

Signály

signály umožňují oznámit procesům výskyt událostí v systému

jde o krátké zprávy, kde se procesům oznámí číslo signálu

systémová volání pro signály i vnitřní implementace se u jednotlivých variant a verzí značně liší, System V vs. BSD

problémy:

pro tvůrce přenositelných aplikací – může používat taková volání která jsou všude stejná

pro výrobce operačních systémů, které chtějí být kompatibilní s více variantami – musí poskytovat všechna systémová volání

standardní rozhraní specifikuje POSIX, včetně zpětné kompatibility







čísla některých signálů závisí na HW, označují se symbolickými konstantami SIG...

signály slouží dvěma hlavními účelům

- uvědomit proces, že nastala určitá událost
- přinutit proces vykonat funkci na zpracování signálu (*signal handler*)

systémová volání umožňují programátorovi zasílat signály a určit jak budou použity

POSIX

Signal	Default Action	Description
SIGABRT	A	Process abort signal.
SIGALRM	T	Alarm clock.
SIGBUS	A	Access to an undefined portion of a memory object.
SIGCHLD	I	Child process terminated, stopped,
[XSI] 		or continued. 
SIGCONT	C	Continue executing, if stopped.
SIGFPE	A	Erroneous arithmetic operation.
SIGHUP	T	Hangup.
SIGILL	A	Illegal instruction.
SIGINT	T	Terminal interrupt signal.
SIGKILL	T	Kill (cannot be caught or ignored).
SIGPIPE	T	Write on a pipe with no one to read it.
SIGQUIT	A	Terminal quit signal.
SIGSEGV	A	Invalid memory reference.
SIGSTOP	S	Stop executing (cannot be caught or ignored).
SIGTERM	T	Termination signal.
SIGTSTP	S	Terminal stop signal.
SIGTTIN	S	Background process attempting read.
SIGTTOU	S	Background process attempting write.
SIGUSR1	T	User-defined signal 1.
SIGUSR2	T	User-defined signal 2.
[XSI]  SIGPOLL	T	Pollable event.
SIGPROF	T	Profiling timer expired.
SIGSYS	A	Bad system call.
SIGTRAP	A	Trace/breakpoint trap. 
SIGURG	I	High bandwidth data is available at a socket.
[XSI]  SIGVTALRM	T	Virtual timer expired.
SIGXCPU	A	CPU time limit exceeded.
SIGXFSZ	A	File size limit exceeded. 

některé signály jsou vzhledem k procesu asynchronní
SIGINT, přerušení od terminálu stisknutím CTRL-C

jiné jsou synchronní
SIGSEV, chyba odkazu na stránku

jádro při zasílání signálů rozlišuje dvě fáze:

- odeslání signálu
 jádro zaznamená v záznamu **proc** (deskriptoru
 procesu) zasílanému procesu odeslání nového
 signálu
- přijetí signálu
 jádro přinutí proces reagovat na signál

pro každý signál je nastavená implicitní reakce, která se vykoná, pokud ji proces nespecifikuje jinak

- T** (*abnormal*) *termination*, také *abort*, *exit*
proces je násilně ukončen se všemi důsledky volání **exit(stav)**, přitom **stav** indikuje pro **wait()** a **waitpid()** abnormální ukončení
- A** *abnormal termination*, také *dump*, *abort*
navíc se vykoná nějaká akce, typicky výpis obsahu paměti procesu a hodnot registrů do souboru s názvem **core**
- I** *ignore*, signál je ignorovaný
- S** *stop*, proces je zastaven (*stopped*)
- C** *continue*, byl-li proces zastaven, může pokračovat, je převeden do stavu připraven, jinak je signál ignorován

proces může potlačit nastavenou akci a specifikovat jinou akci

- explicitně ignorovat signál
- zachytit signál a vykonat uživatelem definovanou funkci, která se nazývá, ošetření/obsluha signálu (*signal handler*)

na druhé straně, proces může obnovit reakci na signál na nastavenou implicitní akci

proces může signál blokovat, co znamená, že signál nebude přijat dokud signál není odblokován

signály SIGKILL a SIGSTOP nemůžou uživatelé ignorovat, blokovat nebo specifikovat pro ně obsluhu

signál je nevyřízen (*pending*), byl-li odeslán, ale nebyl přijat jenom jeden signál každého typu může být nevyřízen

