

Systemové programovanie / ÚINF/SPR₁



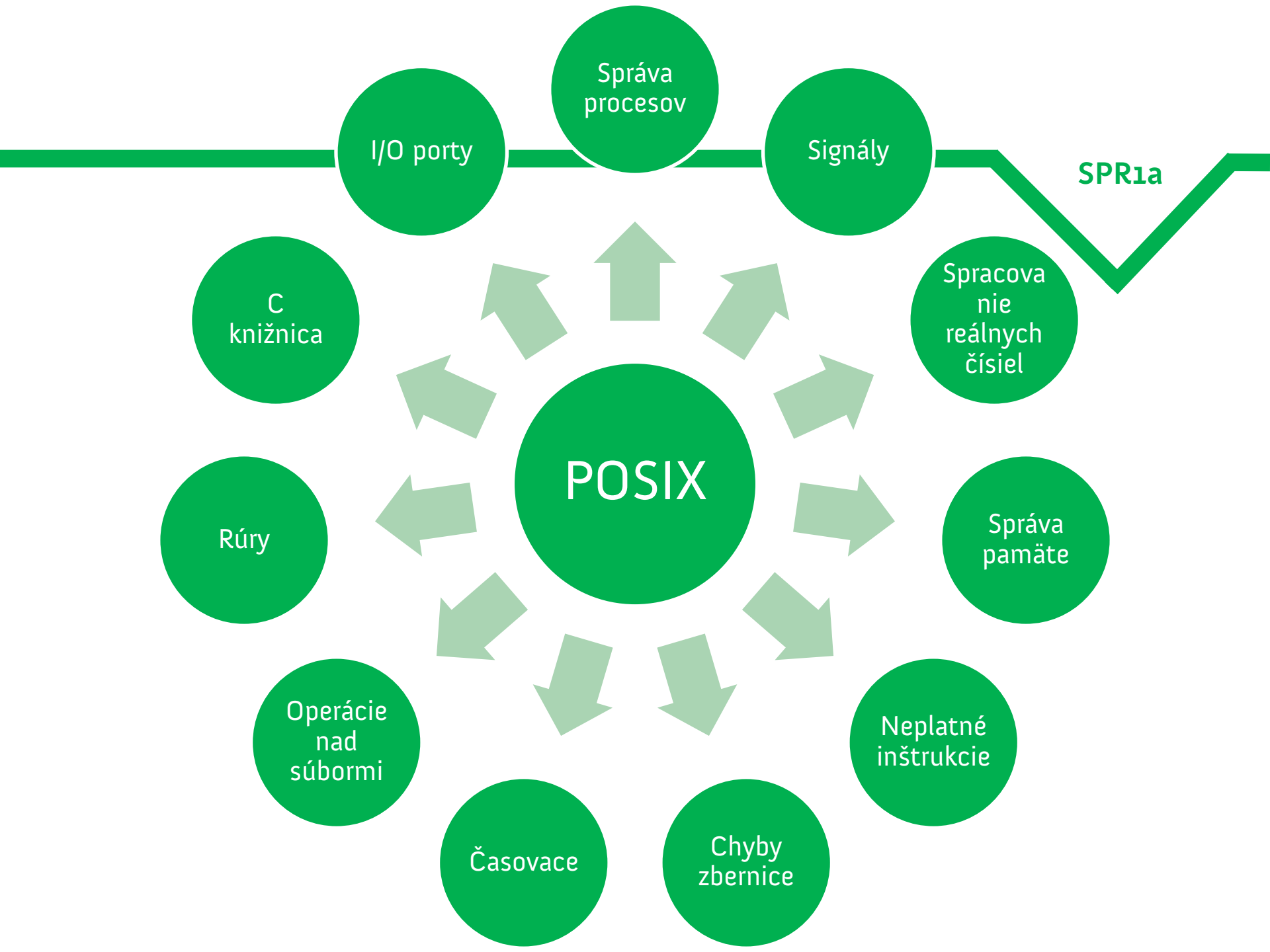
SPR_{1a}

16. 2. 2011

POSIX: Portable Operating System Interface

SPR1a

- Sada štandardov pre **kompatibilitu** medzi operačnými systémami odvodená od Unixu
- Podpora prenositeľnosti softvéru
- Historicky: mnoho vetiev unixovských systémov,
 - rozličné systémové volania,
 - rozličné návratové hodnoty
 - t. j. veľa `#ifdef` pri kompilácii
- Potreba **zjednotiť** systémy



A thick green line that starts horizontally from the left edge of the slide, then descends diagonally to the right, then ascends diagonally to the right, and finally continues horizontally to the right edge of the slide. The label 'SPR1a' is positioned near the first downward slope.

