2. Procesy, signály a výjimky v C

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Část I

Procesy, signály a výjimky v C

Řízení toku programu

Procesory umí pouze jednu věc

- V čase mezi zapnutím a vypnutím (restartem) CPU čte a vykonává instrukce – program

Vnitřní mechanismy změny toku programu

- Skoky a větvení
- Volání a návraty z funkcí

Programy obtížně reagují na změny stavu systému

- Zápis nebo změna dat jiným procesem
- Neošetřené dělení nulou
- Uživatel stiskl Ctrl+C
- Přetekl systémový čítač

Mechanismy změny toku programu

Nízkoúrovňové

- **Výjimky**
  - Změna toku programu na základě událostí v operačním systému
  - Implementace: hardware + operační systém

Vysokoúrovňové

- **Přepnutí kontextu**
  - Přepnutí řízení mezi procesy, kontext je především stav registrů procesoru
  - Implementace: programový čítač + operační systém
- **Signály**
  - Implementace: operační systém
- **Nelokální skoky**
  - `setjmp()` a `longjmp()`
  - Implemetance: C runtime

I. Procesy, signály a výjimky v C

Proces

Proces je instance běžícího programu

- Proces představuje záznam v tabulce spravované operačním systémem (OS)
- Procesor
  - rozvrhuje proces ke spuštění (scheduler) na dostupných procesorech
  - přiděluje procesu pamět

Proces poskytuje programu

- Exkluzivní přístup k CPU
  - Mechanismus: **přepínání kontextu**
- Privátní adresový prostor
  - Mechanismus: **virtuální pamět**

Proces vs. vlákno

- vlákno (thread) je systémový objekt, který existuje uvnitř procesu, všechna vlákna jednoho procesu sdílí stejný paměťový prostor kromě registrů CPU a zásobníku
- paměť se přiděluje procesům
- CPU se přidělují vláknům
- vlákna jednoho procesu mohou běžet na různých procesorech

Proces

Proces je instance běžícího programu

- Proces představuje záznam v tabulce spravované operačním systémem (OS)
- Procesor
  - rozvrhuje proces ke spuštění (scheduler) na dostupných procesorech
  - přiděluje procesu pamět

Proces poskytuje programu

- Exkluzivní přístup k CPU
  - Mechanismus: **přepínání kontextu**
- Privátní adresový prostor
  - Mechanismus: **virtuální paměť**

Proces vs. vlákno

- vlákno (thread) je systémový objekt, který existuje uvnitř procesu, všechna vlákna jednoho procesu sdílí stejný paměťový prostor kromě registrů CPU a zásobníku
- paměť se přiděluje procesům
- CPU se přidělují vláknům
- vlákna jednoho procesu mohou běžet na různých procesorech

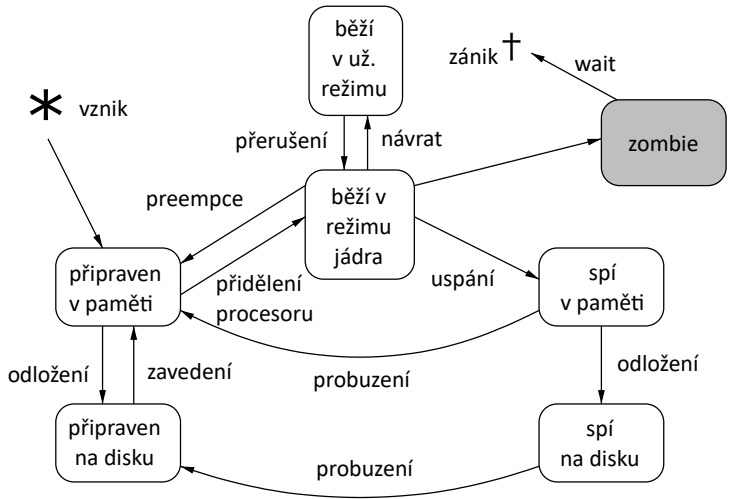
Životní cyklus procesu

Z perspektivy programátora je proces v jednom z následujících stavů

- **Spuštěný**
  - běží a využívá procesor
- **Blokovaný**
  - čeká na periferii
- **Uspaný**
  - čeká na CPU / plánovač

Preemptivní plánování

Jestliže se proces nevzdá CPU (neuspí se čekáním na nějakou událost), je mu odebrán procesor po uplynutí časového kvanta



Vytvoření a ukončení procesu

Proces může být ukončen z několika důvodů

- Příjem signálu, jehož defaulní akcí je ukončení procesu (terminate)
- Návrat z hlavní funkce `main()`, nebo volání funkce `exit()`

Běžící procesy mohou vytvářet potomky pomocí funkce `fork()`

- Funkce vrací `0` potomkovi, PID potomka rodiči
- Potomek je téměř identický jako rodič
  - potomek získá kopii (otisk) virtuálního adresového prostoru rodiče
  - potomek získá deskriptory otevřených souborů
  - potomek má svoje vlastní PID
- `fork()` je funkce zajímavá (a občas matoucí) tím, že se jednou volá, ale dvakrát vrací
- Proces lze identifikovat pomocí funkcí `getpid()` (vrací PID volajícího procesu) a `getppid()` (vrací PID rodiče)

