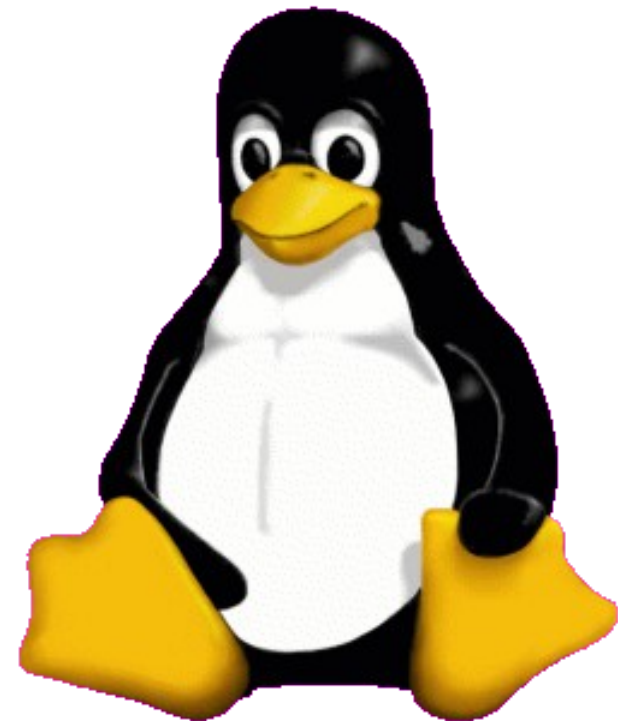
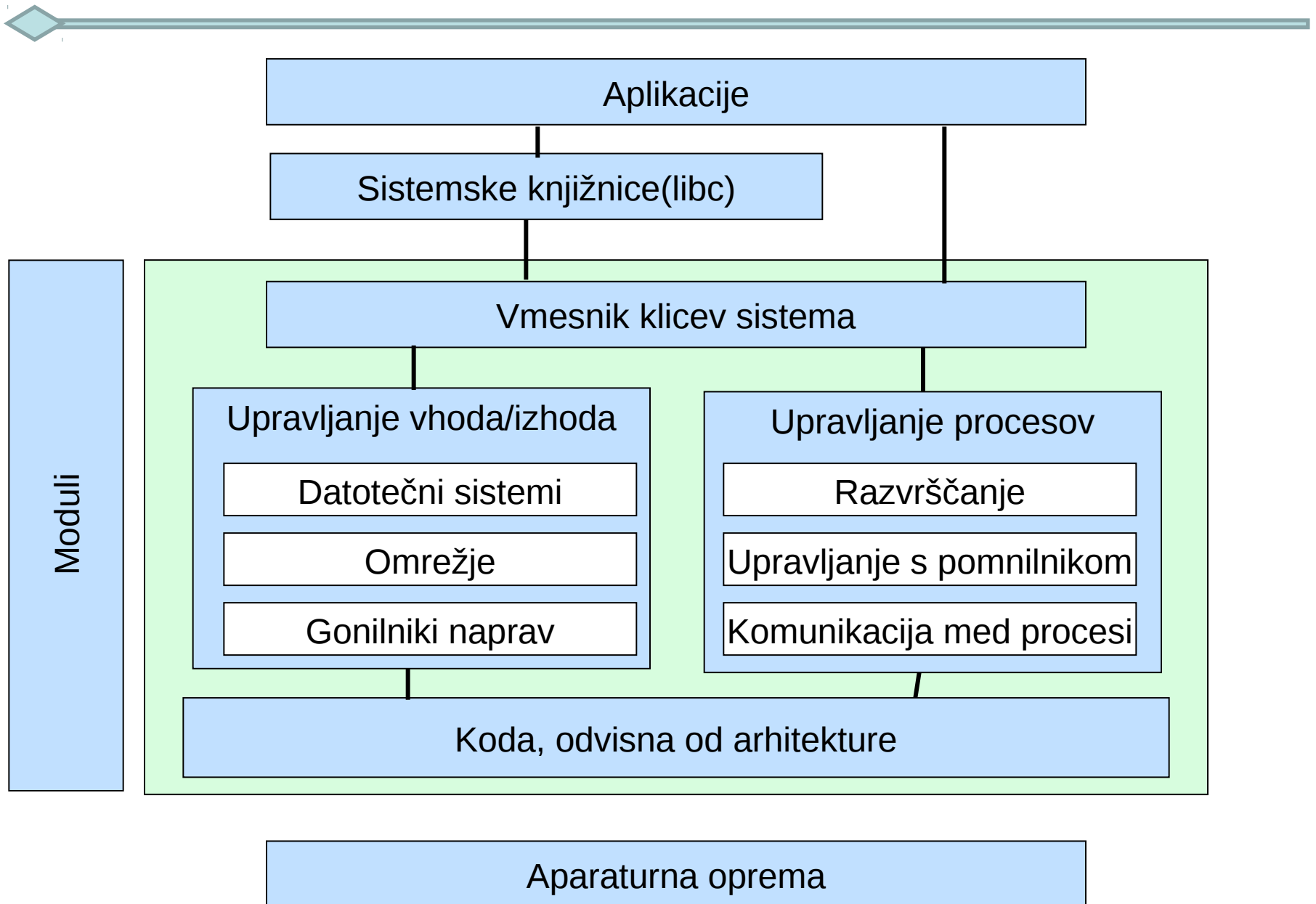