reakci na signál vykonává proces, kterému je signál zaslán, včetně ukončení procesu, to znamená, že musí být aspoň plánován stát se běžícím

má-li nízkou prioritu může mezi odesláním signálu a jeho přijetím, kdy se vykoná odpovídající akce uplynout dosti dlouhá doba

další prodlení může způsobit je-li proces v čase odeslání signálu ve stavu

- **zastaven**
- **spící**

signály pro zastavení procesu (*stop signals*) **SIGSTOP**, **SIGTSTP**, **SIGTIN**, **SIGTOUT** mění okamžitě stav procesu na **zastaven** nebo **spící_a_zastaven** a signál **SIGCONT** je vrací do původního stavu

co se má stát, když je odeslán signál spícímu procesu?

činnost jádra záleží na tom proč proces přešel do stavu **spící**

- čeká-li na událost, která zakrátko nastane, např. dokončení diskové V/V operace, je spící v kategorii nepřerušitelný a signál je pouze zaznačen jako nevyřízen
- čeká-li na událost, o které nevíme kdy nastane nebo dokonce nemusí nastat vůbec, např. skončení potomka, vstup z terminálu, je spící v kategorii přerušitelný, je jádrem vzbuzen a přejde do stavu připraven

Linux nemá stav spící, ale stavy **úloha_přerušitelná** (***TASK_INTERRUPTIBLE***) **úloha_nepřerušitelná** (***TASK_UNINTERRUPTIBLE***)

přijímající proces je přinucen vykonat odpovídající akci, když pro něj jádro zavolá funkci **issig()** na zjištění nevyřízených signálů

jádro zavolá **issig()** :

- před návratem do uživatelského módu ze systémového volání nebo z obsluhy přerušení
- před zablokováním procesu v přerušitelné kategorii
- když se stane běžícím po vzbuzení ze stavu spící v přerušitelné kategorii

když proces začal vykonávat systémové volání a nastane některý z posledních dvou případů, proces přijme signál a namísto dokončení systémového volání vykoná obsluhu signálu a systémové volání se obvykle vrátí s hodnotou **EINTR** v proměnné **errno**

scénář asynchronního signálu

- uživatel stiskne **CTRL-C**
- generuje se přerušení (jako u každého stisknutí klávesy)
- ovladač rozpozná, že jde o kombinaci generující signál a odešle signál **SIGINT** procesu v popředí
- když je proces naplánována jako běžící při návratu do uživatelského módu anebo byl-li běžící při návratu z přerušení proces najde signál

scénář synchronního signálu

- výjimka (dělení nulou, nedovolená instrukce,..) způsobí přechod do módu jádro
- jádro vykoná její obsluhu a zašle se odpovídající signál běžícímu procesu
- při návratu z obsluhy proces najde signál

Nespolehlivé signály

funkce pro obsluhu signálů nejsou perzistentní, po zachycení (nalezení) signálu, jádro ještě před vyvoláním funkce obsluhy signálu nastaví implicitní akci, tedy pro následující signál, chceme-li opět vykonat obslužní funkci musíme ji znovu instalovat, vzniká soutěž (*race condition*)

instalace obslužní funkce signálu

```
oldfunction=signal(sig, function) ;
```

function

- **SIG_IGN** ignorování, ne **SIGKILL**, **SIGSTOP**
- **SIG_DFL** nastavit implicitní akci
- adresa obslužní funkce

sig číslo signálu

oldfunction předcházející obsluha

zaslání signálu

```
kill(pid, sig) ;
```

pid pid procesu, kterému bude zaslán signál (viz dále)

sig číslo signálu

Příklad

```
sig_obsluha()
{
    printf("signal zachycen");
    signal(SIGINT, sig_obsluha);
}

main()
{
    int rpid;

    signal(SIGINT, sig_obsluha);

    if (fork == 0)
    {
        sleep(5);
        rpid = getpid();
        for(;;)
            if (kill(rpid,SIGINT) == -1)
                exit(1);
    }

    /* snížíme prioritu */
    nice(10);
    for(;;)
        ;
}
```

rodičovský proces má nízkou prioritu a je-li mu odebrán procesor v obslužné funkci **sig_obsluha()** signálu **SIGINT** před opětovnou instalací obslužní funkce a potomek zašle další signál, proces rodič při jeho přijetí vykoná nastavenou implicitní akci, tj. **exit**

bylo by řešením nenastavovat implicitní akci?