Příklad vytvoření procesu pomocí `fork()`

```
1  int main int argc , char** argv
2  {
3    pid_t pid;
4    int x = 1;
5
6    pid = Fork ();
7    if (pid == 0) { /* Child */
8      printf ("child : x=%d\n", ++x);
9      return 0;
10   }
11   /* Parent */
12   printf ("parent: x=%d\nn", x);
13   return 0;
14 }
```

lec02/fork.c

Drobná vylepšení

- Zřejmě nelze zaručit pořadí, v jakém vznikají procesy
- Situace se dá zlepšit zařazením náhodně dlouhého čekání

```
1  pid_t fork(void) {                    // fork wrapper
2    initialize();
3    int parent_delay, child_delay;  // podle potreby
4    pid_t parent_pid = getpid ();
5    pid_t child_pid_or_zero = real_fork();
6    if (child_pid_or_zero > 0) {    // rodic
7      printf ("Child PID=%i, delay=%ims. Parent PID=%d, delay=%i ms\n",
8      child_pid_or_zero, child_delay, parent_pid, parent_delay);
9      fflush (stdout);
10     ms_sleep(parent_delay);
11   } else ms_sleep (child_delay); // potomek
13     return (child_pid_or_zero);
14 }
```

Zombie proces

- Může se stát, že jeden z procesů (rodič nebo potomek) neskončí
- Vzniká proces zombie, který se sám neukončí

```
1  void f ()
2  {
3    if (fork() == 0) /* Child */
4    {
5      printf ("Terminating Child, PID = %i\n", getpid ());
6      exit(0);
7    }
8    else
9    {
10     printf ("Running Parent, PID = %i\n", getpid());
11     while(1); // nekonecna smycka
12   }
13 }
```

Synchronizace rodiče s potomkem funkcí `wait()`

```
1  void f ()
2  {
3    int child_status;
5    if (fork() == 0) {
6      printf ("HC: hello from child\n");
7      exit(0);
8    }
9    else {
10     printf ("HP: hello from parent\n");
11     wait (&child_status);
12     printf ("CT: child has terminated\n");
13   }
14   printf ("Bye\n");
15 }
```

I. Procesy, signály a výjimky v C

Co je to signál?

- Signály jsou mechanizmem, pomocí kterého jádro asynchronně komunikuje s procesem.
- Každý signál je unikátní a je třeba ho zkoumat samostatně.
- Příkazem `kill -l` je možné zjistit, které signály váš systém podporuje.

```
standa@teplaky:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
...
```

Důležité signály

- SIGHUP – ukončení procesu
- SIGINT – ukončení procesu
  - úloze na popředí kdykoli se ve vstupním bufferu objeví znak interaktivního vyvolání pozornosti (typicky `Ctrl+C` s ASCII kódem 3)
- SIGQUIT – výpis paměti procesu (core dump)
  - SIGQUIT funguje stejně jako SIGINT, ale výchozí akce je jiná a aktivační znak je typicky `Ctrl+\`
- SIGCHLD
  - Když proces skončí nebo se pozastaví, jádro pošle rodičovskému procesu SIGCHLD. Spolu se signálem se přenáší další informace, jako číslo procesu, uživatele a návratová hodnota.
- SIGCONT – pokračování procesu
  - SIGCONT probudí pozastavený proces. Typicky ho posílá shell po použití příkazu `fg`. Protože signál SIGSTOP není možné zamaskovat, může neočekávaný signál SIGCONT sloužit jako indikace, že proces byl na nějakou dobu pozastaven.
- SIGTSTP – pozastavení
  - SIGTSTP funguje stejně jako SIGINT a SIGQUIT, jen používá kombinaci `Ctrl+Z` a výchozí akcí je pozastavení procesu.

Příklad – posílání signálu `kill` mezi procesy

```
1   pid_t pid [N];
2   int i,  status;
4   for (i = 0; i < N; i++)
5     if (pid [i] = fork()) == 0) while (1);  // potomek nekonci
7   for (i = 0; i < N; i++) {
8     printf ("Zabijim proces %i\n", pid [i]);
9     kill (pid [i], SIGINT);
10  }
12  for (i = 0; i < N; i++) {
13    pid_t wpid = wait (&status);
14    if (WIFEXITED (status))
15      printf ("PID %d ukoncen (ex: %i)\n", wpid, WIFEXITED (status));
16    else
17      printf ("PID %d ukoncen nenormalne\n", wpid);
18  }
```

Příklad – reakce na signál

```c
void sigint_handler (int sig)   // SIGINT handler
{
  fprintf (stdout, "Co si to dovolujes?\n");
  fflush (stdout);
  sleep(2);
  fprintf (stdout, "OK, cau :-)\n");
  exit(0);
}
int main (int argc , char** argv)
{
  if (signal(SIGINT, sigint_handler) == SIG_ERR) // SIGINT handler
  fprintf (stderr, "signal_error");
  pause();  // cekani na prijem signalu
  return 0;
}
```

I. Procesy, signály a výjimky v C

Výjimky v C

- Jazyk C z nedisponuje sofistikovanými mechanismy ošetření výjimek
- Kromě funkce `exit()` se využívá také funkce `abort`, která zajistí
  - vyprázdnění vyrovnávací paměti,
  - ukončení celého programu,
  - vrácení čísla nadřazeného procesu,
  - vypsání `Abnormal program termination` na `stderr`

Dlouhé skoky

- Umožňují opustit aktuální funkci a vrátit se do některého z nadřízených (běžících!) bloků
- Pro práci s dlouhými skoky lze využít:
  - Datový typ `jmp_buf` pro uchování stavu prostředí
  - Makro `setjmp()` – zaznamenává stav prostředí a místo kam se program vrátí po provedení dlouhého skoku. Podle návratové hodnoty zjistíme, jestli se jedná o návrat po uložení prostředí (0) nebo po dlouhém skoku.
  - Funkci `long_jmp()` – provedení dlouhého skoku.

Dlouhé skoky

```
1  int i = setjmp (env);
3  if (i == 0)
4  {
5    f();
6    // zpracovani normalniho navratu z f()
7  }
8  else
9  {
10   // zpracovani navratu po dlouhem skoku
11 }
13 void f()
14 {
15   if(chyba)
16     longjmp (env, 1);
17 }
```