Procesy

- Tvorba procesov
- Ukončenie procesov
- Príkazy typu wait
- Signály
- Spracovanie signálov

Čo je to proces

- Procesom označujeme postupnosť inštrukcií, ktoré tvoria súvislú činnosť.
- Proces je definovaný:
  1. PID (Process Identifier) číslom
  2. vlastnou postupnosťou vykonávateľných inštrukcií
  3. vlastným pamäťovým priestorom
  4. vyhradenou množinou registrov procesora
  5. ukazovateľom adresy vykonávanej inštrukcie (program counter)
  6. hierarchiou vzťahov rodič – potomok

Príklady procesov

- externý príkaz shellu je samostatný proces
- spustením shellovského scriptu alebo skupiny príkazov spojených kanálmi sa štartuje jedna úloha s mnohými procesmi.

Proces ID

- Každý proces má priradený jednoznačný identifikátor tzv. proces ID alebo tiež pid. pid je 16 bitové číslo, ktoré je priradené procesu vtedy keď je vytvorený.
- Systémové volanie getpid( ) umožní získať pid bežiaceho procesu.
- Systémové volanie getppid( ) – vracia pid (ppid-parent proces ID) rodičovského procesu.

Príklad

```
#include <stdio.h>
#include <unistd.h>
int main ( )
{
  printf (“The process ID is %d\n”, (int) getpid ());
  printf (“The parent process ID is %d\n”, (int) getppid ());
  return 0;
}
```

- pid bude pri rôznych volaniach uvedeného programu vždy rôzne.
- ppid bude rovnaké v prípade ak sa program volá v tom istom shelly.

Tvorba procesov

- Existujú dva spôsoby tvorby procesov
  1. volaním funkcie system()
  2. volaním funkcií fork() a exec()
- Funkcia *system*

  Pomocou funkcie *system* možno volať príkazy shellu práve tak, ako by bol príkaz napísaný v príkazovom riadku shellu. Pri volaní sa vytvorí subproces shellu, ktorý vykoná príkaz.

  Takáto tvorba procesu je neefektívna a nie je **bezpečná.**

Príklad1

prečo je funkcian system() nie bezpečná

```
#include <stdio.h>
#include <stdlib.h>
/* Returns a nonzero value if and only if the WORD appears in
    /usr/dict/words. */
int grep_for_word (const char* word)
{
    size_t length;
    char* buffer;
    int exit_code;
/* Build up the string ‘grep -x WORD /usr/dict/words’. Allocate the
string dynamically to avoid buffer overruns. */
    length = strlen (“grep -x “) + strlen (word) + strlen (“ /usr/dict/words”) + 1;
    buffer = (char*) malloc (length);
    sprintf (buffer, “grep -x %s /usr/dict/words”, word);
```

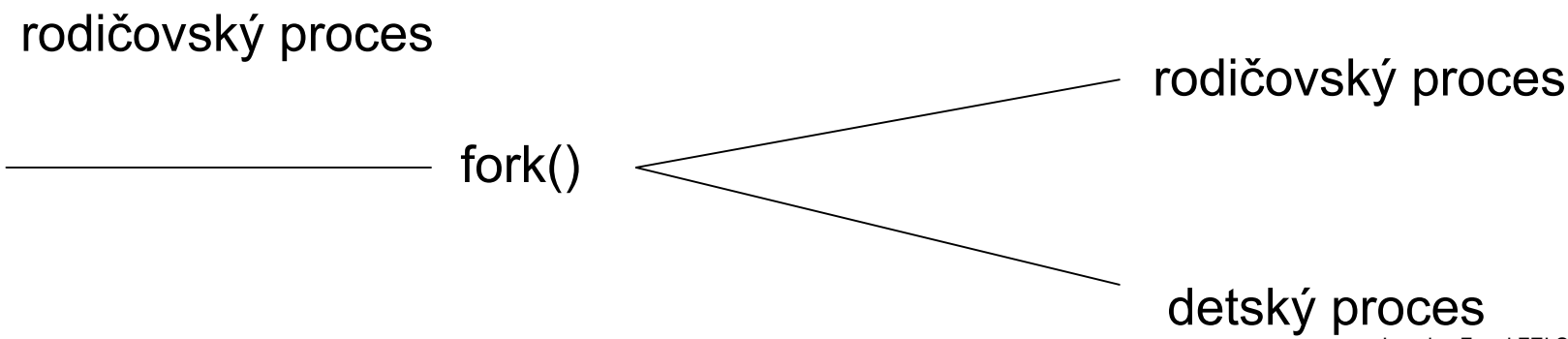
```
  /* Run the command. */
  exit_code = system (buffer);
  /* Free the buffer. */
  free (buffer);
  /* If grep returned 0, then the word was present in the dictionary. */
  return exit_code == 0;
}
```

- Čo sa stane ak hľadaný reťazec je foo /dev/null; rm -rf .
- Výkoná sa nasledovný príkaz:

  grep -x foo /dev/null; rm -rf . /usr/dict/words
- Shell ho interpretuje ako dva sekvenčné príkazy oddelené “;”, vykoná grep a následne vymaže aktuálny adresár.

