file
- OS copies all strings specified by `argv` and `envp` into kernel stack
- OS create new page table for the process (the old page table is deleted)
- OS copies all strings from kernel stack to new user stack
- OS puts the start address of the new program into instruction counter

Example

List of environmental variables of a shell.

```
#include <stdio.h>
main(argc,argv,envp)
int argc;
char *argv[];
char *envp[];
{
    int i;
    for(i=0;envp[i] != (char *) 0; i++)
        printf("%s\n",envp[i]);
    exit(0);
}
```

Example

Starting program **/bin/date** during execution of a program

```
main()
{

    char *(argv[2]);
    argv[0] = "date";
    argv[1] = ((char *)0);
    execve("/bin/date",argv,(char *)0);
}
```

System call *kill()*

Processes send signals to the other processes by system call *kill()*. Also kernel can send a signal to a process.

There are 32 signals.

In process table OS reserves for each process a bit field 32 bits long (for each signal one bit).

Superuser process can send signals to all processes (with some exceptions: it can not send signals to several important system processes, which cannot be aborted).

User process can send signals only to a such process, *UID* of which equals to *EUID* of the sending process.

Syntax of system call *kill()*

```
int  kill(int pid, int sig);
ret=kill(pid, sig);
```

`sig` number of signal

`pid` has the following meaning:

- if `pid` > 0 , signal is sent to process *PID* = `pid`
- if `pid` = 0 , signal is sent to all processes of the same process group
- if `pid` = -1 , signal is sent to all processes
- if `pid` < -1 , signal is sent to all processes of the process group `-pid`

Return value `ret=0` if the call is successful. Otherwise `ret=-1` .

Sending of a signal:

1. Kernel sets corresponding bit in signal bit field in process table.
2. If process is asleep, kernel wakes it.

Processing of a signal:

Received signals are always processed before process continues running.

Processing depends on whether the reaction on a coming signal was specified by system call *signal()* beforehand or not:

1. If not, default action is taken. For most signals default action means abortion of the process. For some signals (e.g. SIGCHLD, SIGCNT) default action simply means to do nothing, i.e. the process is only waked up.

2. If yes, the action is specified in *signal()* system call. It could be:

a) to ignore the signal

b) to reset to default action

c) to execute signal handler code

Signal handler code must be part of the executed program.

Syntax of system call *signal()*

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

`sig` number of signal

`func`

`SIG_IGN` Ignore signal (signals 9 and 19 cannot be ignored)

`SIG_DFL` Reset default action

Pointer to signal handler. Signal handler has one formal variable of type `int`. When signal handler is executed, number of the sent signal is accessible in this variable.

Example

Program that can’t be killed by signal 2. (If this program is running on foreground, it can’t be canceled by `CTRL+C`).

```
#include <signal.h>
#include <stdio.h>
void handler(sig)
int sig;
{
     if (sig == 2)
          printf("I can´t be killed by signal 2\n");
     signal(2,handler);
     signal(14,handler);
}
main()
{
     int i;
     signal(2,handler);
     signal(14,handler);
     for(i=0;i<100;i++){
          alarm(5);
          pause();
          printf("hello\n");
     }
}
```

Important signals:

1 SIGHUP

a) After finishing of a process, that is process group leader and has control terminal, OS sends signal SIGHUP to all processes of its process group.

b) After switching off the control terminal, OS sends signal SIGHUP to all processes that are members of process group of its control process.

2 SIGINT

is sent by the kernel after pressing interrupt key on keyboard (usually `CTRL+C`).

9 SIGKILL

is used for reliable process exit. This signal cannot be caught by *signal()* system call.

14 SIGALRM

is sent by kernel after system call *alarm(n).*

15 SIGTERM

is used for standard process exit. Program **kill** sends this signal, if number of signal is not specified.

17 SIGCHLD

is sent by the kernel to a parent process if its child process exits.

18 SIGCONT

is used to awake a process, which fell asleep after `SIGSTOP` signal.

19 SIGSTOP

is used to make a process to fall asleep. Signal could not be caught.

How shell executes commands?