SPR1a

PROCESY A SIGNÁLY

Procesy

SPR1a

- **program**: postupnosť inštrukcií pre počítač
 - obvykle pre CPU, ale často aj pre iné komponenty
- **proces**: bežiaci inštancia programu
 - notepad.exe si môžeme spustiť viackrát = viac procesov
 - štandard na viacúlohových operačných systémoch

Identifikátory procesov

SPR1a

- každý proces v OS má svoje ID = **pid**
 - Linux: 16-bitové čísla pridelované sekvenčne
 - Windows: analogicky
- rodičovské ID: **ppid**
 - procesy v Linuxe sú v stromovej hierarchii
 - vo Windowse sa ppid eviduje, ale hierarchia nie je udržiavaná v strome

Identifikátory procesov

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

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

```
int main () {
```

```
    printf ("PID: %d\n", (int) getpid());
```

```
    printf ("PPID: %d\n", (int) getppid());
```

```
    return 0;
```

```
}
```

štandardné
POSIX volania

systemové
volania

Zobrazenie aktívnych procesov

SPR1a

```
ps
```

```
ps -e -o pid, ppid, command
```

-e: všetky procesy
-o zoznam vlastností

```
tasklist
```


Štart linuxového systému

SPR1a

- **bootloader** načíta do pamäte
 1. jadro
 2. image iniciálneho koreňového súborového systému
- spustí **jadro** a odovzdá mu **adresu** tohto image v pamäti
- jadro zdetekuje formát imageu (initrd / initramfs)
- namontuje ho ako dočasný súborový systém
- vykoná **prvý skript** /init
 - ten je prvým spusteným procesom v systéme
 - nepredpokladá sa, že dobehne

Init-
ram-
fs

Formát imageu: **initramfs**

SPR1a

- Pre efektivitu 2x **komprimovaný**
 - cpio + gz
- obsahuje **ovládače** a podobne, ktoré sa dynamicky načítajú pri boote
 - ovládače nemusia byť zakompilované v jadre
 - jadro má minimálnu veľkosť
 - jadro bootne na všetkom
 - všetky voliteľné ovládače dotiahne z imageu v závislosti od hardvéru

Pokračovanie bootu

SPR1a

- prvý skript je program prvého procesu v systéme
- **PID = 1**
- inicializácia ďalších procesov závisí od distra
- predovšetkým sa spúšťajú **démoni**
 - programy, ktoré bežia na pozadí
 - a nepotrebujú interakciu s používateľom
 - analógia rezidentných programov v MS-DOS

Debian: štýl SystemV

SPR1a

- vytvorí sa sada skriptov pre spúšťanie jednotlivých procesov
 - shellskripty v **dash**-i
- tie sa sekvenčne spúšťajú
- prítomné v **/etc/rc.2/@XYYYZZZZZZZZZZ**
 - X –skript pre **š**start (S) alebo **s**končenie démona
 - YYY – číselná priorita spúšťania
 - ZZZZZZ - názov démona
 - @ – runlevel: číslo módu operácie stroja

Runlevels v SystemV inicializácii

SPR1a

- zodpovedá stavu konkrétneho stroja po bootnutí
- rozlične definované v závislosti od distra
 - 1: single-user mode: bez siete, bez spúšťania démonov, nahlásený ako root
 - 3: multiuser mode: so sieťou, s používateľmi
 - 5: bežný štart vrátane GUI
 - 6: reboot
- runlevely **nie sú** vykonávané sekvenčne!
 - pri boote sa vykoná len jediný runlevel
 - možno sa však medzi nimi prepínať

Ubuntu: upstart

SPR1a

- SystemV má nevýhody
 - je len sekvenčný
 - dnes máme viacjadrové CPU, môžeme paralelizovať boot
 - ťažkopádna syntax s podivnosťami a malou flexibilitou
- definujú sa pravidlá pre závislosti podobne ako v **make**
 - ak nie sú žiadne závislosti, všetko sa spustí naraz
 - možno definovať rozličné udalosti a reagovať na ne
 - zaručená kompatibilita so SystemV