ano, ale při obsluze signálu by mohla být vnořena další obsluha, ... a uživatelský zásobník by mohl přetéct

Spolehlivé signály

- perzistentní obslužné funkce signálů
- blokování signálu, např. při obsluze signálů → nevznikne hnízdění

zaslání signálu

kill(pid, sig);

pid > 0 signál je zaslán procesu s PID = pid

pid = 0 signál je zaslán všem procesům skupiny

pid = -1 signál je zaslán všem procesům, kromě 0, 1 a běžícího

pid < -1 signál je zaslán všem procesům v skupině -pid

instalace obslužní funkce (náhrada signal)

sigaction(sig, act, oact);

specifikuje obsluhu pro signál **sig**

act ukazuje na záznam, který obsahuje:

- akci – **SIG_IGN**, **SIG_DFL**, nebo obslužní funkci
- masku signálů, které mají být blokovány při vykonávání obslužní funkce
- příznaky
 - SA_NOCLDSTOP** negeneruj **SIGCHLD**, když je potomek zastaven
 - SA_RESTART** signálem přerušené systémové volání, se restartuje
 - SA_ONSTACK** obsluž signál v alternativním zásobníku deklarovaném voláním **sigaltstack()**
 - SA_RESETHAND** akce se nastaví na implicitní
 - SA_SIGINFO**
 - není-li nastaven obslužní funkce je zadána ve tvaru
func (sig);
 - je-li nastaven obslužní funkce je zadána ve tvaru
func (sig, info, kontext);
 - kde **info** vysvětluje příčinu vzniku signálu a **kontext** odkazuje na přerušný kontext procesu, když byl signál dodán

SA_NOCLDWAIT nevytvářej mátohy, když
potomci volajícího procesu skončí, zavolá-li
proces **wait()** čeká až všichni potomci skončí
SA_NODEFER neblokuj automaticky signál, když
bude obsluhován, jako nespolehlivé signály

oact volitelně vrátí předcházející akci signálu

zjištění nevyřízených signálů

sigpending(set) ;

modifikování blokováných signálů

sigprocmask(how, set, oset) ;

oset stará maska signálů

set nová maska signálů

how

SIG_BLOCK nová maska specifikuje signály, které se
přidají k blokováným

SIG_UNBLOCK nová maska specifikuje signály, kterých
blokování se odstraní

SIG_SETMASK nová maska specifikuje blokované
signály

čekání procesu na signál

`sigsuspend(sigmask)` ;

nastaví se blokované signály podle **`sigmask`** a proces přejde do stavu čekající až do zaslání signálu, který není blokován nebo ignorován

není ekvivalentní dvojici **`sigprocmask()`** a **`sleep()`** systémové volání **`sigprocmask()`** mohlo odblokovat signál, na který chceme čekat v **`sleep()`** a může se stát, že signál bude přijat před zavoláním **`sleep()`** a čekání nemusí skončit

obslužní funkce musí používat bezpečná (reentrantní) systémová volání

POSIX.1-2003

`_Exit()` `_exit()` `abort()` `accept()` `access()` `aio_error()` `aio_return()`
`aio_suspend()` `alarm()` `bind()` `cfgetispeed()` `cfgetospeed()` `cfsetispeed()`
`cfsetospeed()` `chdir()` `chmod()` `chown()` `clock_gettime()` `close()` `connect()`
`creat()` `dup()` `dup2()` `execle()` `execve()` `fchmod()` `fchown()` `fcntl()`
`fdatasync()` `fork()` `fpathconf()` `fstat()` `fsync()` `ftruncate()` `getegid()` `geteuid()`
`getgid()` `getgroups()` `getpeername()` `getpgrp()` `getpid()` `getppid()`
`getsockname()` `getsockopt()` `getuid()` `kill()` `link()` `listen()` `lseek()` `lstat()`
`mkdir()` `mkfifo()` `open()` `pathconf()` `pause()` `pipe()` `poll()`
`posix_trace_event()` `pselect()` `raise()` `read()` `readlink()` `recv()` `recvfrom()`
`recvmsg()` `rename()` `rmdir()` `select()` `sem_post()` `send()` `sendmsg()` `sendto()`
`setgid()` `setpgid()` `setsid()` `setsockopt()` `setuid()` `shutdown()` `sigaction()`
`sigaddset()` `sigdelset()` `sigemptyset()` `sigfillset()` `sigismember()` `signal()`
`sigpause()` `sigpending()` `sigprocmask()` `sigqueue()` `sigset()` `sigsuspend()`
`sleep()` `socket()` `socketpair()` `stat()` `symlink()` `sysconf()` `tcdrain()` `tcflow()`
`tcflush()` `tcgetattr()` `tcgetpgrp()` `tcsendbreak()` `tcsetattr()` `tcsetpgrp()` `time()`
`timer_getoverrun()` `timer_gettime()` `timer_settime()` `times()` `umask()`
`uname()` `unlink()` `utime()` `wait()` `waitpid()` `write()`.