**Systemski klici**

# The Kernel API - System Calls



# Komponente sistema LINUX



# Sistemske klici za delo z datotekami

System call	Description
<code>fd = creat(name, mode)</code>	One way to create a new file
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &amp;buf)</code>	Get a file's status information
<code>s = fstat(fd, &amp;buf)</code>	Get a file's status information
<code>s = pipe(&amp;fd[0])</code>	Create a pipe
<code>s = fcntl(fd, cmd, ...)</code>	File locking and other operations

- **s** je koda napake
- **fd** je opisnik datoteke
- **position** je položaj v datoteki

# Delo z datotekami

**Delček kode, ki ponazarja tipično zaporedje dogodkov:**

```
int fd; /*File descriptor */  
...  
fd = open (fileName, ...); /* Open file, return file descriptor */  
    if (fd == -1) { /* Set some error condition */ }  
...  
read (fd, ...); /* Read from file */  
...  
write (fd, ...); /* Write to file */  
...  
lseek (fd, ...); /* Seek within file */  
close (fd); /* Close the file, freeing file descriptor */
```

# Sistemske kliče za zaščito datotek

System call	Description
<code>s = chmod(path, mode)</code>	Change a file's protection mode
<code>s = access(path, mode)</code>	Check access using the real UID and GID
<code>uid = getuid( )</code>	Get the real UID
<code>uid = geteuid( )</code>	Get the effective UID
<code>gid = getgid( )</code>	Get the real GID
<code>gid = getegid( )</code>	Get the effective GID
<code>s = chown(path, owner, group)</code>	Change owner and group
<code>s = setuid(uid)</code>	Set the UID
<code>s = setgid(gid)</code>	Set the GID

- **s** je koda napake
- **uid** in **gid** sta identifikatorja uporabnika in skupine

# Sistemski klici za upravljanje z direktoriji

System call	Description
<code>s = mkdir(path, mode)</code>	Create a new directory
<code>s = rmdir(path)</code>	Remove a directory
<code>s = link(oldpath, newpath)</code>	Create a link to an existing file
<code>s = unlink(path)</code>	Unlink a file
<code>s = chdir(path)</code>	Change the working directory
<code>dir = opendir(path)</code>	Open a directory for reading
<code>s = closedir(dir)</code>	Close a directory
<code>dirent = readdir(dir)</code>	Read one directory entry
<code>rewinddir(dir)</code>	Rewind a directory so it can be reread