[Linux] Vytváranie procesov: system

SPR1a

- možnosť použiť systémové volanie **system**
- vyrobí sa podproces, v ktorom sa spustí shell
- v tomto shelli sa spustí zadaný príkaz

```
#include <stdlib.h>
int main () {
    int return_value;
    return_value = system("ls -l /");
    return return_value;
}
```

- implementačne závislé
- neefektívne
- pozor na bezpečnosť

[Linux] Vytváranie procesov: fork/exec

SPR1a

- **fork():** systémové volanie, ktoré vyrobí kópiu rodičovského procesu
- po zavolaní sa vytvorí potomkovský proces
 - má PID=0
- rodičovský i potomkovský proces potom pokračujú v ďalšom vykonávaní programu
- fork() vracia PID rodičovského procesu

Windows: jediné systémové volanie **spawn**

[Linux] Vytváranie procesov: fork/exec

SPR1a

```
pid_t child_pid;
printf ("Rodič. PID: %d\n", (int) getpid ());
child_pid = fork();
if (child_pid != 0) {
    printf("Sme u rodiča:pid=%d\n", (int) getpid ());
    printf ("PID pre potomka: %d\n", (int) child_pid);
} else {
    printf ("Sme u potomka, pid=%d\n", (int) getpid ());
}
return 0;
```

main()

[Linux] Vytváranie procesov: fork/exec

SPR1a

- v podmienke sa prvá vetva vykoná len pre rodičovský proces
- druhá len pre potomkovský
- dátové typy:
 - `stdio.h`
 - `sys/types.h`: **pid_t**: typedef pre pid
 - `unistd.h`

[Linux] Vytváranie procesov: fork/exec

SPR1a

- **exec**: nahradí program bežiaci v procese iným programom
 - po zavolaní `execu` sa aktuálny program prestane vykonávať
 - začne sa vykonávať nový program
- **exec** nikdy nevracia hodnotu
 - predošlý program je totiž ukoneční

[Linux] Rodina funkcií exec()

SPR1a

Znak	Význam	Ostatné funkcie
p	Berú názov programu, ktorý sa vyhľadá v ceste	Ostatné funkcie berú úplnú cestu k súboru
v	Berú argumenty pre program v podobe NULL-terminated poľa reťazcov	
l	Berú hodnoty argumentov z parametrov funkcie	
e	Berú pole premenných prostredia	Reťazec je v tvare PREMENNA=hodnota

execvp, execlp, execv, execve,execle

Použitie fork/exec

SPR1a

- forkneme proces
- execneme podprogram
 - volajúci program pokračuje vo vykonávaní
 - potomkovský program sa nahradí podprogramom

Použitie fork/exec

SPR1a

```
int spawn(char * program, char ** argumenty) {
    pid_t pid_dietata;
    pid_dietata = fork();
    if(pid_dietata != 0) {
        return child_pid; /* sme v rodicovi */
    } else {
        execvp(program, argumenty);
        /* execvp() vracia len vtedy, ak nastala chyba */
        printf("Pri spustani dietata nastala chyba.");
    }
}
```

program: názov programu hľadaný v ceste
argumenty: NULL-terminated pole reťazcov

Použitie fork/exec

SPR1a

```
int main () {  
    char * parametre [] = { "ls", "-l", "/", NULL };  
  
    spawn("ls", parametre);  
  
    printf("Hlavný program dobeho1\n");  
    return 0;  
}
```

spúšťame ls
funkcia na predošlom
slajde

Procesy vznikajú delením

SPR1a

- fork/exec je tradičný spôsob spúšťania procesov
- „procesy vznikajú delením“
- vzniká tak **stromová hierarchia** procesov
- PIDy sú pridelované **inkrementálne**
 - do 32768
 - medzery sa vyplňajú až po **pretečení** SIGINT

Plánovanie (scheduling) procesov

SPR1a

- plánovanie procesov je v Linuxe nedeterministické
- rodičovské a potomkovské procesy bežia nezávisle
 - žiadna garancia o poradí behu!
 - spomnime race conditions z vlákien
- možnosť meniť prioritu: príkazy **nice** a **renice**
 - vyšší niceness: menšia priorita
 - default: 0