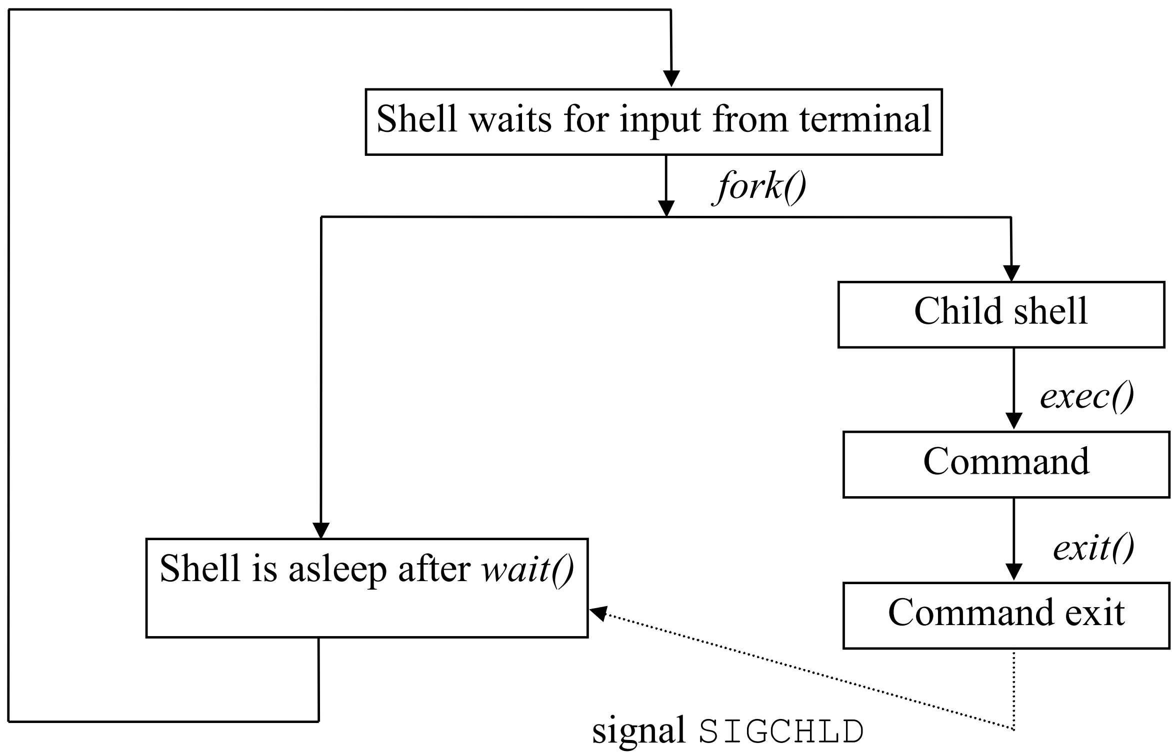


Fig. Running command on the foreground

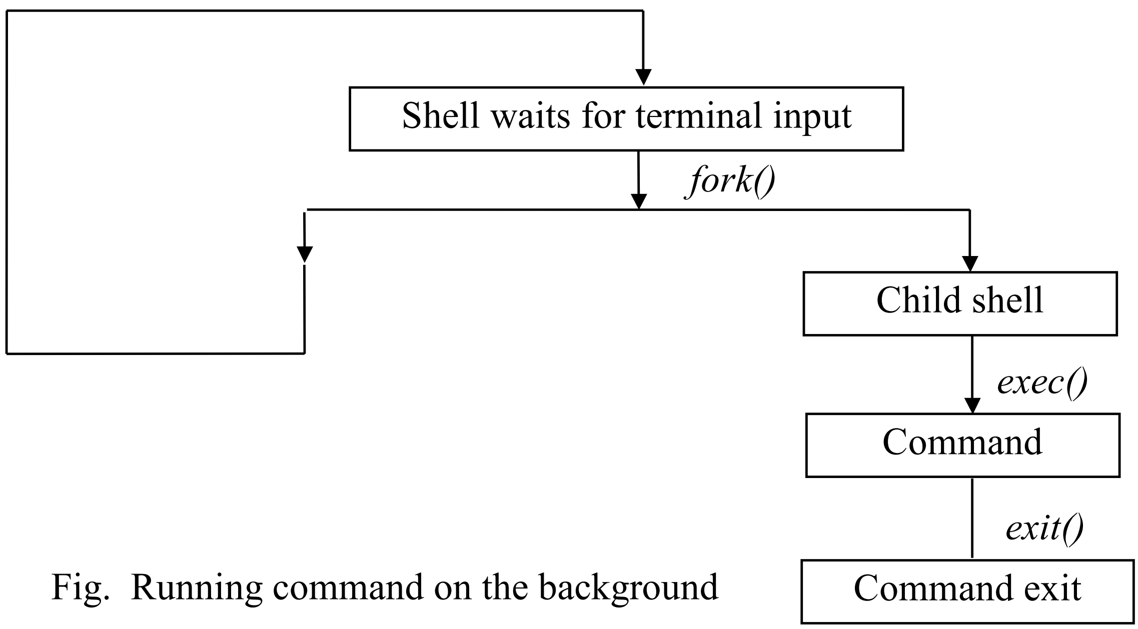


Fig. Running command on the background

Unix file manager system calls

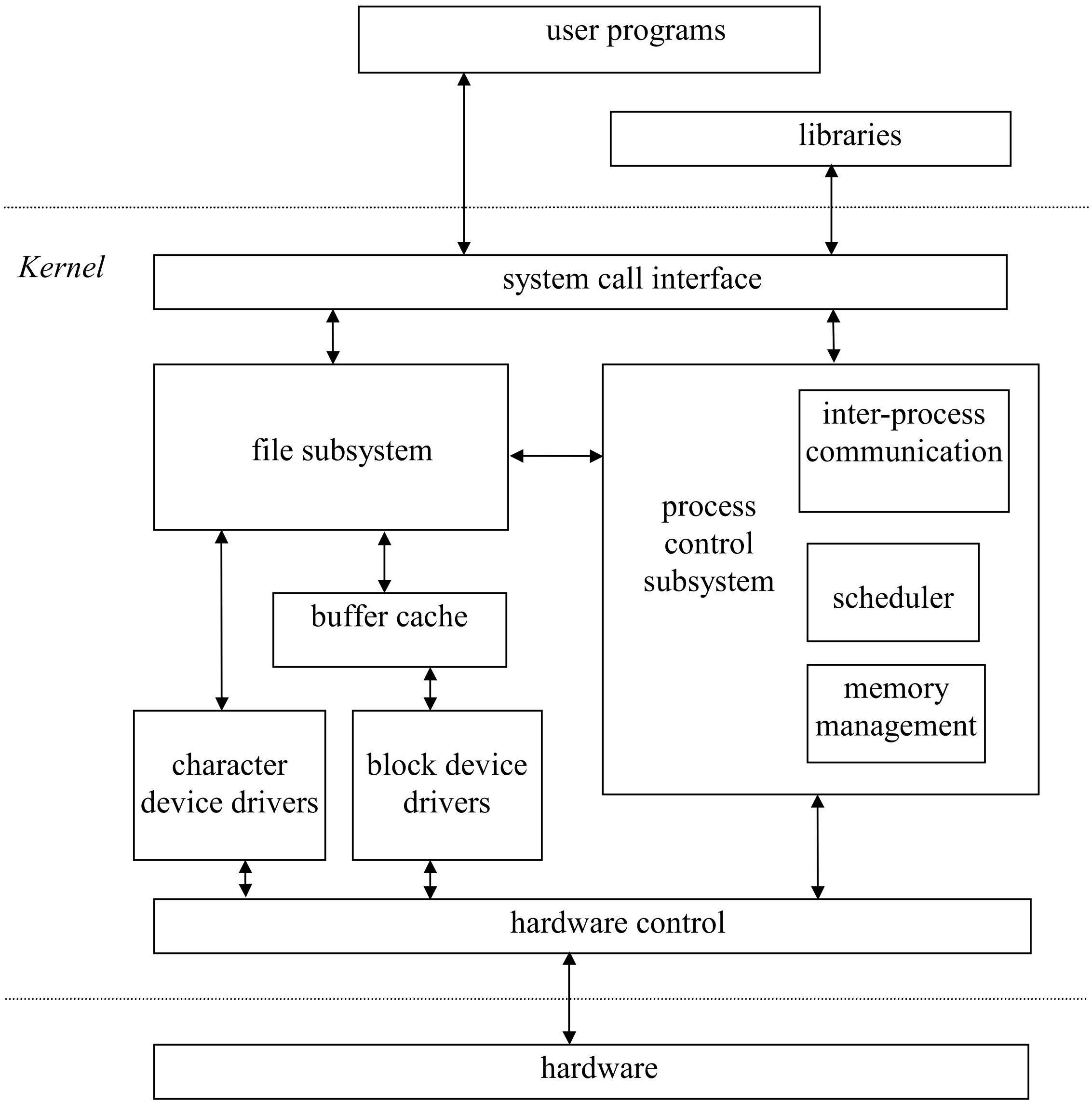


Fig. Block structure of the system kernel

System calls

Open or create and open a file

```
int open(char *pathname,int flag, int mode);
        fd=open(pathname,flag,mode);
```

`pathname`	name of the file to be opened	
`flag`	defines mode of opening	
	`O_RDWR`	open for reading and writing
	`O_RDONLY`	open for reading only
	`O_WRONLY`	open for writing only
	`O_CREAT`	create the file if does not exist. Mode specifies permissions. The flag has no meaning if the file already exists.
	`O_EXCL`	open fails if this bit and O_CREAT bit are set and file exists (exclusive open)
`mode`	specification of access rights if new file is created	

