Signály.

Thursday, May 30, 2013 8:28 AM

- Jen u Linuxu, Windows mají "alternativu" - message
- Signály jsou přerušení generována softwarem, která jsou zaslána procesu, pokud nastane nějaká událost

  Unix systémy používají signály, aby daly **vědět procesu, že nastala určitá událost** (proces je pak např. vzbuzen, přerušen, …). Oznamují se jen čísla signálu, žádné parametry.

Signál vs. přerušení

Hezky podané na:

- http://stackoverflow.com/questions/13341870/signals-and-interrupts-a-comparison
- Na *přerušení* jde pohlížet jako na prostředek komunikace *mezi procesorem a jádrem OS*
- *Signály* mohou být brány jako prostředek komunikace *mezi jádrem OS a procesy*
  - Mohou být započaty jádrem OS (SIGSEGV, SIGIO) nebo procesem (*kill()*)
  - Jsou nakonec spravovány jádrem OS, které je doručí do cílového procesu/vlákna a spustí buď nějakou obecnou akci (ignorovat, ukončit, ukončit + dump core) a nebo obsluhu signálu, kterou poskytl proces

Synchronní a asynchronní signály

Signály mohou být buď **synchronní nebo asynchronní**, záleží na zdroji a důvodu, proč byla událost signalizována.

Mezi synchronní signály patří např. neoprávněný přístup do paměti a dělení nulou. Pokud běžící program provede některou z těchto akcí, vygeneruje se signál. Synchronní signály jsou doručeny stejnému procesu, který provedl operaci, která ten signál způsobila (což je důvod, proč jsou považovány za synchronní).

Když je signál generován událostí, která je externí vzhledem k běžícímu procesu, tak ten proces přijme ten signál asynchronně. Příklady takových signálů jsou ukončení procesu pomocí konkrétní klávesové kombinace (třeba <control><C>) a vypršení časovače. Asynchronní signál je typicky zaslaný jinému procesu.

Obsluha signálu

Jakmile byl signál generován výskytem nějaké události (např. dělením nulou, neoprávněným přístupem do paměti, uživatel stisknul CTRL+C = SIGINT signál), signál je dopraven procesu, kde musí být zpracován. Proces, který přijme signál, jej může zpracovat různými způsoby:

- Ignorování signálu (kromě SIGKILL a SIGSTOP signálů)
- Použití defaultního (výchozího) signal handleru (obsluhy signálu/funkce na zpracování signálu dle Košičana)
- Poskytnutí vlastní signal-handling funkce = vlastní obsluha signálu (kromě SIGKILL a SIGSTOP).

Každý signál má svou **výchozí obsluhu signálu (*default signal handler*),** která běží v jádře, když je signál zpracováván **(???**that is run by the kernel when handling that signal). Tato výchozí akce může být přepsána **uživatelem definovanou obsluhou signálu**, který je volán pro obsluhu daného signálu. Signály mohou být obslouženy různými způsoby. Některé signály (jako změna velikosti okna) mohou být jednoduše ignorovány; jiné (jako třeba neoprávněný přístup do paměti) mohou být obslouženy ukončením programu.

Signály mohou být obslouženy nastavením určitých proměnných v C struktuře `struct sigaction` a pak předáním této struktury `sigaction()` funkci. Signály jsou definovány v include souboru `/usr/include/sys/signal.h`. Např. signál SIGINT reprezentuje signál pro ukončení programu pomocí <Control> <C>. Výchozí obsluha signálu (signal handler) pro SIGINT je

ukončit program.

Další možností je, že v programu může být nastavena vlastní funkce pro obsluhu signálu, a to nastavením `sa_handler` proměnné ve struktuře `struct sigaction` na název funkce, která obslouží ten signál, a pak zavoláním funkce `sigaction()`. Té se předají jako parametry (1) signál, pro který nastavujeme obsluhu, a (2) pointer na `struct sigaction`.

Fáze signálů

Všechny signály, ať už synchronní nebo asynchronní, mají stejný životní cyklus:

1. Signál je vygenerován a **odeslán** poté, co nastane nějaká událost.
   - PM (*Process manager*) nejprve zjistí, které procesy mají obdržet signál
     - V tabulce procesů je pro každý proces několik `sigset_t` proměnných (=bitmapy, definují ignorované, zachycované signály)
   - Pro procesy zachycující daný signál:
     1) jádro zaznamená v záznamu proc (deskriptoru procesu) cílového procesu odeslání nového signálu.
     2) Jádro přeruší standardní provádění posloupnosti instrukcí cílového procesu, uloží informace o stavu procesu, aby se pak mohl opět pokračovat v běhu. Informace jsou uloženy na zásobníku toho procesu (kterému má dorazit signál) + kontrola, že je dost místa na zásobníku (dělá PM).
   - PM pak volá system task in jádře pro uložení informací do zásobníku. System task také manipuluje s program counterem procesu, aby proces mohl být spuště v kódu obsluhy.
2. Vygenerovaný signál je **doručen – přijat -** procesu.
3. Jakmile je signál doručen, musí být zpracován.
   - Když obsluha skončí, je provedeno systémové volání `sigreturn`. Prostřednictvím tohoto volání se PM i jádro podílejí na obnově kontextu signálu a registrů „signalizovaného“ procesu, aby mohl pokračovat v normálním běhu.
   - Pokud není signál zachycen (nemá def. handler), podnikne se defaultní akce, která se může týkat volání souborového systému pro vytvoření **core dumpu** (zápis obrazu paměti procesu do souboru, který může být prozkoumán debuggerem) či zabití procesu, pro něž je třeba zapojit PM, souborový systém a jádro.
   - PM řídí jednu nebo více opakování akcí výše - podle toho, jestli je signál doručen jednomu procesu nebo skupině procesů.

Fáze vypořádání se signály jinak:

1. Příprava (Preparation) – kód programu se připraví pro možný signál
   - Několik systémových volání, která lze nastavit jako odpověď na signál
   - `sigaction` → co má proces dělat se signálem: ignorovat/zachytit/nastavit defaultní reakci
   - `sigprocmask()` → blokování signálu; signál bude zařazen do fronty či se jím bude řídit až jej process později odblokuje
   - `sigsuspend(sigmask)` → nastaví se blokované signály podle `sigmask` a process přejde do stavu *čekající* až do zaslání signálu, který není blokován/ignorován. Dle *[Košičan, prednaska07, slide 13]* není to samý, co `sigprocmask()` a `sleep()`, ???nechápu
2. Odpověď (Response) – **signál je přijat** a příslušná akce je vykonána
3. Vyčištění (Cleanup) – obnova normální operace procesu
   - Viz bod 3 předchozího rozfázování

Příklady scénářů synchronního a asynchronního signálu

Synchronní:

- 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

Asynchronní:

- 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 naplánován jako běžící při návratu do uživatelského módu anebo byl-li běžící při návratu z přerušení, proces najde signál

Signály a vícevláknové procesy

Obsluha signálů v jednovláknových programech je přímočará; signály jsou vždy doručeny procesu. Doručení signálů je však komplikovanější u vícevláknových programů, kde proces může mít několik vláken. Kam se má potom signál doručit?

Obecně existují následující možnosti:

4. Doručit signál vláknu, ke kterému se signál vztahuje.
5. Doručit signál každému vláknu v procesu.
6. Doručit signál určitým vláknům v procesu.
7. Pověřit jedno konkrétní vlákno, aby přijímalo všechny signály pro proces.

Metoda pro doručení signálu záleží na typu generovaného signálu. Například synchronní signály je třeba doručit vláknu, které ten signál způsobilo a už ne ostatním vláknům procesu. U asynchronních signálů to ale není tak jasný. Některé asynchronní signály, třeba signál, který ukončuje proces (např. <control><C>), by měly být poslány všem vláknům.

Většina vícevláknových verzí UNIXu umožňuje vláknu určit, které signály bude přijímat a které blokovat. V některých případech proto může být asynchronní signál doručen pouze těm vláknům, která jej neblokují. Protože však signály musí být obslouženy pouze jednou, signál je obvykle doručen prvnímu nalezenému vláknu, které jej neblokuje.

Standardní UNIXová funkce pro doručení signálu je `kill (aid_t aid, int signal);` uvádíme zde proces (aid), kterému bude příslušný signál doručen. POSIX Pthreads taky ještě poskytují funkci `pthread_kill(pthread_t tid, int signal`, která umožňuje doručit signál konkrétnímu vláknu (tid.)

Windows APC

Ačkoliv Windows neposkytuje přímo podporu signálů, mohou být emulovány pomocí **asynchronních volání procedur** - ***asynchronous procedure calls*** **(APCs).** APC umožňuje uživatelskému vláknu (user thread) uvést funkci, která má být zavolána, když tomu user threadu přijde oznámení o určité události. Jak už název napovídá, APC je zhruba to samý co asynchronní signál v UNIXu. Zatímco se však UNIX musí potýkat s pořešením signálů ve vícevláknovém prostředí, možnost APC je přímočařejší, protože APC je doručeno konkrétnímu vláknu a ne procesu.