- **s** je koda napake
- **dir** identificira direktorij
- **dirent** je vhod direktorija

# Nekaj sistemskih klicev za različne naloge

## Miscellaneous

Call	Description
<code>s = chdir(dirname)</code>	Change the working directory
<code>s = chmod(name, mode)</code>	Change a file's protection bits
<code>s = kill(pid, signal)</code>	Send a signal to a process
<code>seconds = time(&amp;seconds)</code>	Get the elapsed time since Jan. 1, 1970



# Primer: c simulacija linux ukaza "cd"

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

main(int argc,char **argv)  {
    if (argc < 2) {
        printf("Usage: %s <pathname> \n",argv[0]);
        exit(1);
    } if (chdir(argv[1]) != 0) {
        printf(`Error in chdir n"); exit(1);
    }
}
```

# Upravljanje programskih procesov

## **int fork()**

tvorba novega procesa (otroka), ki je (ob rojstvu) enak svojemu očetu, od katerega se razlikuje le po svoji, specifični procesni številki (PID). Ker od trenutka rojstva oba procesa (oče in otrok) živita ločeno, lahko spoznata, kdo je kdo (oče ali otrok) po tem, da očetu vrne klic fork vrednost PID procesa-otroka, otroku pa vrne vrednost 0.

**getpid()** vrne PID kličočega procesa,

**getppid()** vrne PID njegovega očeta.

## **int wait( int \* status)**

Status je kazalec na celoštevilčno vrednost, v katero shrani UNIX vrednost, ki jo vrača ob svojem koncu proces-otrok.

## **void exit (int status)**

Pomen spremenljivke status smo že pojasnili pri opisu klica wait().

## **int execv(char \* fileName, \*argv[ ])**

*fileName* je ime izvršljive (programske) datoteke, ki naj se transformira v proces. V polju *arg* pa so argumenti, ki jih temu procesu posredujemo.

# Sistemske klici za upravljanje procesov

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

**s** je koda napake

**pid** je identifikator procesa

**residual** je preostali čas od zadnjega alarma

# Animirana demonstracija

— /users/colos/PARIS/VII

PID: 2013    PPID: 2012    State: Finished

SIGUSR1    SIGUSR2    SIGCHLD    SIGPIPE

```
b=13;
if ( fork() == 0 )
{
  a=a+1;
  b=b-1;
  c=a+b;
  exit(0);
}
else
{
  a=a-1;
  b=b+1;
  c=a+b;
  a=a-1;
  b=b+1;
  exit(0);
}
}
```

Process data

a	=	12
b	=	15
c	=	26

— /users/colos/PARIS/VII

PID: 3870    PPID: 2013    State: Zombie

SIGUSR1    SIGUSR2    SIGCHLD    SIGPIPE

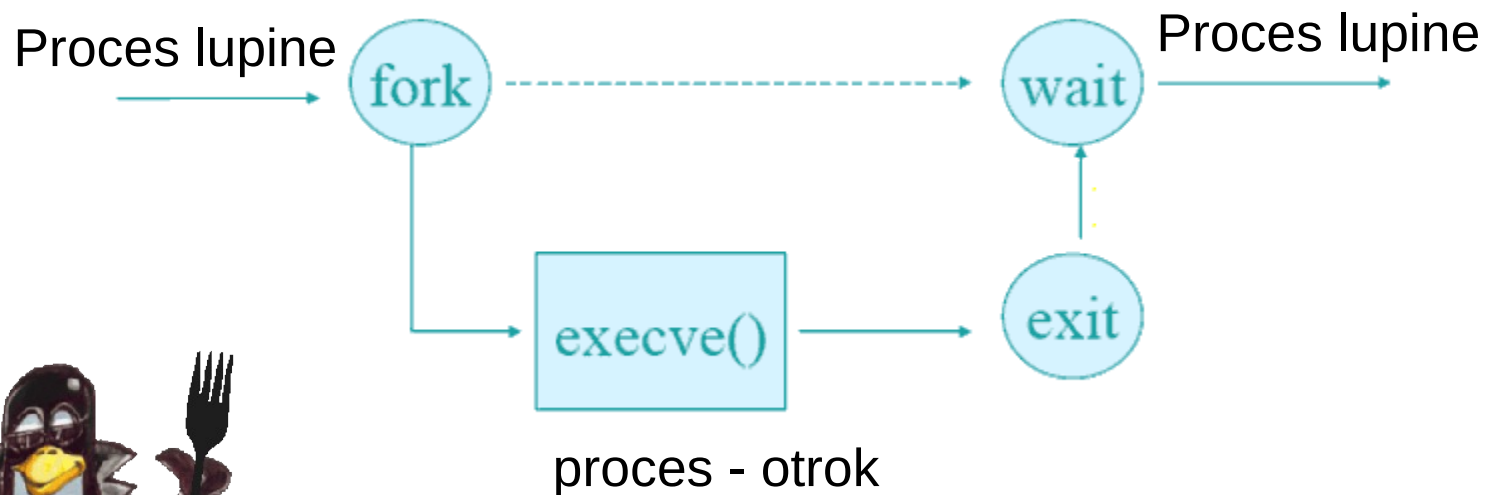
```
/*test3.c*/
void main ()
{
  int a, b, c;
  a=12;
  b=13;
  if ( fork() == 0 )
  {
    a=a+1;
    b=b-1;
    c=a+b;
    exit(0);
  }
  else
  {
    a=a-1;
    b=b+1;
    c=a+b;
  }
}
```

Process data

c	=	26
b	=	13
a	=	13

# Kako deluje lupina LINUX

- Interpreter ukazne vrstice
- Vgrajeni ukazi
- Zunanji ukazi



# Poenostavljena koda lupine