open() returns a file descriptor `fd` for use in other system calls

Example

```
fd=open("/etc/group", O_RDWR|O_CREAT, 0666);
```

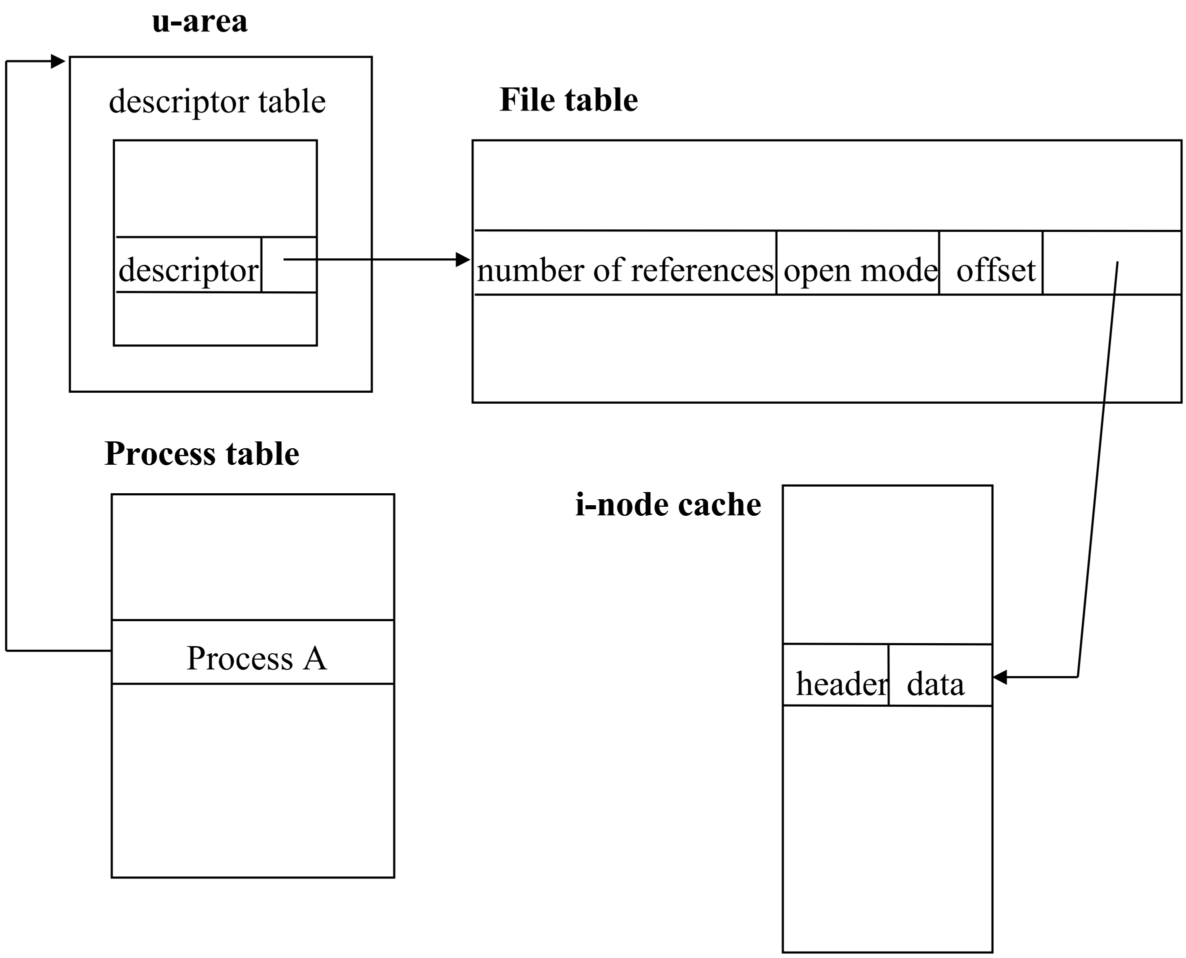


Fig. File open

OS actions during *open()* call:

1. It looks through directories for i-node number of the file.
2. It checks the access rights.
3. If i-node is not in i-node cache, it copies it there.
4. It allocates the first free row in descriptor table.
5. It allocates the first free row in file table.
6. In descriptor table it sets pointer to file table.
7. In file table: it sets pointer to i-node cache, it sets open mode and it initializes the number of references from descriptor table to 1 and the offset to zero

Example

```
#include <stdio.h>
main()
{
    int fd1,fd2,fd3;
    fd1=open("/etc/hosts",O_RDWR);
    fd2=open("/etc/passwd",O_RDONLY);
    fd3=open("/etc/passwd",O_WRONLY);
    continue:
    . . . . . .
}
```

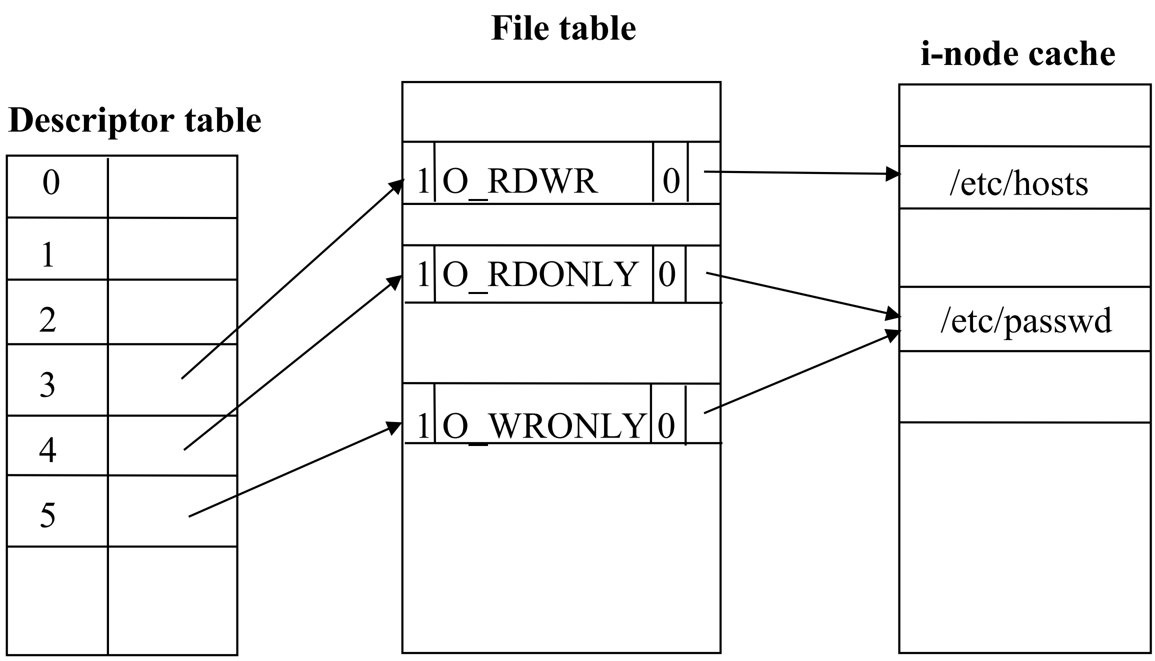


Fig. File open

Access rights of a new created file

For each process system maintains a parameter called *mask*

Access rights of a new created file are set:

$$access\_rights = mode \wedge \neg mask$$

Example

mask = 022

```
fd=open("/usr/smith/data", O_RDWR|O_CREAT, 0777);
```

If file `/usr/smith/data` does not exist, it will be created with access rights:

$access\_rights = mode \wedge \neg mask = 0777 \wedge \neg 0022 = 0755$

Mask set:

```
int umask(int mask);
old_mask=umask(mask);
```

File close

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

Close() closes the file with descriptor `fd`

OS actions during system call *close()*

in descriptor table: OS deletes the row of closing descriptor `fd` .

in file table: OS decreases the number of references by 1. If the number of references equals to zero, OS deletes the row from file table.

in i-node cache: If file table row is deleted, OS decreases the number of references to i-node buffer by 1. If the number of references equals to zero, OS puts the i-node buffer on free list.