Příklady signálů

SIGINT – signál pro přerušení od terminálu (stisknutím Ctrl+C)

SIGQUIT – ukončí proces + core dump (záznam stavu pracovní paměti procesu do souboru, často při abnormálním ukončení)

SIGKILL – zabije proces, nemůže být zachycen ani ignorován + proces nemůže po jeho přijetí provést úklid

Proč může dojít ke zpoždění vyřízení signálu?

- 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

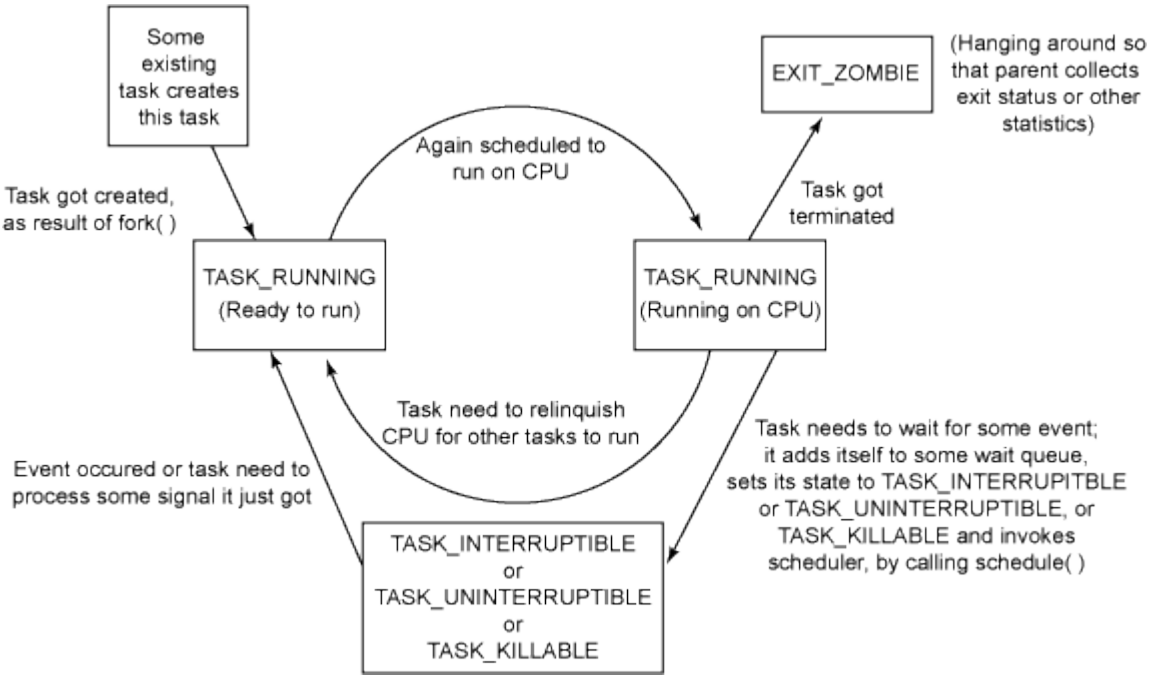
Další prodlení může způsobit, je-li proces v čase odeslání signálu ve stavu **zastaven** nebo **spící**

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***) a úloha_nepřerušitelná (***TASK_UNINTERRUPTIBLE***)



Spolehlivé a nespolehlivé signály

Nespolehlivé = můžou se ztratit; proces někdy zachytí signál, jindy ho ztratí; kvůli resetu signálu na implicitní akci.

Neumožňují ignorovat signál v daný moment, ale pamatovat si, že nastal, aby jej šlo blokovat a zpracovat později (třeba po skončení důležitého výpočtu).

Pokud to chceme emulovat (nastavit nějakou akci pro následující signál), musíme ji opět instalovat (= volat funkci `signal(sig, function)`) - vznik nedeterminismu, někdy se to stihne, někdy ne

Spolehlivé = perzistentní obslužné funkce signálů; blokování signálu, např. při obsluze signálů → nevznikne hnízdění

Další zdroje

- Pěkně vysvětleno na http://www.cs.uregina.ca/Links/class-info/330/Signals/signals.html

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Printout

Sunday, June 9, 2013 1:02 AM

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 hlavním 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 zasílaní 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

Page 2

Sunday, June 9, 2013 1:02 AM

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 sytémového 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

Page 3

Sunday, June 9, 2013 1:02 AM

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 zaslám 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** obslouž signál v alternativním zásobníku deklarovaném voláním sigalstack()
  - **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 přerušený 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 blokovaných signálů

```
sigprocmask(how, set, oset);
```

oset stará maska signálů

set nová maska signálů

how

- **SIG_BLOCK** nová maska specifikuje signály, které se přidají k blokovaným
- **SIG_UNBLOCK** nová maska specifikuje signály, kterých blokování se odstraní
- **SIG_SETMASK** nová maska specifikuje blokované signály