Implementace (Linux)

základní datová struktura pro uložení odeslaných signálů je pole bitů typu **sigset_t**, jeden bit pro každý signál

```
typedef struct {  
    unsigned long sig[2];  
} sigset_t;
```

0 nemá žádný signál, v prvním prvku 31 tradičních signálů, ve druhém prvku signály pro reálný čas

deskriptor procesu obsahuje položky

signal typu **sigset_t** označující dodané signály

blocked typu **sigset_t** označující blokované signály

sigpending příznak, který je nastaven je-li jeden nebo více neblokovaných signálů nevyřízeno

gsig ukazatel na záznam **signal_struct** opisující obsluhu každého signálu

```
struct signal_struct {  
    atomic_t          count;  
    struct k_sigaction action[64];  
    spinlock_t        siglock;  
};
```

count počet procesů (a vláken) sdílejících
signal_struct - **clone()**, **fork()**,
vfork(), **CLONE_SIGHAND** příznak nastaven

siglock zajišťuje výhradný přístup k položkám
signal_struct

action[64] 64 **k_sigaction** záznamů specifikujících
obsahu jednotlivých signálů

sa_handler - **SIG_IGN**, **SIG_DFL**, nebo
ukazatel na obslužní funkci

sa_flags - příznaky pro obsluhu signálu

sa_mask - maskované signály při obsluze

Příklad1 – signal()

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void sigobsluha()
{
    int pid, stav;
    pid = wait(&stav);
    /*exit ulozi navratovy kod v bitech 8 az 15*/
    printf("skoncil potomek %d s navratovym kodem
           %d\n", pid, stav/256);
}

main()
{
    signal(SIGCLD, sigobsluha); /*standardne je
                                ignorovan*/

    if (fork() == 0)
    {
        printf("potomek pracuje\n");
        sleep(1);
        printf("potomek dopracoval\n");
        exit(1);
    };

    /*rodic neco dela*/
    printf("rodic pracuje\n");
    sleep(5);
    printf("rodic dopracoval\n");
    return(0);
}
```


Výstup:

```
potomek pracuje
rodic pracuje
potomek dopracoval
skoncil potomek 7734 s navratovym kodem 1
rodic dopracoval
```

Příklad2 – sigaction()

```
#include <signal.h>
#include <stddef.h>
#include <stdio.h>
#include <sys/wait.h>

void zastav() {
    printf ("Nechci zastavit!\n");
}

main()
{
    int i;
    struct sigaction akce;
    sigset_t blokujevse;

    /*blokuj signaly, nechceme byt preruseni*/
    sigfillset (&blokujevse);
    akce.sa_mask = blokujevse;
    akce.sa_handler = zastav;
    akce.sa_flags = 0;

    sigaction (SIGTSTP, &akce, NULL);

    for (i=0; i<10; i++) {
        printf("Spim %d\n", i);
        sleep(2);
    }
}
```

Výstup:

```
Spim 0
Spim 1
                                CTRL Z
Nechci zastavit!
Spim 3
                                CTRL Z
Nechci zastavit!
...
```

Příklad3 – sigaction(), siginfo

```
#include <stdio.h>
#include <signal.h>
#include <wait.h>
#include <ucontext.h>

void obsluha_potomka (int sig, siginfo_t *sip,
    void *notused)
{
    int stav;

    printf("Signal generoval proces: %d\n",
        sip->si_pid);
    fflush(stdout);