File delete

```
int unlink(char *pathname);
unlink(pathname);
```

Unlink() removes the directory entry.

Descriptor duplication

```
int dup(int fd);
newfd = dup(fd);
```

Dup() duplicates specified file descriptor. The row of descriptor `fd` is copied into the first free row of descriptor table.

Example

```
#include <stdio.h>
main()
{
    int fd1,fd2,fd3;
    fd1=open(”/etc/hosts”,O_RDWR”);
    fd2=open(”/etc/passwd”,O_RDWR”);
    fd3=dup(fd2);
    continue:
    . . . . . .
}
```

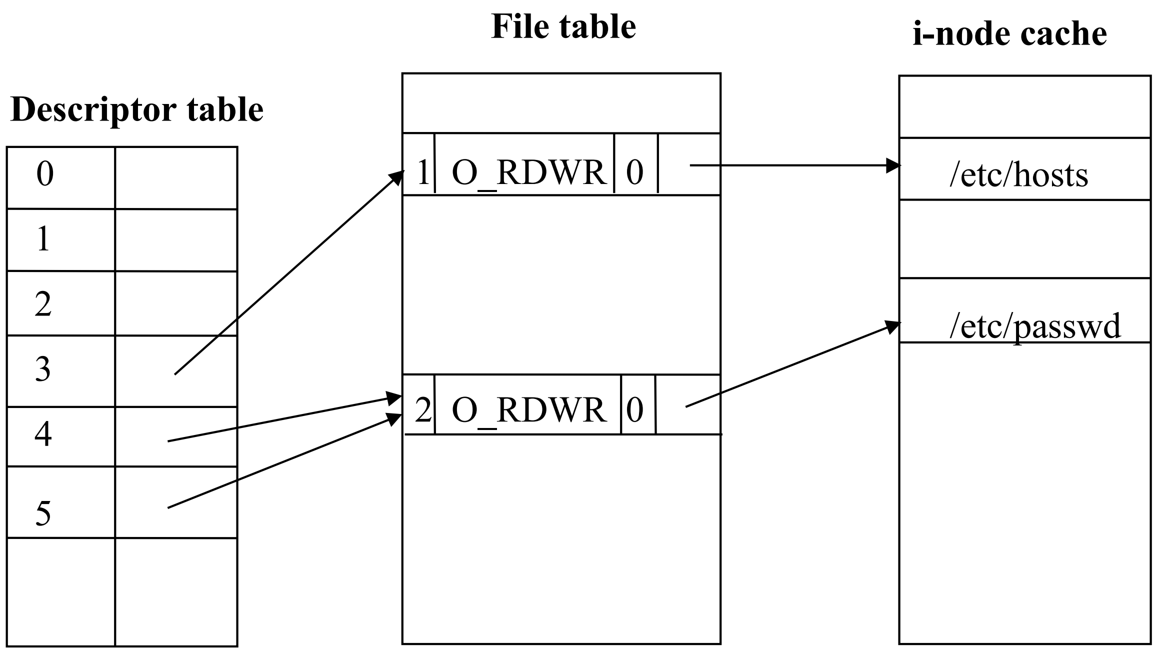


Fig. Descriptor duplication.

Read and write a file

```
int read(int fd,char *buffer,unsigned count);
number = read(fd, buffer, count);
```

`fd` descriptor

`buffer` user read buffer

`count` number of bytes to read

Read() reads up to `count` bytes from the file `fd` into the user read buffer `buffer`. *Read()* returns the number of bytes actually read. *Read()* during reading updates offset.

```
int write(int fd,int buffer,unsigned count);
number = write(fd, buffer, count);
```

`fd` descriptor

`buffer` user write buffer

`count` number of bytes to write

Write() writes up to `count` bytes into the file `fd` from the user buffer `buffer`. *Write()* returns the number of bytes actually written. *Write()* during writing updates offset.

Example

Function *getchar()* reads one character from standard input and returns its value. When reading behind the end of the file, it returns `EOF` (usually -1).

Following program copies standard input on standard output:

```
#include <stdio.h>
main()
{
    int c;
    while((c=getchar()) != EOF)
        putchar(c);
    return 0;
}
```

***getchar()* can be realized:**

```
#include <stdio.h>
#define CMASK 0377
int getchar()
{
    char c;
    return (read(0,&c,1) >0 ? c & CMASK : EOF);
}
```

Direct access to a file

```
int lseek(int fd,int offset,int reference);
position = lseek( fd, offset, reference);
```

`fd` descriptor

`offset` number of bytes

`reference` reference position:

the beginning of the file (`reference` = 0)

current position (`reference` = 1)

the end of file (`reference` = 2)

System call ***lseek()*** changes offset (the position of the read-write pointer) for the file `fd` and returns the new value. The value of `reference` defines the reference position for offset counting.

Communication support

1. Communication support for TCP/IP protocols was originally built into Berkeley Unix in early eighties.
2. Very soon it was built also into System V Unix and other OS.

Sockets

1. OS supports communication by means of sockets.
2. Sockets are data structures by means of which processes running on different systems exchange data.

Input or output data that go through sockets are sequences of bytes.

Processes can read and write socket data in the similar manner as any other I/O data.

Thus socket data are considered to be data streams.

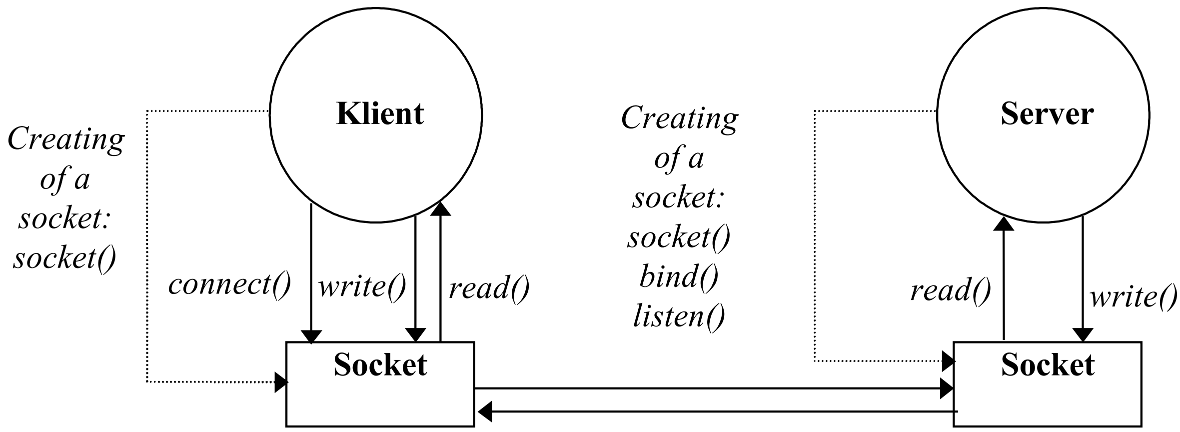


Fig. Communication between processes by means of sockets

Communication model is client-server

Server:

1. It creates socket and initializes its parameters (*socket()*, *bind()*, *listen()*)
2. It calls system service *read()*. As soon as a request comes, server answers by the system call *write()*.