Multitasking

SPR1a

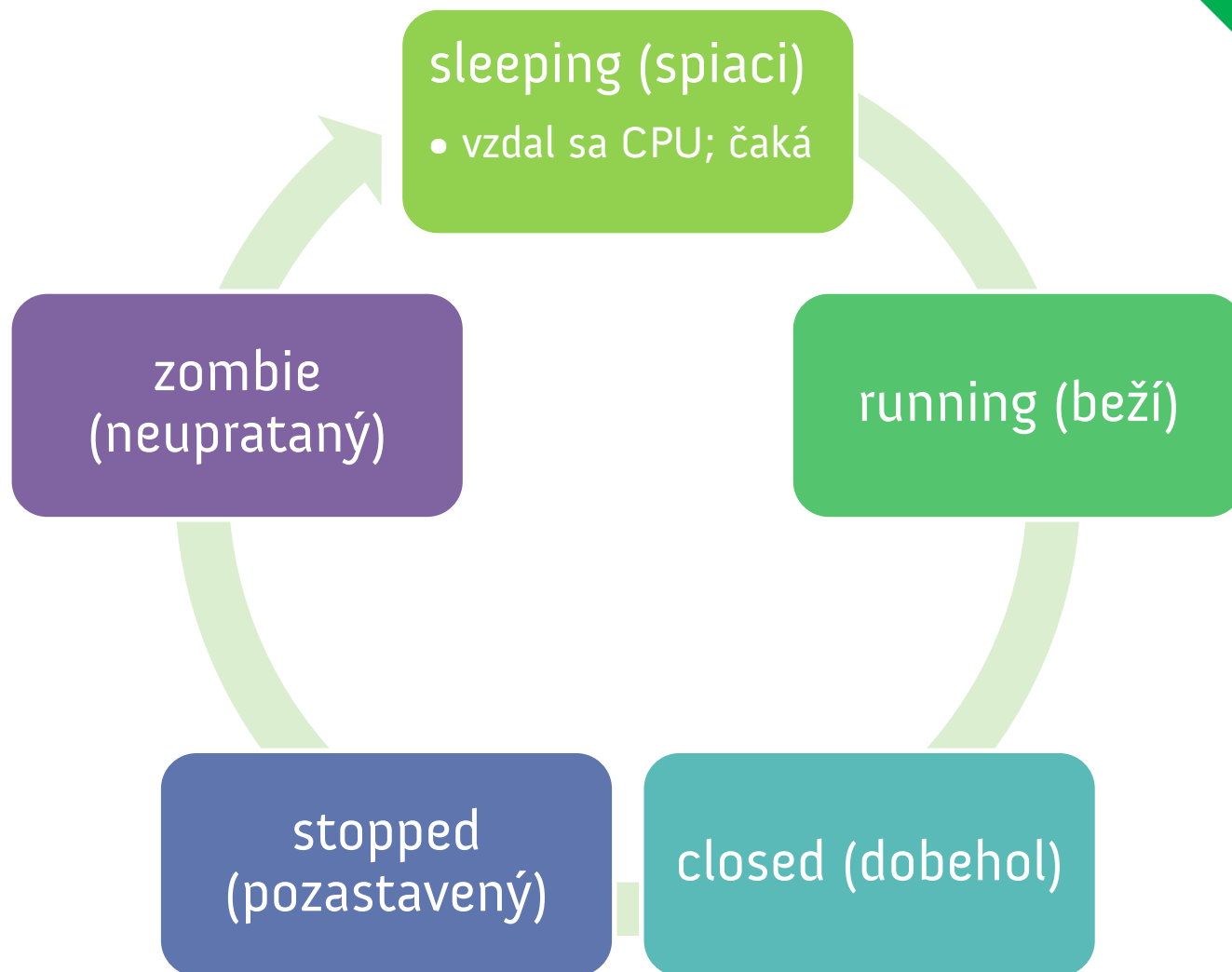
- **Kooperatívny** multitasking (MS-DOS)
 - aplikácia sa sama musí zriecť procesorového času
 - procesu nemožno odobrať procesorový čas
- **Preemptívny** multitasking
 - riadenie na úrovni jadra
 - jadro obsahuje plánovač (scheduler)
 - rozličné spôsoby:
 - **time-slicing**: dokola sa prideluje procesorový čas
 - systém **prioritných front**