    /*WNOHANG není-li skončený potomek,
       waitpid() nečeká a vrátí 0*/
    if (sip->si_pid == waitpid(sip->si_pid,
        &stav, WNOHANG)){
        if(WIFEXITED(stav)){
            printf("Potomek skončil, návratový kód:
                %d.\n",WEXITSTATUS(stav));
        }
        else printf("Žádný potomek neskončil\n");
    }
}
```

```

main()
{
    struct sigaction akce;

    akce.sa_sigaction = obsluha_potomka;
    /*ne sa_handler*/
    sigfillset(&akce.sa_mask);
    akce.sa_flags = SA_SIGINFO;
    /*jinak NULL*/

    sigaction(SIGCHLD, &akce, NULL);

    if (fork()== 0) {
        printf ("Potomek PID: %d\n", getpid());
        sleep(1);
    }
    else {
        printf ("Rodic PID: %d\n", getpid());
        sleep(5);
    };
}

```

Výstup:

Potomek PID: 8502

Rodic PID: 8501

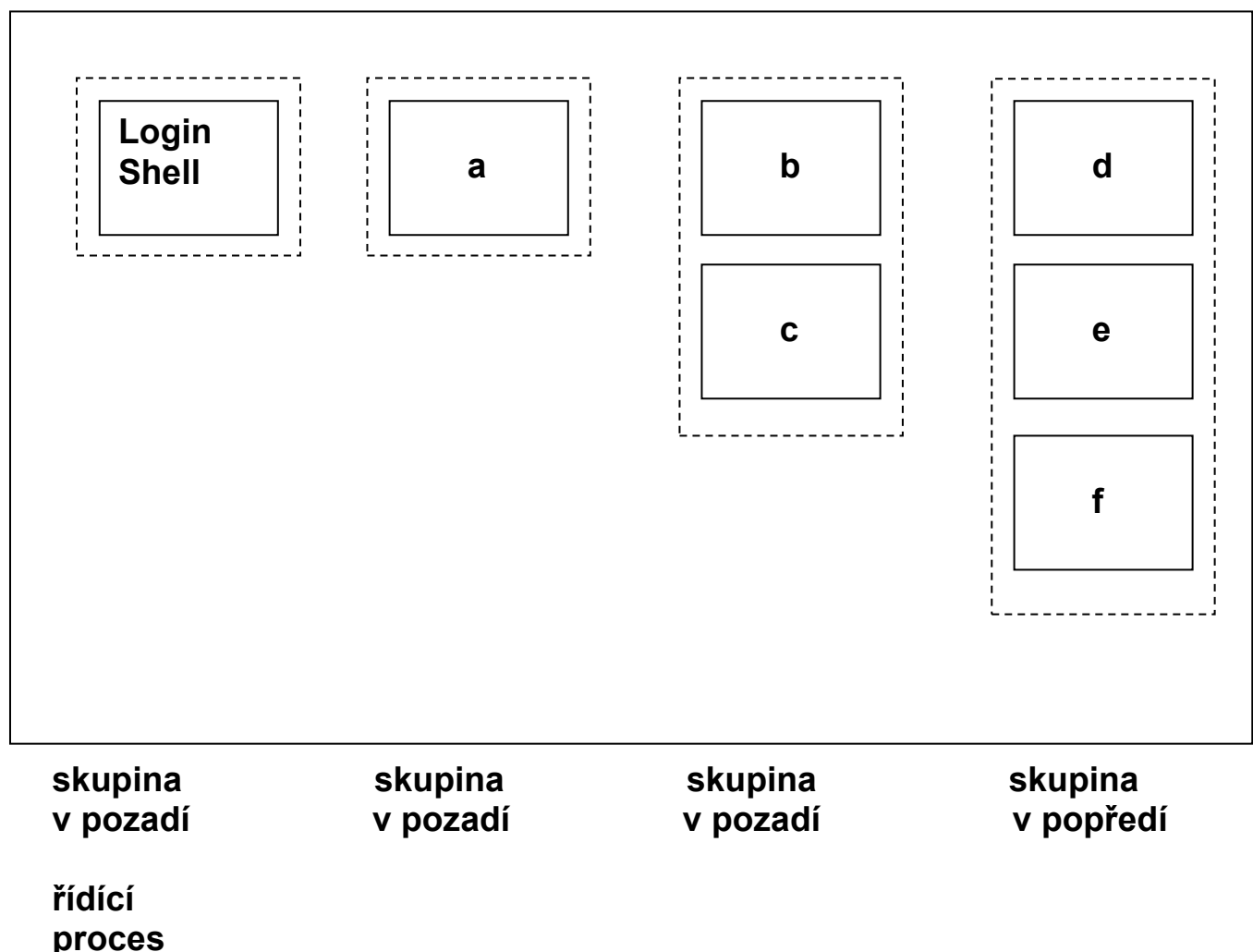
Potomek skoncil, navratovy kod: 0.

Session (práce, relace, sezení) a skupiny procesů

- umožňují vykonávat vícenásobné, souběžné úlohy (jobs) v jednom loginovém sezení, umístit je do pozadí, přenést do popředí, zastavit je a umožnit pokračování.