Server process is running all the time.

When server does not execute client's request, it is in asleep state.

Client:

It is running only when user has some request to a server.

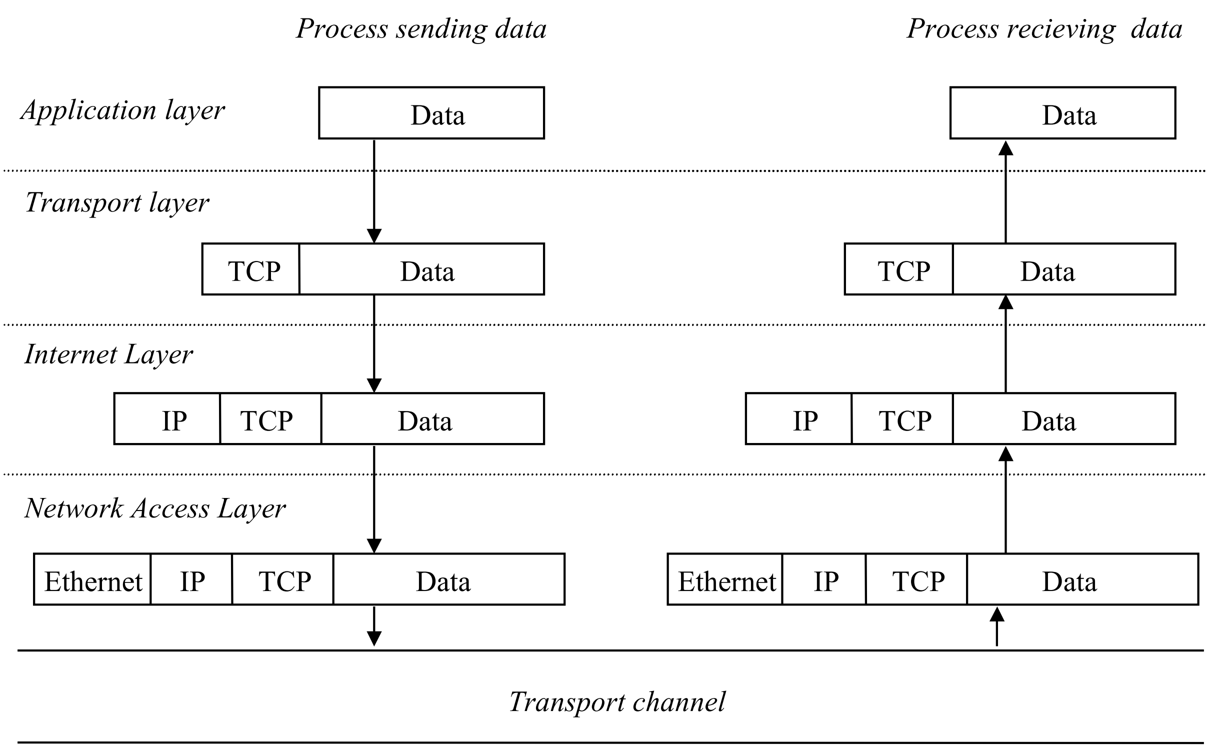
1. Client creates socket (*socket()*)
2. Client builds up connection with server (*connect()*)
3. Client sends request (*write()*)
4. Client waits for an answer (*read()*)

Comparison of RM OSI and Unix communication model

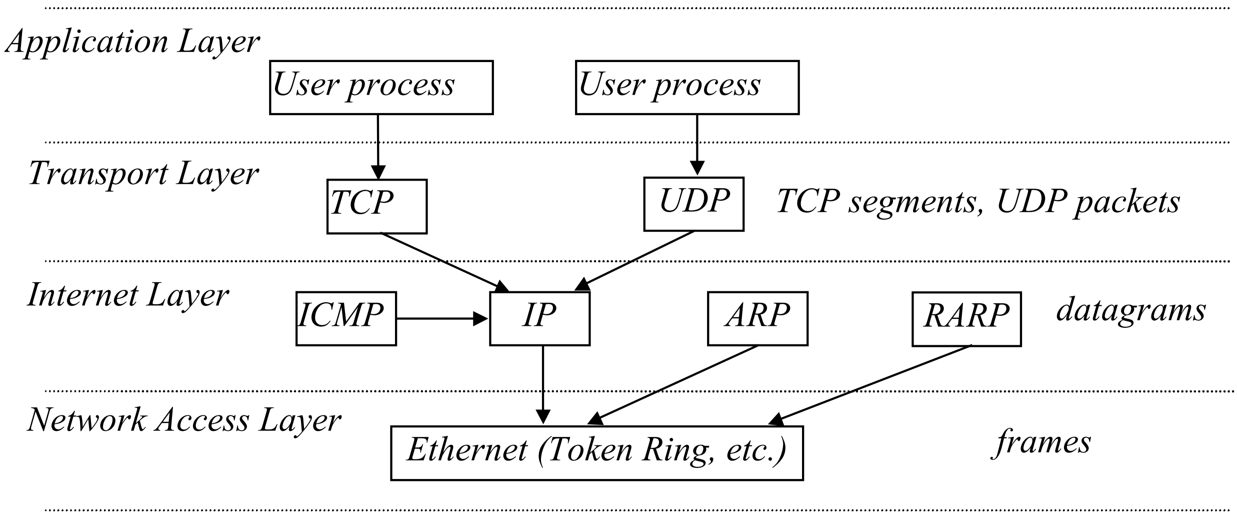
Unix communication model	*RM OSI model*
Application Layer	*Aplication Layer* *Presentation Layer*
Transport Layer	*Relation Layer* *Transport Layer*
Internet Layer	*Network Layer*
Network Acces Layer	*Data Link Layer* *Physical Layer*

Communication process

(Sending data accross transport channel (e.g. Ethernet))



Family of protocols TCP/IP



Difference between TCP and UDP protocols

TCP:

- TCP protocol grants reordering of received packets
- If some packet is missing, it asks for its resending

UDP:

It does not provide such facilities, but transfer using this protocol is quicker and more efficient.

Protocol ARP

Resolves the relationship: Internet address – local net address (e.g. Ethernet address)

All hosts store this relationship in their ARP caches

ARP request: A host, which needs to know Ethernet address of an a host with specified Internet address broadcasts ARP request.

Host with this specified Internet address will broadcast **ARP reply**, in which it will put its Internet address together with its Ethernet address. All hosts in local net will write this information into their ARP caches.