```
while (TRUE) {
    type_prompt( );
    read_command(command, params);

    pid = fork( );
    if (pid < 0) {
        printf("Unable to fork0);
        continue;
    }

    if (pid != 0) {
        waitpid (-1, &status, 0);
    } else {
        execve(command, params, 0);
    }
}
```

*/\* repeat forever \*/*  
*/\* display prompt on the screen \*/*  
*/\* read input line from keyboard \*/*  
*/\* fork off a child process \*/*  
*/\* error condition \*/*  
*/\* repeat the loop \*/*  
*/\* parent waits for child \*/*  
*/\* child does the work \*/*

# Procesni signali

**signal (signame,SIG\_IGN);**  
**signal (SIGINT, funkcija)**

SIGHUP	Ta signal je posredovan procesom, katerih terminal je bil izklopljen.
SIGINT	Prekinitveni zahtevek s tastature terminala
SIGILL	Nelegalna instrukcija
SIGFPE	" <i>Floating point</i> " napaka: delitev z 0, prekoračitev in podobno
SIGKILL	<i>Kill</i> . Ta signal lahko ignoriramo, ujamemo ali blokiramo.
SIGSYS	Napačen argument v sistemskem klicu
SIGPIPE	Pisanje v cev, ki je nihče ne bere
SIGALRM	Signal "alarmne ure"

# Kontrola časa

**localtime()** Vrne lokalni čas.

```
longint t;  
.....  
time(t);  
printf(" time:\%s" ,asctime( localtime (t)));
```

**sleep( int secs)**

Proces bo zaspal za dano število sekund

**alarm(int secs),**

Čez koliko sekund dvignemo alarmni signal *SIGALRM*.

S klicem **alarm(0)**, torej z argumentom z vrednostjo 0, nastop alarma preprečimo.



# Primer uporabe funkcije time in localtime

```
#include <time.h>
#include <stdio.h>
#define SIZE 256

int main (void) {
    char buffer[SIZE];
    time_t curtime;
    struct tm *loctime;

    /* Get the current time. */
    curtime = time (NULL);

    /* Convert it to local time representation. */
    loctime = localtime (&curtime);

