Systémové programovanie / ÚINF/SPR1b

SPR1a

Róbert Novotný
robert.novotny@upjs.sk

15. 2. 2011

Formality a byrokracie

- Prednáška
  - utorok, 8.55, P10
- Praktické cvičenie
  - streda, 15:00, P7

Procesy

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

Identifikátory procesov

- každý proces v OS má svoje ID = **pid**
  - Linux: 16-bitové čísla prideľované 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

systémové volania

Zobrazenie aktívnych procesov


```
ps
```

```
ps –e –o pid, ppid, command
```

-e: všetky procesy
-o zoznam vlastností

```
tasklist
```

[Linux] Vytváranie procesov: system

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

```
#include <stlib.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

- **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 kódu
- fork() vracia PID rodičovského procesu

Windows: jediné systémové volanie **spawn**

[Linux] Vytváranie procesov: fork/exec


```
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

- 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

- **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()

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

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

Použitie fork/exec


```
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


```
int main () {
  char * parametre [] = { “ls”, “-l”, “/”, NULL };

  spawn(“ls”, parametre);

  printf(“Hlavný program dobehol\n”);
  return 0;
}
```

spúšťame **ls**
funkcia na predošlom slajde

Plánovanie (scheduling) procesov

- 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

[Linux] Signály

- mechanizmus pre komunikovanie a manipulovanie s procesmi
- **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

[Linux] Signály

- 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**)

[Linux] 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

[Linux] 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

[Linux] Signály

- **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

kill -9 *pid*

[Linux] Signály

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

[Linux] Signály

- 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

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

Ukončovanie procesov, part II

- 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

- **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

- 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

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

Ako môže rodičovský proces čakať na dobehnutie potomka?

- systémové volania **wait()**

Systémové volanie wait()

- 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

Systémové volanie wait()


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

wait (&child_status);

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

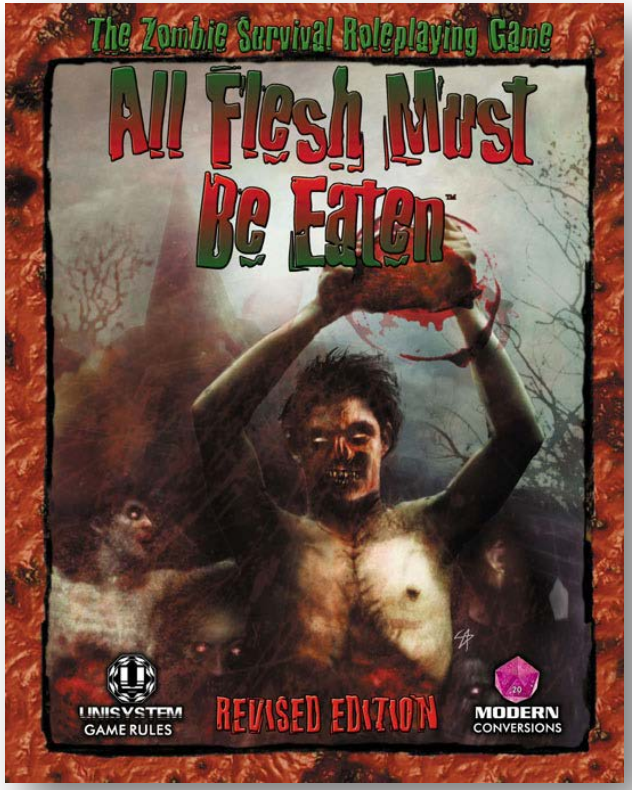
čakáme na dobehnutie potomka

Ďalšie volania z rodiny wait()

- **waitpid()**
  - čakanie na dobehnutie konkrétneho potomka
- **wait3()**
  - možnosť získať štatistiky o využití CPU
- **wait4()**
  - možnosť získať bonusové informácie
  - getrusage(): využitie systémových zdrojov

Zombie procesy

- 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

- zombie proces dobehol, ale nebol upratený
  - možno z neho získať stavové informácie
- rodič je povinný upratať svoje zombie deti
  - systémové volania wait() to robia automaticky

Ako vytvárať zombie


```
pid_t child_pid;
/* Vytvorme potomka */
child_pid = fork ();
if (child_pid > 0) {
  /* Sme u rodiča. Uspime sa na minútu. */
  sleep (60);
} else {
  /* Sme u potomka. Okamžite sa ukončime. */
  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 upratený a stav nedostupný

Ako vytvárať zombie


```
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