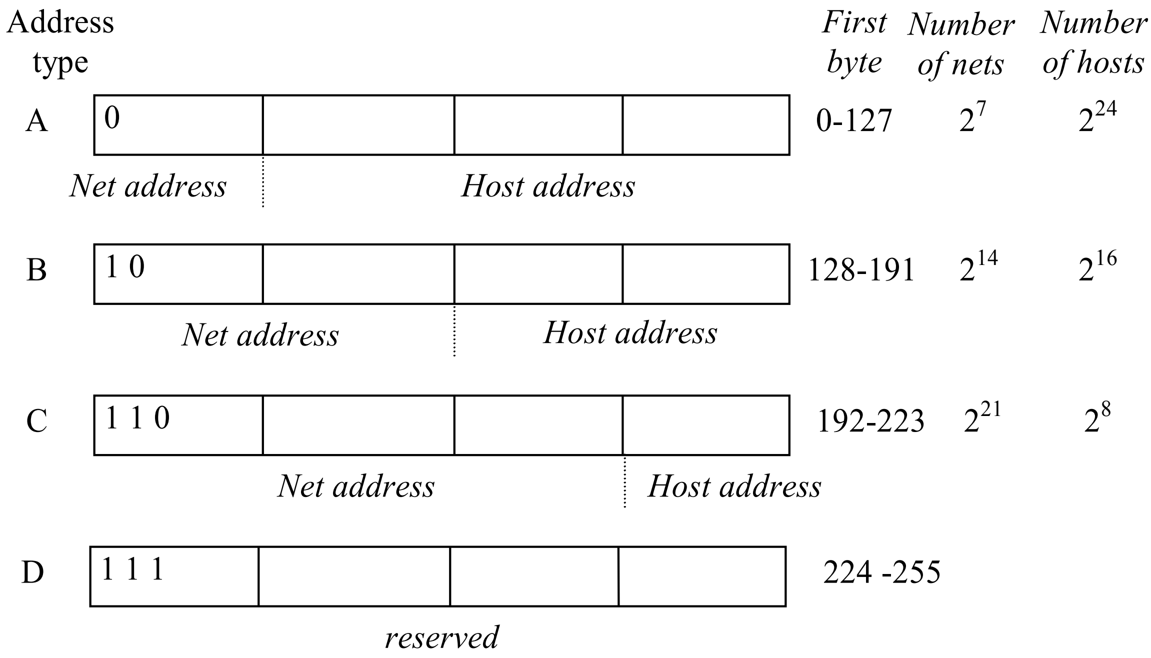
IP address

IP address has 32 bits. It consists of net address and host address.
First 3 bits define type of address.
IP address is divided into net address and host address according to its type.

8 bits	8 bits	8 bits	8 bits
193	84	34	18

IP adress : 193.84.34.18 (hex C1542212)

IP address types

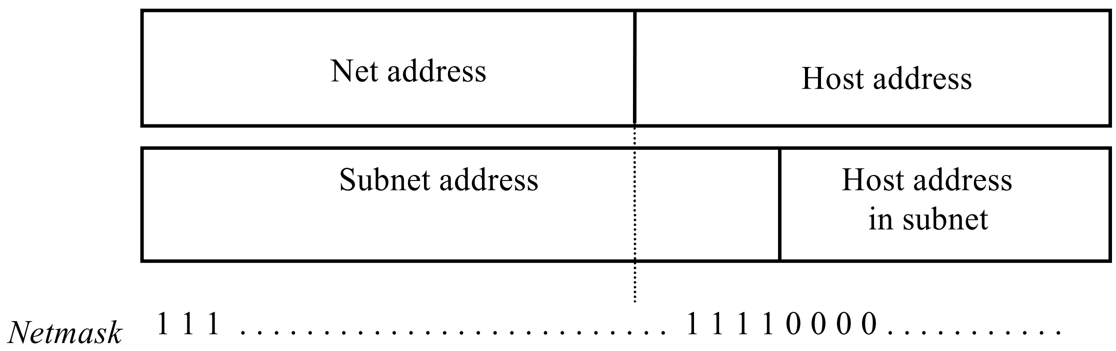


Internet addresses with special meaning

1. ***Net address***: all bits in host part equal 0 (e.g. 151.30.0.0)
2. ***Loopback address***: 127.0.0.0
3. ***Default direction of routing***: 0.0.0.0
4. ***Broadcast address***: all bits in host part equal 1 (e.g. 151.30.255.255)

Subnets

1. Lack of Internet addresses led to incorporation of subnet facility into IP protocol (Different local nets must have different internet addresses)
2. Host part of address is divided into subnet address and host address in this subnet. Division is done according to a netmask.



Example

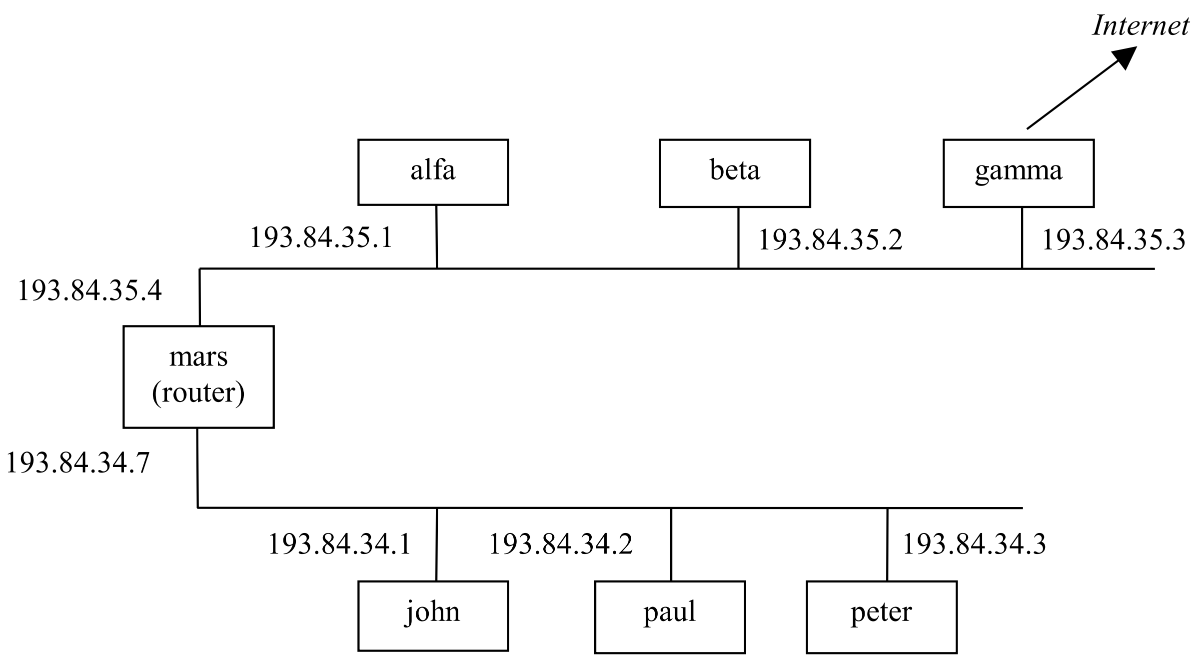
In the net 193.84.34.0 the first 3 bits of the host part form address of subnet.

Then netmask must be set to 255.255.255.224.

Broadcast address must be set to 193.84.34.31.

Routing

1. Routing facility provides Internet layer.
2. Internet consists of local nets, which are connected via computers called routers or gateways.
3. Routers are able to transfer datagrams from one local net to another. Transfer of datagrams is also called datagram switching (packet switching).