Page 4

Sunday, June 9, 2013 1:02 AM

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 prvím 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 obsluhu 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);
}
```

Page 5

Sunday, June 9, 2013 1:02 AM

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 blokujvse;

  /*blokuj signaly, nechceme byt preruseni*/
  sigfillset (&blokujvse);
  akce.sa_mask = blokujvse;
  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 neni-li skonceny potomek,
    waitpid() neceka a vrati 0*/
  if (sip->si_pid == waitpid(sip->si_pid,
       &stav, WNOHANG)){
    if(WIFEXITED(stav)){
       printf("Potomek skoncil, navratovy kod:
                %d.\n",WEXITSTATUS(stav));
    }
  else printf("Zadny potomek neskoncil\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);
  };
}
```

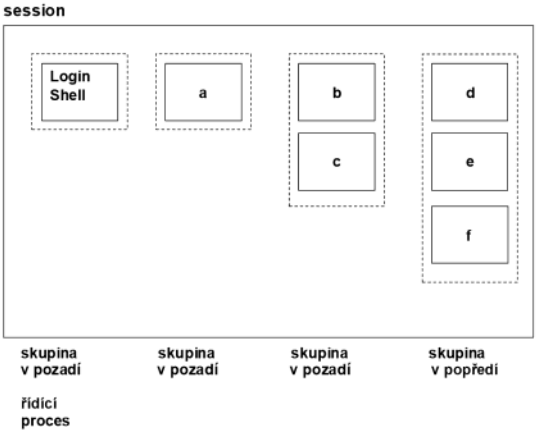
Výstup:

```
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, umístnit je do pozadí, přenést do popředí, zastavit je a umožnit pokračování.

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



Page 6

Sunday, June 9, 2013 1:02 AM

- 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 umístněn do vlastní skupiny voláním

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

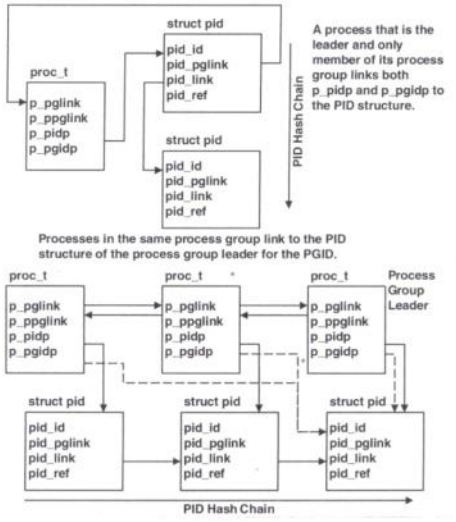
procesy kolony budou v jedné skupině, vedoucím bude první vytvořený proces

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 řídícímu (login) terminálu
- procesy skupiny v pozadí při čtení nebo zápisu na řídící terminál obdrží signál SIGTTIN nebo SIGTTOU

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

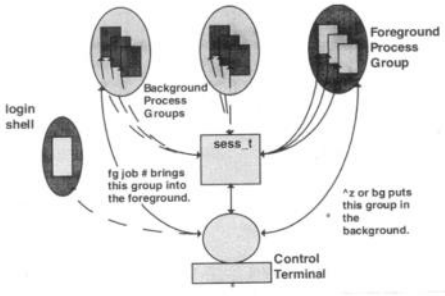


Zdroj: McDougalR., Mauro J.: Solaris Internals. Prentice Hall 2006

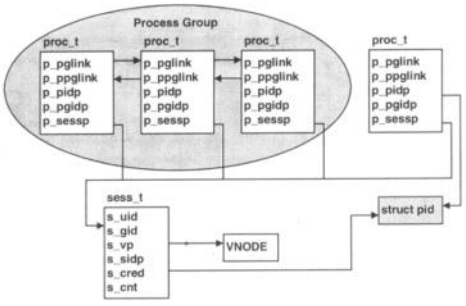
sezení obsahuje skupiny procesů se společným řídícím terminálem

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

shell s řízením úloh
- po stisknutí CTRL Z vyšle všem procesům skupiny v popředí signál SIGTSTP a procesy jsou zastaveny a je možno je umístnit do pozadí - bg
- úlohu možno přenést do popředí příkazem fg, kdy vedoucí sezení voláním **`tcsetpgrp()`** jí přiřadí řídící terminál
-



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



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