Tvorba procesov

- Funkcia *fork*
- Fukcia *fork* vytvorí kópiu procesu, ktorý ju volá - detský proces.
- Oba procesy vykonávajú ten istý program od toho miesta, kde bola funkcia fork volaná a majú rozdielne pid.



Systémovým volaním: fork()

– sa vytvorí nový proces. Nový proces (child) je presná kópia volajúceho procesu (parent).

Príklad:

```
#include <unistd.h>
int main(int argc, char **argv){
fork();                                          //vytvorime kopiu procesu
write(1,"Tento text sa vypise 2x\n",24);         //tu uz budu 2 procesy
return 0;
}
```

Čo majú vzniknuté procesy rovnaké?

- Obidva procesu majú rovnaky kód a údaje. "Rovnaky" ale v tomto pripáde neznamená "ten isty".
- Každý proces ma vlastnú kópiu údajov a jeden druhému do nich nemožu zapisovať

```
int main (int argc, char **argv){
  int i=0;
  fork();
  i++;
  printf("%d\n",i);       // program vypíše 2=krát 1
  return 0;               // každý proces obsahuje vlastnú kópiu i
}
```

Čo vracia *fork*

- Funkcia *fork* poskytuje dve rozdielne návratové hodnoty. V rodičovskom procese je to pid detského procesu v detskom procese je to 0.
- V nasledovnom programe sa prvý blok if príkazu vykonáva v rodičovskom procese zatiaľ čo blok po else sa vykonáva v detskom procese.

Príklad2

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
  pid_t child_pid;
  printf (“the main program process ID is %d\n”, (int) getpid ());
  child_pid = fork ();
  if (child_pid != 0) {
  printf (“this is the parent process, with id %d\n”, (int) getpid ());
  printf (“the child’s process ID is %d\n”, (int) child_pid);
  }
  else
  printf (“this is the child process, with id %d\n”, (int) getpid ());
  return 0;
}
```

Tvorba procesov

- Čo je cieľom tvorby procesov?
  - Vykonávať príkazy alebo programy .
  - fork() samotné to však celkom nespĺňa keďže kódy procesov sú rovnaké.
  - Potrebujeme niečo ako príkaz, ktorý spúšťa vopred skompilované príkazy alebo programy.
  - V UNIXe to zabezpečujú systémové volania triedy exec.

Systémové volania triedy *exec*

- Natiahnu nový vykonávací kód na miesto procesu, ktorý volá exec.
- Volajúci proces sa v prípade úspešného volania exec nikdy nevráti na inštrukciu nasledujúcu po exec.
- Viac detailov v ‘man –a exec’
- Príklad:
  *int execv(const char *path, char *const argv[]);*

```
#include <stdlib.h>
#include <unistd.h>
#include <iostream>
#include <stdio.h>
Using namespace std;
main( ) {
  char command[100];
  char *argv[10];
  strcpy(command, "/bin/ls");
  argv[0] = command;
  argv[1] = "/";
  argv[2] = NULL;
  execv(command, argv);
  cout << "toto sa nikdy nevykona\n";
}
```

vykoná sa príkaz:
("/bin/ls ")

- argv má význam ako v programoch C/C++ .
- argv[0] musí vždy obsahovať názov programu (príkazu).
- Posledný prvok v argv[] musí obsahovať smerník na NULL.

Funkcia *exec*

- Funkcia *exec* nahradí program bežiaci v procese iným programom, ktorý je zvyčajne argumentom funkcie.
- Triedy funkcií *exec*

  Ak obsahujú *p* (execvp a execlp) argumentom je meno programu, ktorý sa má vykonať a program musí byť v nastavených adresároch,

  ak neobsahujú *p* treba zadať úplnú cestu k programu, ktorý sa má vykonať.

Triedy funkcií *exec*

- Ak obsahujú písmeno *v* (execv, execvp, execve) akceptujú zoznam argumentov programu ako pole smerníkov na reťazce (ukončené NULL).
- Funkcie obsahujúce *l* (execl, execlp, execle) akceptujú argumenty ako C jazyk vararg.
- Ak obsahujú *e* (execve, execle) akceptujú prídavné argumenty poľa premenných prostredia. Argumentom je pole smerníkov na reťazce tvaru „VARIABLE=value“. Pole je ukončené NULL.

Príklad3

- Nasledovný program vytvori detský proces, ktorý sa zmení na výkon shellovského príkazu ls -l

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
/* Spawn a child process running a new program.
   PROGRAM is the name of the program to run; the path
   will be searched for this program. ARG_LIST is a NULL-
   terminated list of character strings to be passed as the
   program's argument list. Returns the process ID of the
   spawned process. */
```