System prioritných front

SPR1a

- dobré procesy sú odmeňované, zlé trestané
- procesy, ktoré žerú veľa CPU, budú mať zníženú prioritu
- ppačne: proces, ktorý čaká na vstup: čím dlhšie spí, tým lepšiu prioritu dostane
 - a nepoužíva `while-sleep()`, ale je kultúrny

Životný cyklus procesu



SPR1a

Životný cyklus procesu

SPR1a

- **Sleeping:**
 - proces sa aktívne zastaví a čaká
 - napr. na vstup
- **Running:** bežiaci
 - hneď ako dostane procesorový čas, rozbehne sa
 - v tomto stave môže byť viacero procesov
- **Stopped:** pozastavený
 - Zvonku, napr. z gdb alebo signálom SIGQUIT
 - Ctrl-Z
 - Čaká na pokračovanie
- **Zombie**
 - dobehnutý, ale neuprataný potomok po fork()
 - o tom neskôr

Životný cyklus: zombie

SPR1a

- dobehnutý, ale neupratali sa všetky zdroje
- po forku() sa alokujú zdroje pre dcérsky proces
- Ak dcéra dobehne, zdroje ostávajú alokované
 - Ten treba upratať
- **Na starých systémoch:** veľa alokovaných procesov mohlo zabrániť súvislej alokácii pamäte
- Plus stále existuje limit na počet procesov (maximálne limit na PID)

Signály

SPR1a

- mechanizmus pre komunikovanie a manipulovanie s procesmi (IPC)
 - IPC: inter-process communication
 - historicky najstarší v UNIXe
- **signál** = správa odoslaná procesu
- proces spracuje signál ihneď
 - asynchrónne spracovanie
 - beh programu sa pozastaví, vykoná sa obsluha signálu, beh programu pokračuje
 - "softvérové prerušenie"

signal.h

Signály

SPR1a

- v POSIX norme 28 štandardných signálov
- procesu možno poselať signál cez **kill**

```
kill -s SIGTERM pid  
kill -s 9 pid
```

- buď číslo signálu alebo skratka
- ak číslo signálu neuvedieme, posiela sa **TERM**

Ako reagovať na signál?

SPR1a

- pre každý signál je definovaná štandardná obsluha (**default disposition**)
 - EXIT: násilne ukončiť proces
 - CORE: násilne ukončiť proces + core dump
 - STOP: ukončiť proces
 - IGNORE: ignorovanie signálu
- program môže definovať vlastnú obsluhu signálu (**signal handler**)

Starý spôsob reakcie na signály

SPR1a

- systémové volanie **signal()**
 - <http://pubs.opengroup.org/onlinepubs/09695399/functions/signal.html>
- dva parametre
 - číslo signálu
 - pointer na funkciu, ktorá obslúži signál
- **zastaralé**, neodporúča sa používať

Signály

"vláknovo" bezpečný int
s atomickým zápisom

```
sig_atomic_t sigusr1_count = 0;
```

```
void handler(int signal_number) {  
    ++sigusr1_count;  
}
```

```
int main() {  
    struct sigaction sa;  
    memset(&sa, 0, sizeof (sa));  
    sa.sa_handler = &handler;  
    sigaction(SIGUSR1, &sa, NULL);  
    /* vykonaj dlhodobé operácie */  
    printf("SIGUSR1 nastal %d-krát\n", sigusr1_count);  
    return 0;  
}
```

signal handler

Signály

```
sig_atomic_t sigusr1_count = 0;
void handler(int signal_number) {
    ++sigusr1_count;
}
int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof (sa));
    sa.sa_handler = &handler;
    sigaction(SIGUSR1, &sa, NULL);
    /* vykonaj dlhodobé operácie */
    printf("SIGUSR1 nastal %d-krát\n", sigusr1_count);
    return 0;
}
```

nastavenie obsluhy signálu

pointer na funkciu obsluhy

asociovanie obsluhy so signálom

Signály

SPR1a

- **sa_handler** v structe sigaction:
 - SIG_DFL: zavolať štandardnú signal disposition
 - SIG_IGN: ignorovať príslušný signál
 - pointer na funkciu s obsluhou
 - funkcia berie int, vracia void
- takto možno ignorovať napríklad SIGTERM
 - ale už nie SIGKILL ani SIGSTOP

```
kill -9 pid
```

Signály

SPR1a

- obsluha signálov je **asynchrónna**
- počas nej môže byť program v prapodivnom stave
- odporúčania:
 - vyhnúť sa I/O operáciám v obsluhu signálov
 - vyhnúť sa systémovým volaniam / knižniciam
 - obsluha by mala byť čo najrýchlejšia

Čo ak pri obsluhu signálu nastane iný signál?