    /* Print out the date and time in the standard format. */
    fputs (asctime (loctime), stdout);
    /* Print it out in a nice format. */
    strftime (buffer, SIZE, "Today is %A, %B %d.\n", loctime);
    fputs (buffer, stdout);
    strftime (buffer, SIZE, "The time is %l:%M %p.\n", loctime);
    fputs (buffer, stdout);
    return 0;
}
```

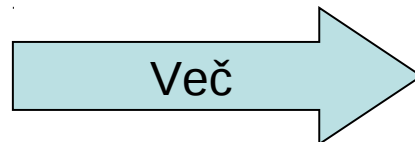
Kakšen bo izpis:

```
Wed Jul 31 13:02:36 1991
Today is Wednesday, July
31. The time is 01:02 PM.
```

# Podatki o uporabnikih

---

- getlogin()** Vrne vstopno (login) ime uporabnika
- getuid()** Vrne identiteto (UID) uporabnika procesa
- setuid()** Spreminjanje identitete (UID) uporabnika
- getuid()** Vrne ident skupine (grupe) uporabnikov



# Primer getlogin()

```
/* implementing who am i using system calls */

#include<stdio.h>
#include<utmp.h>
int main() {
    char *s,*c;
    struct utmp *u;
    int i;
    c=getlogin();
    setutent();
    u=getutent();
    while(u!=NULL) {
        if(u->ut_type==7 && strcmp(u->ut_user,c)==0) {
            printf("%-12s",u->ut_user);
            printf("%-9s",u->ut_line);
            s=ctime(&u->ut_time);
            for(i=4;i<16;i++)
                printf("%c",s[i]);
            printf("(%s",u->ut_host);
            printf(") ");
        }
        u=getutent();
    }
}
```

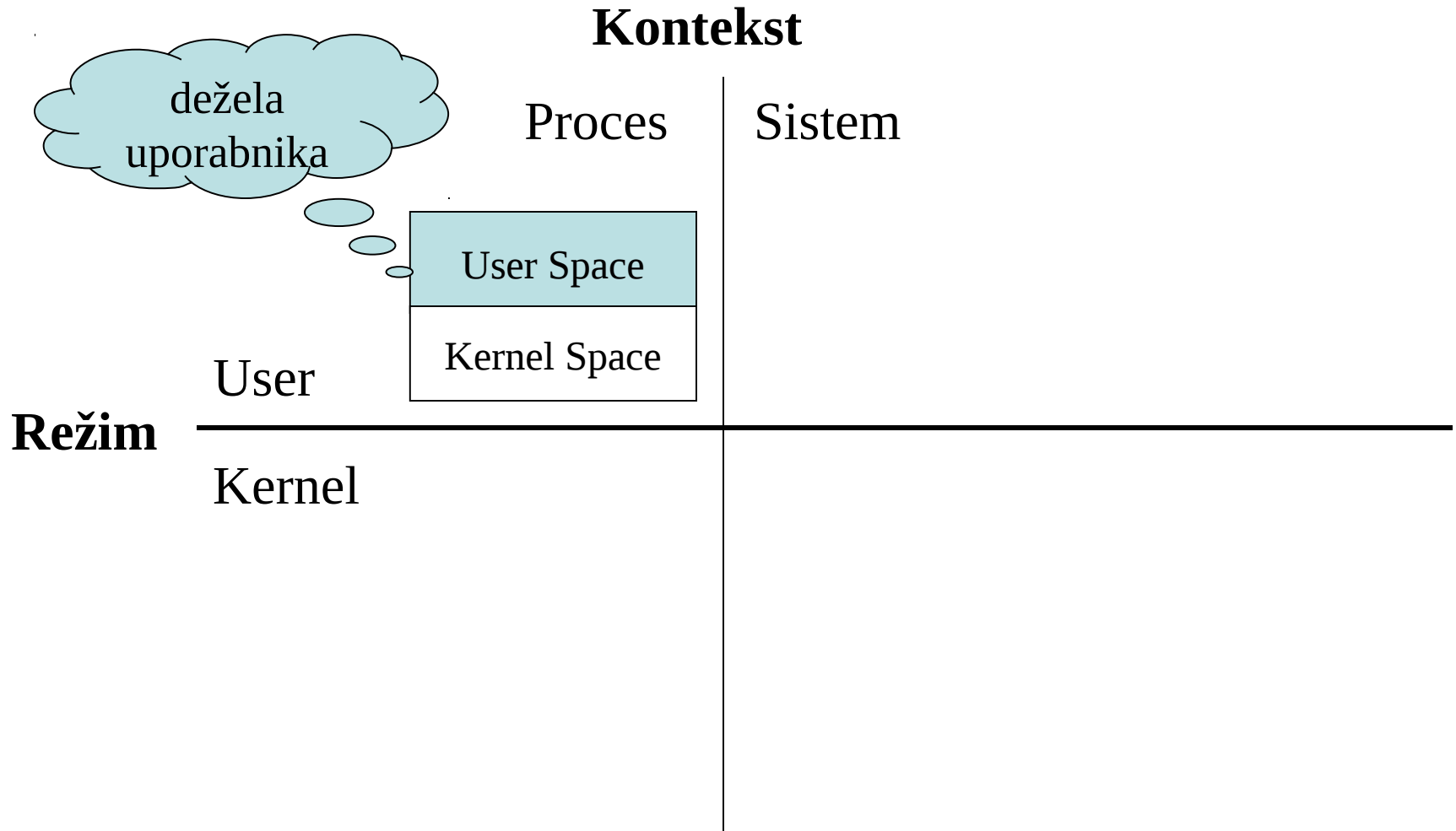
# Primerjava sist.klicev UNIX in Win32

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

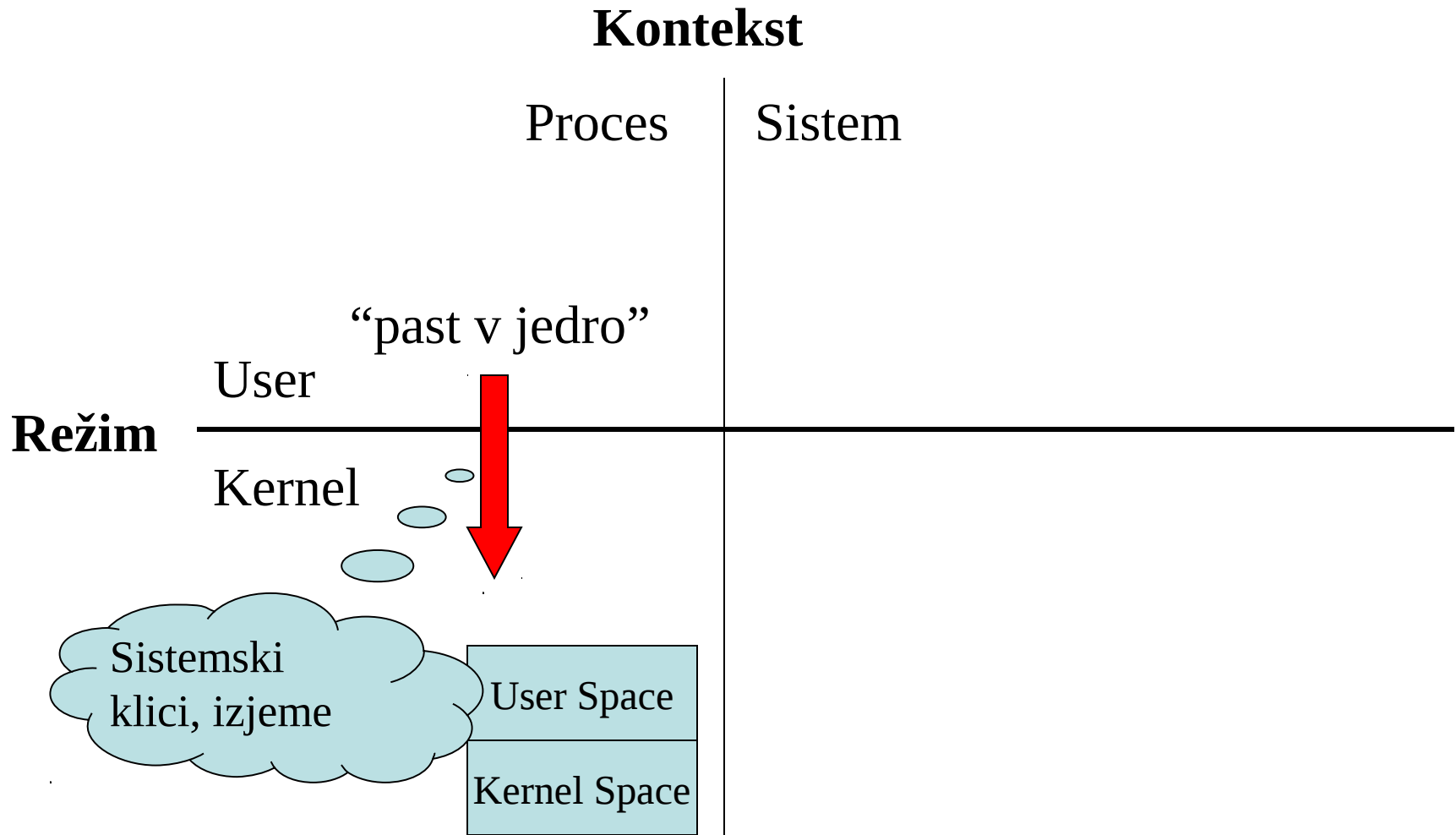
# Režim, prostor, kontekst

- **Režim (mode):** z aparaturno opremo omejeno stanje izvajanja
  - omejen dostop, privilegirane instrukcije
  - Uporabniški in sistemski režim (user mode, kernel mode)