```
$ a &  
$ b | c &  
$ d | e | f  
$
```

session



- každý proces patří do skupiny procesů identifikované ukazatelem v deskriptoru procesu (**proc** záznamu)
- je vytvářen v průběhu **fork**
- procesy rodič, potomek, sourozenec jsou ve stejné skupině
- **PGID** je **PID** vedoucího procesu

po zahájení procesu ze shellu, je proces umístěn do vlastní skupiny voláním

```
int setpgid(pid, pgid) ;
```

procesy kolony budou v jedné skupině, vedoucím bude první vytvořený proces

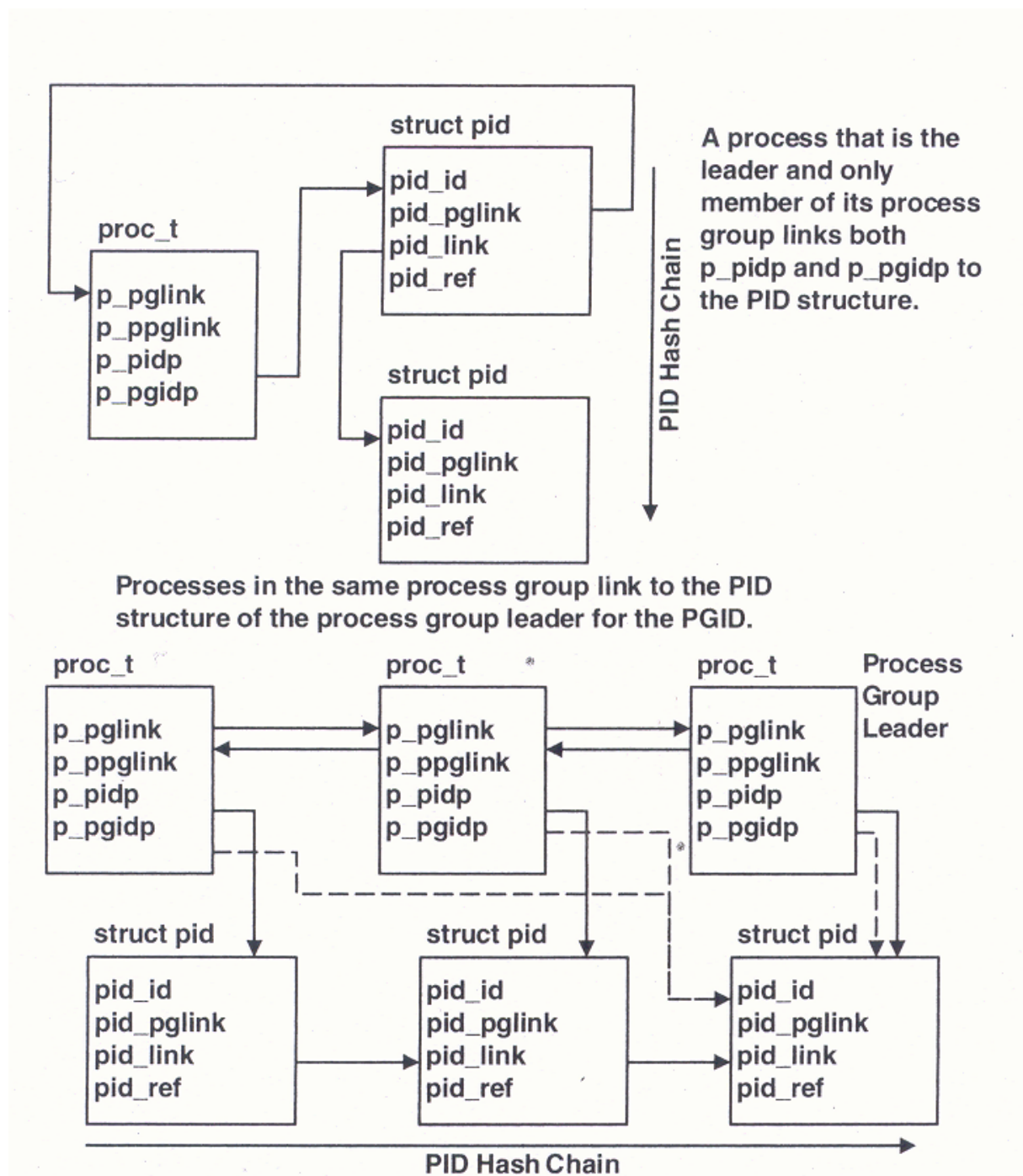