SPR1a

- Do funkcie môže vbehnúť viacero obslúh naraz
- Funkcia obsluhy signálu musí byť reentrantná
 - Existuje zoznam funkcií, ktoré možno bezpečne volať
 - <http://book.chinaunix.net/special/ebook/addisonWesley/APUE2/0201433079/ch10lev1sec6.html>

Signály



SPR1a

- typická obsluha:
 - poznamenať si, že nastal signál
 - mimo obsluhy periodicky kontrolovať, či signál nastal a obslúžiť ho
- bonusovka:
 - obsluha signálu môže byť prerušená iným signálom
 - vyplývajú z toho veselice známe z vláknového programovania

Ukončovanie procesov, part II

SPR1a

- proces môže skončiť viacerými spôsobmi
- dobehne `main()`
 - návratovú hodnotu získa rodičovský proces
- program zavolá `exit()`
 - parametrom je návratová hodnota
- ako dôsledok obsluhy signálu
 - napr. `SIGINT` (Ctrl-C v termináli)

Ukončovanie procesov, part II

SPR1a

- niektoré signály vedú k ukončeniu
 - SIGTERM: posielaný killom
 - SIGINT: Ctrl-C
 - SIGABRT: volanie funkcie abort() + core dump
 - SIGKILL: neobslúžiteľné ukončenie

```
kill -KILL pid
```

```
kill(pid, signál)
```

```
sys/types.h  
signal.h
```

Návratová hodnota: exit code

SPR1a

- **silná konvencia**: návratová hodnota indikuje úspešnosť spustenia
 - **nula** = program dobehol **správne**
- množstvo vlastností shellu sa spolieha na toto správanie
 - napr. booleovské funkcie pri spúšťaní viacerých programov
- **main()** preto nech vždy vracia konkrétnu hodnotu
 - inak je návratová hodnota nedefinovaná

Návratová hodnota: exit code

SPR1a

- v mnohých shelloch možno zistiť exit code posledného programu

```
echo $?
```

- v Linuxe: do úvahy sa berie len exit code medzi 0 a 127
 - ak je proces ukončený signálom, exit code = 128 + číslo signálu

Čakanie na ukončenie procesu

SPR1a

- procesy sú plánované nedeterministicky
- to platí aj pre rodičovský proces a potomkovský proces

Ako môže rodičovský proces čakať na dobehnutie potomka?

- systémové volania **wait()**

Systemové volanie **wait()**

SPR1a

- najjednoduchší prípad: **wait()**
- blokuje vykonávanie procesu, kým nedobehne aspoň jeden potomok
 - potomok môže dobehnúť úspešne alebo s chybu
- návratovou hodnotou je stavový kód.
- makrami možno zistiť **spôsob ukončenia**
 - WEXITSTATUS: exit code
 - WIFEXITED: skončil proces kultúrnym spôsobom?
 - WTERMSIG: číslo signálu, ktorým proces skončil

Systemové volanie `wait()`

SPR1a

```
int child_status;  
char* arg_list[] = { "ls", "-l", "/", NULL };  
/* spustíme potomka, jeho pid ignorujeme */  
spawn("ls", arg_list);
```

```
wait (&child_status);
```

čakáme na dobehnutie
potomka

```
if (WIFEXITED(child_status))  
    printf ("Potomok dobehol normálne, kód: %d\n",  
           WEXITSTATUS(child_status));  
else  
    printf("Potomok dobehol abnormálne.\n");  
return 0;
```

Ďalšie volania z rodiny wait()