OS makes routing according to its **routing table**

Configuration of a host net interface

IP address is written into Internet layer of OS and routing table is initialized.

ifconfig *interface* *IP_address*

ifconfig associate *IP_address* with the interface *interface* and activates it

Standard names of interfaces:

lo0	loopback
le0, le1, . . .	Ethernet interfaces

Example (host john)

```
ifconfig  le0  193.84.34.1
```

or if net is divided into subnets:

```
ifconfig le0 193.84.34.1 netmask 255.255.255.224\
        broadcast 193.84.34.31
```

Routing table after **ifconfig** command

Destination	Gateway	Flags	Refcnt	Use	Interface
127.0.0.1	127.0.0.1	UH	1	140	lo0
193.84.34.0	193.84.34.1	U	12	4523	le0

List of routing table

```
netstat -nr
```

`U` (up)	routing is active
`H` (host)	only one host is reachable
`G` (gateway)	routing via router

After configuration of host net interface, the host can communicate in the local network

Add communication via router

Routing could be:

Static - Routing table is set by command **route**, usually contained in one of start scripts

Dynamic - Daemon **routed** or g**ated** is running. Daemon configures route table in collaboration with other routers in the net according to the immediate situation. Special protocols for this task are used.

Static routing

route **[-net]** **add** *target* *gateway* **1 | 0**

target	IP address of target host or target net or `default`
gateway	IP address of the router
0	gateway is net interface of local host
1	gateway is another host in local network

Delete row from routing table

route **[-net]** **del** *target* *gateway*

Example (host john)

```
route  add  default  193.84.34.7  1
```

Routing table after ifconfig command and route command:

Destination	Gateway	Flags	Rfcnt	Use	Interface
127.0.0.1	127.0.0.1	UH	1	140	lo0
default	193.84.34.7	UG	8	3765	le0
193.84.34.0	193.84.34.1	U	12	4523	le0

Example

Configuration of **mars**

```
ifconfig   le0   193.84.34.7
ifconfig   le1   193.84.35.4
route   add  -net   193.84.35.0   193.84.35.4
route   add  -net   193.84.34.0   193.84.34.7
```

Routing table of mars after configuration

Destination	Gateway	Flags	Rfcnt	Use	Interface
127.0.0.1	127.0.0.1	UH	1	120	lo0
193.84.34.0	193.84.34.7	U	17	8508	le0
193.84.35.0	193.84.35.4	U	6	7642	le1

Example

Net `193.84.35.0` is connected to Internet via router **gamma (**193.84.35.3). Then into routing table of **mars** routing via router **gamma** must be added

```
route  add  default  193.84.35.3   1
```

Routing table of mars after configuration

Destination	Gateway	Flags	Rfcnt	Use	Interface
127.0.0.1	127.0.0.1	UH	1	120	lo0
193.84.34.0	193.84.34.7	U	17	8508	le0
193.84.35.0	193.84.35.4	U	6	7642	le1
default	193.84.35.3	UG	11	9876	le1

Ports

On a host usually more servers are running. For identification of servers numbers called ports are used.

A lot of Internet applications have been designed and many of them are in use.

These applications have:

- *permanently assigned ports*
- *dynamically assigned ports*

Applications with permanently assigned ports are e.g. **ftp**, **telnet**, **www**, **finger**

Applications with dynamically assigned ports are based on Sun Microsystems software package RPC (Remote Procedure Calls), e.g. applications NFS (Network File System), NIS (Network Information Service), etc.

Port assignment to applications with dynamically assigned ports:

Daemon **portmapper** must be running in the system. If server of an application starts running, it must ask portmapper for port number. The given port number is registered in portmapper table.

Client must at first build up connection with portmapper and ask it for port number of the server. After receiving the port number, client could establish connection with the server.

Identification of connections

On a host many servers and clients with established connections to other hosts can be running. Connections are identified by means of associations.

Association:

(*protocol*, *local address IP*, *local port*, *remote address IP*, *remote port*)

Example

ftp client at **john** tries to connect to ftp server at **mars**

protocol: tcp

remote IP : **mars**

remote port: 21 (number reserved for ftp service)

local IP: **john**

local port : here OS puts a number unique at **john** (e.g. 1000)

association:

```
(tcp, mars, 21, john, 1000)
```

If another process on **john** will ask ftp server on **mars** then a new association may be:

```
(tcp, mars, 21, john, 1001)
```

If some process on **peter** will ask ftp server on **mars** then association may be:

```
(tcp, mars, 21, peter, 1000)
```

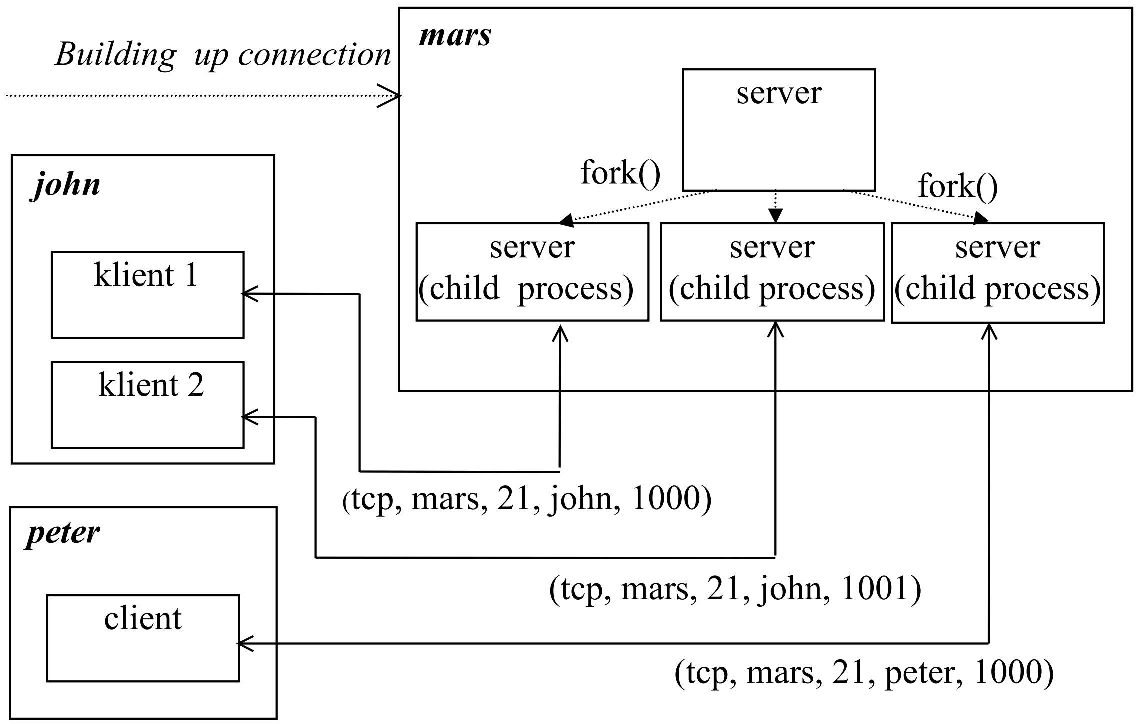


Fig. Process associations.

Only some servers are running all the time: e.g. routed, named, name server, sendmail etc.

Most servers begin to run after a request has come: telnet, ftp, talk, finger etc.