    - “sistemski režim je privilegiran
  - Intel podpira 4 obročne zaščite: 0 kernel, 1 unused, 2 unused, 3 user
- **prostor:** naslovni prostor sistema in uporabnika
  - zahteva podporo MMU (navidezni pomnilnik)
  - “dežela uporabnika”: naslovni prostor vsakega procesa; imamo več uporabniških naslovnih prostorov
  - jedro je pogosto preslikano v naslovni prostor uporabnika
- **Kontekst:** aktivnost jedra “v imenu...”
  - proces: v imenu tekočega procesa
  - sistem: nevezan na tekoči proces (morda nimamo nobenega procesa!)
    - primer “kontekst prekinitve”
    - blokiranje ni dopustno!

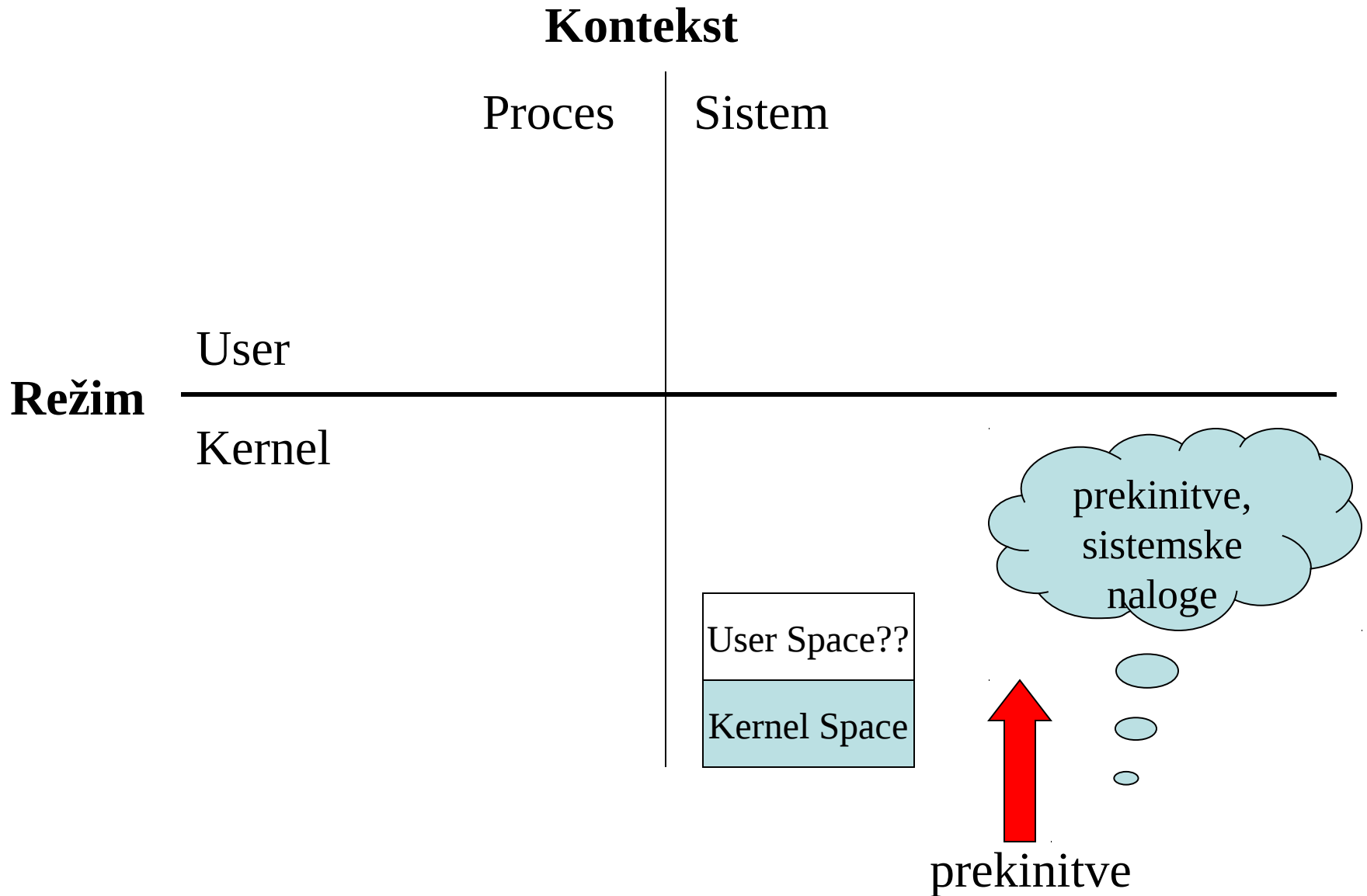
# Uporabniški režim, kontekst procesa



# Režim jedra, kontekst procesa



# Režim jedra, Kontekst sistema





# Prečkanje “meje jedra”

- Več kot le klic procedure
- manj kot preklop konteksta
- cena:
  - Vzpostavljanje sklada jedra
  - validacija parametrov
  - jedro preslikano v naslovni prostor uporabnika?
    - dovoljenja za ažuriranje preslikav strani
  - Jedro v ločenem naslovnem prostoru?
    - ponovno nalaganje preslikav strani
    - razveljavitev "cache"

# Povzetek

---

- Sistemski klici predstavljajo "API" do jedra.
- Sistemski klici so način vstopanja v "zaščiteni režim".
- Prečkanje "meje" jedra je drago.
- Sistemski klici so pogosto oviti v rutine knjižnic.
- Blokiranje "upočasni" sistemske klice, ki bi jih lahko prekinjali signali.
- Sistemskim klicem lahko "sledimo" ( `ptrace()` ).
- Intel Linux implementira sistemske klice s pomočjo prekinitev in nizkonivojske možnosti, ki ji pravimo "call gate".
- Sistemski klici pogosto zahtevajo kopiranje podatkov med uporabniškim in sistemskim naslovnim prostorom.