SPR1a

- **waitpid()**

- čakanie na dobehnutie konkrétneho potomka

- **wait3()**

- možnosť získať štatistiky o využití CPU

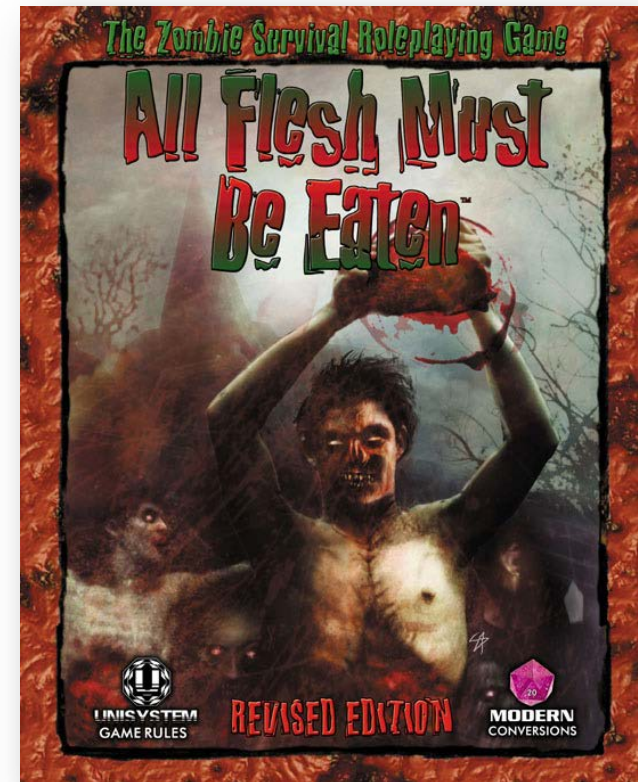
- **wait4()**

- možnosť získať bonusové informácie
- `getrusage()`: využitie systémových zdrojov

Zombie procesy

SPR1a

- ak rodič čaká vo wait() a potomok dobehne, stav dobehnutia získa v návratovej hodnote
- čo ak potomok dobehne a rodič nečaká?
- z potomka sa stáva **zombie**



Zombie procesy

SPR1a

- zombie proces dobehol, ale **nebol upratany**
 - možno z neho získať stavové informácie
- rodič je povinný **upratať** svoje zombie deti
 - systémové volania wait() to robia automaticky
- inak **problémy**:
 - ak máme málo RAM: veľa alokovaných procesov môže zabrániť súvislej alokácii pamäte
 - existuje limit na počet procesov (maximálne limit na PID)

Ako vytvárať zombie

SPR1a

```
pid_t child_pid;
/* Vytvoríme potomka */
child_pid = fork ();
if (child_pid > 0) {
    /* Sme u rodiča. Uspíme sa na minútu. */
    sleep (60);
} else {
    /* Sme u potomka. Okamžite sa ukončíme. */
    exit (0);
}
return 0;
```

- toto je zlý príklad! dieťa je zombie!
- našťastie zombie umierajú s rodičom

Ako zakopávať zombie?



- jednoduché riešenie: `exec + wait`
 - čakáme, kým dieťa nedobehne
- čo ak chceme deti spúšťať paralelne?
- **`wait3()/wait4()`**
 - budeme ich volať periodicky
 - `wait()` je tu nepoužiteľný kvôli blokovaniu
 - pomocou parametra `WNOHANG` budú bežať v neblokujúcom režime
 - ak sa nájde zombie proces, upracú ho, inak ihneď vrátia hodnotu

Ako zakopávať zombie?



- elegantnejšie riešenie
- pri ukončení potomka je rodičovi odoslaný signál **SIGCHLD**
 - default disposition: nič
- signál odchytíme v handleri
- zavoláme wait()
 - týmto sa odprace zombie dieťa a vráti jeho stav dobehnutia
 - stav dobehnutia si poznačíme, ak treba
 - po dobehnutí wait()u je zombie uprataný a stav nedostupný

Ako vytvárať zombie

SPR1a

```
sig_atomic_t child_exit_status;  
void clean_up_child_process (int signal_number) {  
    int status;  
    wait (&status);  
    child_exit_status = status;  
}  
int main () {  
    struct sigaction sa;  
    memset (&sa, 0, sizeof (sigchld_action));  
    sa.sa_handler = &clean_up_child_process;  
    sigaction(SIGCHLD, &sa, NULL);  
    /* pracujme ďalej, najmä sa forknime do potomka */  
}
```

získame stav ukončenia