These servers are started by Internet demon **inetd**

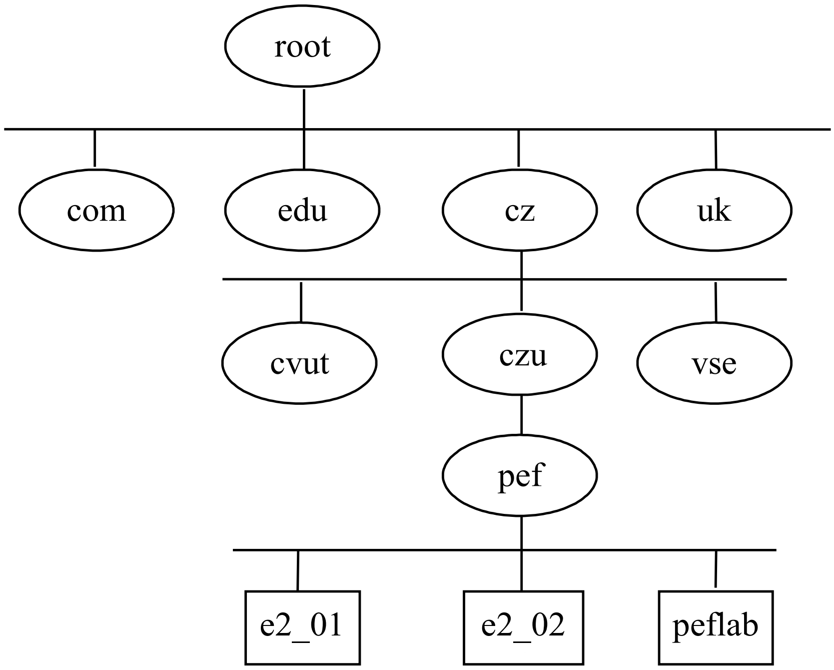
Configuration files of **inetd**:

```
/etc/inetd.conf
/etc/services
```

Domain Name System (DNS)

Using IP address is cumbersome. Therefore DNS was developed.

Hosts are organized into hierarchical domain structure.



Host identification

- The path going to a host from root identifies it in the domain structure.
- Root is denoted as **.**
- Path is written as a host name following by the sequence of domain names that are on the path from the host to the root.
- Names are separated by dots.
- The root domain **.** is usually omitted.

Example

Host peflab in domain pef is identified by the path

peflab.pef.czu.cz

Name servers

Every domain has its **primary name server** and one or more **secondary name servers.**

Secondary name servers maintain copies of primary server data. They read primary name server data periodically and substitute primary name server if necessary.

Primary name server must know:

- IP addresses of all hosts of its domain
- IP addresses of name servers of all its subdomains

Communication using domain names

If any process wants to communicate with a host that is identified by its domain address, it must at first find out its IP address.

In order to do that, it must ask some name server (usually the name server of its domain):

- If the host with unknown IP address resides in the same domain as the asked name server, the answer is straight.
- If not the name server will find out the IP address in collaboration with name servers of other domains.

OS support communication with name server by functions contained in **resolver library**.

The main function that establishes connection to the name server, gives it domain address and takes over IP address is **gethostbyname()**

Resolver has to be configured:

Configuration files are

`/etc/host.conf` `/etc/resolv.conf` `/etc/hosts`

Usual configuration (specified in `/etc/host.conf`) is:

1. Look through file `/etc/hosts`
2. If the IP address has not been found, ask name servers specified in `/etc/resolv.conf`

Network Information System (NIS)

NIS enables sharing of some important chosen system files, e.g. /etc/passwd , /etc/group , /etc/hosts etc.

NIS is based on Sun Microsystems remote procedure call software package. Its original name was Yellow Pages.

NIS architecture

1. NIS architecture is client-server.
2. NIS server creates and maintains shared files.
3. NIS clients are running processes that ask NIS server for data.
4. NIS server shared files are organized in so called maps that enable direct access to their data. For example `/etc/passwd` is organized in two maps: `passwd.byname` that enables direct access using username `passwd.byuid` that enables direct access using UID

Content of a shared file can be listed by command **ypcat**

A change of a password in the shared password file can be done by command **yppasswd**

Network File System (NFS)

NFS enables to access to files that reside on a remote host in the same manner as to the local files

User can mount directories of a remote host in the similar manner as local directories, i.e. using command **mount**

Example

```
mount -t nfs mars:/usr /home
```

User can mount from a remote host only such directories that are on remote host exported.

Export of directories on the remote host is done by proper setting of the file `/etc/exports`

Export could be set with different restrictions, e.g. directory can be exported only for reading.

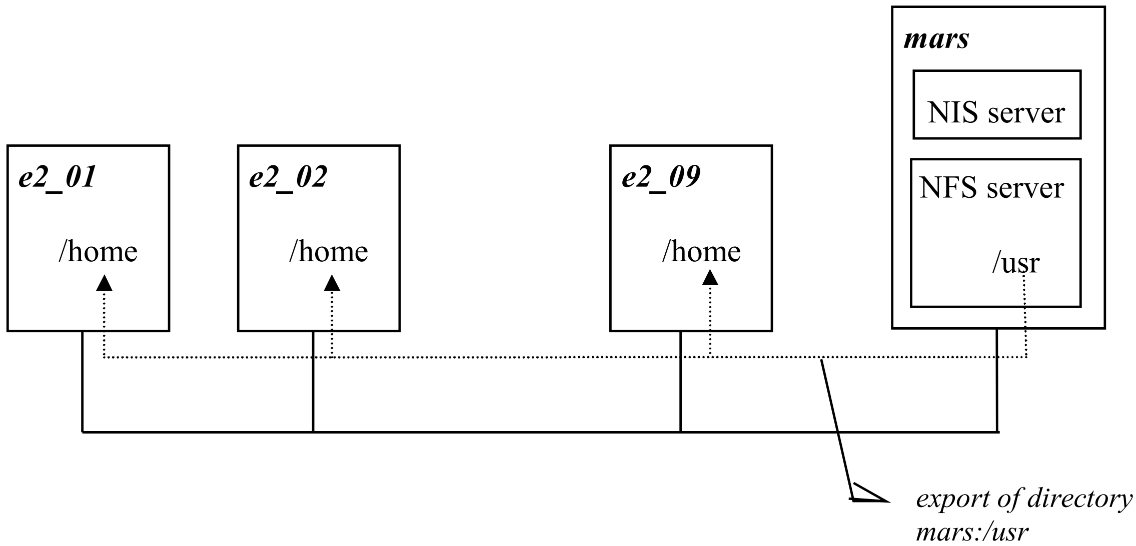
NFS architecture

NFS architecture is client-server

Server is realized by daemon **mountd** and one or more daemons **nfsd** (for each client one nfsd daemon must be running)

Client is realized by daemons **biod**. Some OS support client architecture directly and daemons **biod** are not used.

Example
Using NIS and NFS in computing laboratory



References

Bach, M.J., 1986: *The Design of the UNIX Operating System*, Englewood Cliffs, Prentice Hall.

Hunt, C., 1994: *TCP/IP Network Administration*, O´Reilly&Associates.

Leffler, S.J., McKusick, M.K., Karels, M.J., Quarterman, J.S., 1989: *The Design and Implementation of the 4.3BSD UNIX Operating System*, New York , Addison-Wesley.

Tanenbaum, A., 1992: *Modern Operating Systems*, Englewood Cliffs, Prentice Hall.