nejen shell, ale i aplikace mohou vytvářet skupiny procesů

signály možno zaslat všem procesům skupiny

skupina procesů může být v popředí nebo v pozadí

- procesy skupiny v popředí mají přístup k řídicímu (login) terminálu
- procesy skupiny v pozadí při čtení nebo zápisu na řídicí terminál obdrží signál SIGTTIN nebo SIGTTOU

procesy jedné skupiny jsou v obousměrném spojovém seznamu, p_pglink a p_ppglink

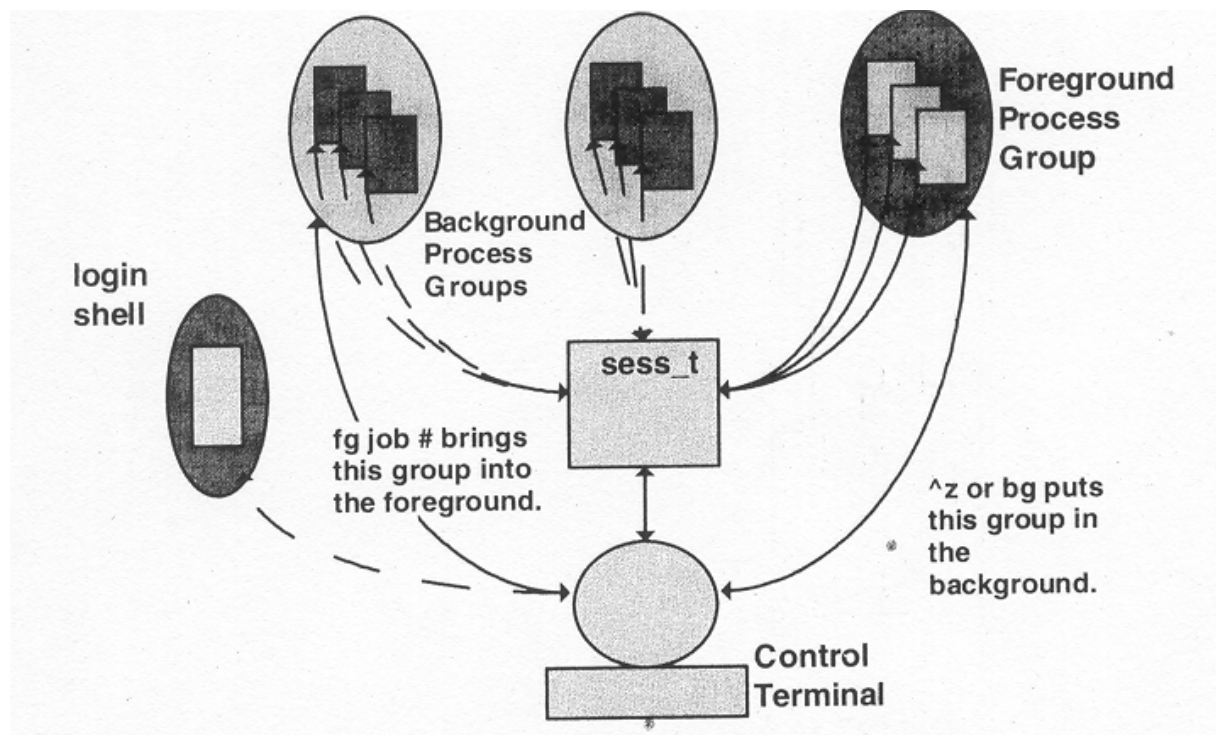


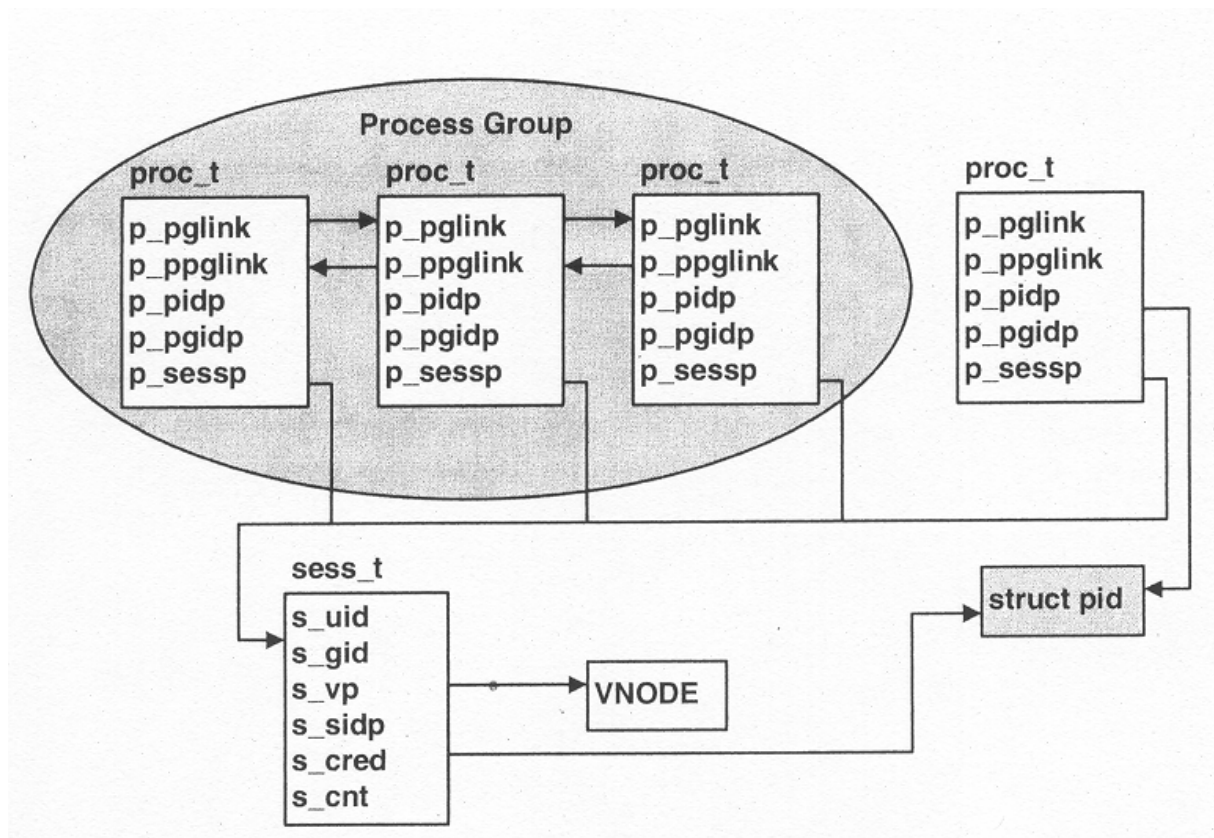
sezení obsahuje skupiny procesů se společným řídicím terminálem

- je reprezentováno datovou strukturou, na kterou ukazují procesy
- dědí se v průběhu **fork()**
- vedoucí sezení, proces který vytvořil spojení s řídicím terminálem, typicky login shell
- SID je PGID vedoucího skupiny

shell s řízením úloh

- po stisknutí CTRL Z vyše všem procesům skupiny v popředí signál SIGTSTP a procesy jsou zastaveny a je možno je umístit do pozadí - bg
- úlohu možno přenést do popředí příkazem fg, kdy vedoucí sezení voláním **tcsetpgrp()** jí přiřadí řídicí terminál
-





Zdroj: McDougalR., Mauro J.: Solaris Internals. Prentice Hall 2006