Príklad3

```
int spawn (char* program, char** arg_list)
{
  pid_t child_pid;
  /* Duplicate this process. */
  child_pid = fork ();
  if (child_pid != 0)
  /* This is the parent process. */
  return child_pid;
  else {
  /* Now execute PROGRAM, searching for it in the path. */
  execvp (program, arg_list);
  /* The execvp function returns only if an error occurs. */
  fprintf (stderr, “an error occurred in execvp\n”);
  abort ();
  }
}
```

Príklad3

```
int main ()
{
  /* The argument list to pass to the “ls” command. */
  char* arg_list[] = {
 “ls”,                    /* argv[0], the name of the program. */
 “-l”,                    /* option of the command */
 “/”,                     /* root directory*/
  NULL                    /* The argument list must end with a NULL. */
  };
  /* Spawn a child process running the “ls” command.  */
  spawn (“ls”, arg_list);
  printf (“done with main program\n”);
  return 0;
}
```

Ukončenie procesov

- Normálne ukončenie

  Proces skončí dvoma spôsobmi:

  volaním *exit* funkcie

  návratom z *main* funkcie
- Každý proces má exit hodnotu – číslo, ktoré po ukončení vracia rodičovskému procesu, buď ako argument exit funkcie alebo návratovú hodnotu main funkcie.

Ukončenie procesov

- Abnormálne ukončenie
  1. Odozva na signály: SIGBUS (bus error), SIGSEGV (segment violation), SIGFPE (floating point exception) – ak sa proces pokúša vykonať ilegálnu operáciu.
  2. Ukončenie inými signálmi: SIGTERM (proces ho môže ignorovať maskou), SIGKILL (proces je ukončený okamžite), SIGINT (stlačením Ctrl+C).

Ukončenie z iného programu

- kill funkcia

  kill(proces_pid, signal)

  argumenty sú:

  proces_pid – ID cieľového procesu

  signal – číslo signálu

  Pr.

  kill(child_pid, SIGTERM)
- Ako návratová hodnota ukončeného procesu môže byť číslo 0-127.
- Návratová hodnota nad 128 má špeciálny význam: keď je proces, ukončený signálom má exit hodnotu 128+číslo signálu.

Čakanie na ukončenie procesu

- Pri spustení programu z príkladu3 sa často stane, že výstup z ls nastane často až po ukončení funkcie main.

  Dôvod je ten že detský proces, ktorý volá ls a rodičovský proces sú nezávislé procesy, ktoré bežia súčasne a nie je možné predikovať, ktorý bude bežať prv alebo neskôr.
- Je nutné aby rodičovský proces čakal na ukončenie jedného alebo viacerých svojích detských procesov (aby mali komu odovzdať návratovú hodnotu a nedostali sa do stavu „zombie“).

  Na to slúži systémové volanie z *wait* triedy funkcií.

Systémové volania *wait*

- wait (int* child_status) – blokuje volajúci proces pokiaľ niektorý z jeho detských procesov neskončil (normálne alebo abnormálne).
- Funkcia vracia cez svoj argument stav, z ktorého pomocou makier:
  1. WIFEXITED- možno zistiť príčinu ukončenia (normálne, abnormálne)
  2. WTERMSIG- možno zistiť číslo signálu, ktorý detský proces ukončil.

main funkcia z príkladu3

```
int main ()
{
  int child_status;
  /* The argument list to pass to the “ls” command. */
  char* arg_list[] = {
  “ls”,                    /* argv[0], the name of the program. */
  “-l”,
  “/”,
  NULL          /* The argument list must end with a NULL. */
  };
 /* Spawn a child process running the “ls” command. */
  spawn (“ls”, arg_list);
```

```
/* Wait for the child process to complete. */
  wait (&child_status);
  if (WIFEXITED (child_status))
  printf (“the child process exited normally, with exit code
    %d\n”, WEXITSTATUS (child_status));
  else
  printf (“the child process exited abnormally\n”);
  return 0;
}
```

Ďalšie funkcie triedy *wait*

- waitpid() – čaká sa na ukončenie špecifikovaného detského procesu
- wait3(), wait4()- funkcie poskytujú ďalšie informácie o ukončených procesoch

Zombie proces

- Ak detský proces skončil zatiaľ čo rodičovský proces čaká na jeho ukončenie- detský proces prestane ďalej existovať.
- Ak detský proces skončil zatiaľ čo rodičovský proces ešte nevykonal wait()- detský proces existuje ďalej ako zombie proces.

- Zombie proces je proces, ktorý je ukončený ale je ešte stále vedený ako proces. Keď rodičovský proces vykoná wait() zombie stav detského procesu je ukončený a proces je vymazaný z tabuľky procesov.

- Čo sa stane ak rodičovský proces nečaká na detský proces?

Príklad

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
  pid_t child_pid;
  /* Create a child process. */
  child_pid = fork ();
  if (child_pid > 0) {
  /* This is the parent process. Sleep for a minute. */
  sleep (60);
  }
  else {
  /* This is the child process. Exit immediately. */
  exit (0);
  }
  return 0;
}
```

Zombie proces

- Čo zistíme:

  ak rodičovský proces ešte beží (sleep (60)) po príkaze

  % ps -e -o pid,ppid,stat,cmd

  zistíme, že existuje zombie proces označený ako <defunct > so stavom Z.
- Po ukončení rodičovského procesu, ktorý nevolá wait() je jeho detský proces zdedený procesom init(), ktorý automaticky ukončí všetky zdedené zombie procesy a vymaže ich z tabuľky procesov.

Signály

- Signály sú asynchrónne prerušenia generované shellom alebo iným procesom ako odozva na nejakú chybovú situáciu.
- Sú reprezentované číslom a názvom

Signály

Signal Nmb.	Name	Function
1	SIGHUP	prerušenie spojenia modemu
2	SIGINT	prerušenie z terminálu
3	SIGQUIT	proces vykoná core dump súbor
9	SIGKILL	ukončenie procesu
15	SIGTERM	ukončenie procesu používané príkazom kill
24	SIGTSTP	suspendovanie procesu *[Ctrl-z]*

Spracovanie signálov

- Proces reaguje na signál nasledovne:
  1. ignoruje ho
  2. vykoná sa defaultová funkcia
  3. vykoná sa špeciálna funkcia (signal handler function)
- Funkcia sigaction spracuje signál

  int sigaction(int sig_number, sigaction* sa_str, sigaction* sa_str)

  Kde

  prvý argument je číslo signálu

  druhé dva parametre sú smerníky na štruktúru sigaction, ktorá obsahuje člena sa_handler, ktorý môže mať jednu z nasledovných hodnôt:
  1. SIG_DFL – špecifikuje defaultové spracovanie signálu
  2. SIG_IGN – signál bude ignorovaný
  3. smerník na funkciu, korá sa po vzniknutí signálu vykoná. Jej argumentom je číslo signálu.

Štruktúra *sigaction*

```
struct  sigaction   {
                    sigset_t   sa_mask;
                    int          sa_flags;
                    void         sa_handler;
        /*podla nastavenia sa_flags
          buď smerník na obslužnú funkciu typu
         void  meno (int)
          alebo smerník na obslužnú funkciu typu:
         void   meno (int,  siginfo_t  *,  struct ucontext  *)*/
    }
```

Príklad

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

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);
  /* Do some lengthy stuff here. */
  printf (“SIGUSR1 was raised %d times\n”, sigusr1_count);
  return 0;
}
```

- Poznámka

  Ak sa vo funkcii signál handlera použije globálna premenná musí byť špeciálneho typu **sig_atomic_t**. Tým je zaručené, že priradenie hodnoty premennej tohto typu nemôže byť prerušené.

Asynchrónne vymazanie zombie procesov

- Niekedy je potrebné aby rodičovský proces nečakal na ukončenie svojich potomkov ale aby pokračoval v činnosti.
- Ako zabezpečíme, aby všetky detské procesy po ukončení nezostávali v stave zombie, v ktorom zaberajú systémove zdroje?
- Elegantné riešenie je použiť asynchrónny spôsob spracovania signálov.

  Využíva sa tá skutočnosť, že detský proces po svojom ukončení zašle signál SIGCHLD rodičovi.

Asynchrónne vymazanie zombie procesov

- Nasledovný program zachytí signál a funkciou signál handlera, ktorá obsahuje wait( ) zabezpečí, že všetky detské procesy, ktoré skončili svoju činnosť budú zrušené.
- Výstupný stav procesov sa odpamätá do globálnej premennej pre prípadné ďalšie použitie.

Príklad

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
sig_atomic_t child_exit_status;
void clean_up_child_process (int signal_number)
{
  /* Clean up the child process. */
  int status;
  wait (&status);
  /* Store its exit status in a global variable. */
  child_exit_status = status;
}
```

```
int main ()              /*Parent process */
{
  /* Handle SIGCHLD by calling clean_up_child_process. */
  struct sigaction sigchld_action;
  memset (&sigchld_action, 0, sizeof (sigchld_action));
  sigchld_action.sa_handler = &clean_up_child_process;
  sigaction (SIGCHLD, &sigchld_action, NULL);
  /* Now do things, including forking a child process. */
  return 0;
}
```