# Řízení procesů

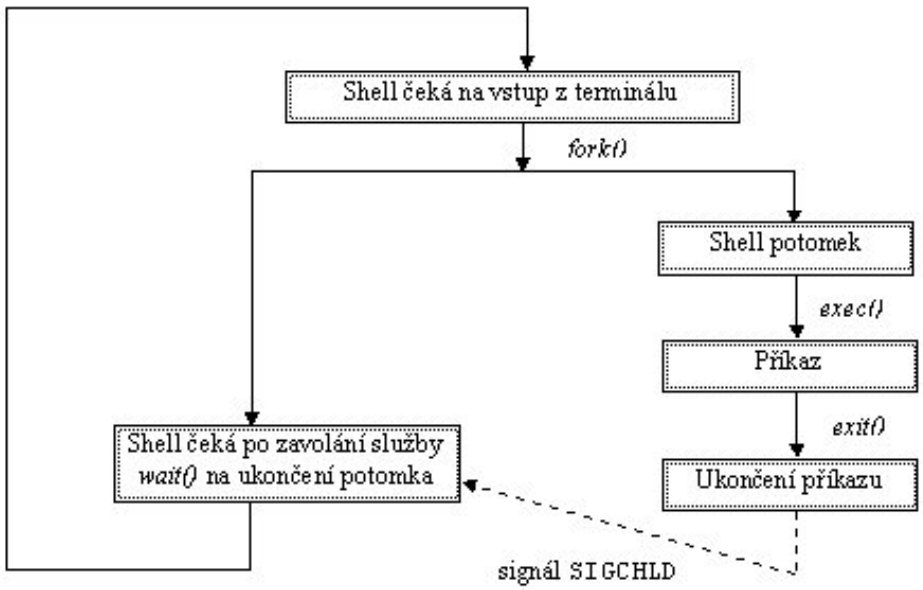
Součástí strojového kódu programu, který řídí proces, může být volání služby systému realizované pomocí instrukce vnitřního přerušení. Mezi službami systému jsou také služby, které mohou zásadním způsobem ovlivnit další průběh procesu. V této kapitole probereme pouze ty nejdůležitější :

| | |
|---|---|
| ***fork()*** | - vytvoření nového procesu (potomka) rodičovským procesem |
| ***pause()*** | - operační systém uvede proces do stavu zablokován |
| ***wait()*** | - proces rodič čeká na ukončení potomka |
| ***exit()*** | - proces žádá operační systém o své ukončení |
| ***exec()*** | - operační systém zamění text procesu, změní řízení procesu |
| ***signal()*** | - slouží k ošetření signálu zaslaného procesu |
| ***alarm()*** | - OS zašle procesu signál SIGALRM po uplynutí časového intervalu |
| ***kill()*** | - proces žádá OS o zaslání signálu jinému procesu |

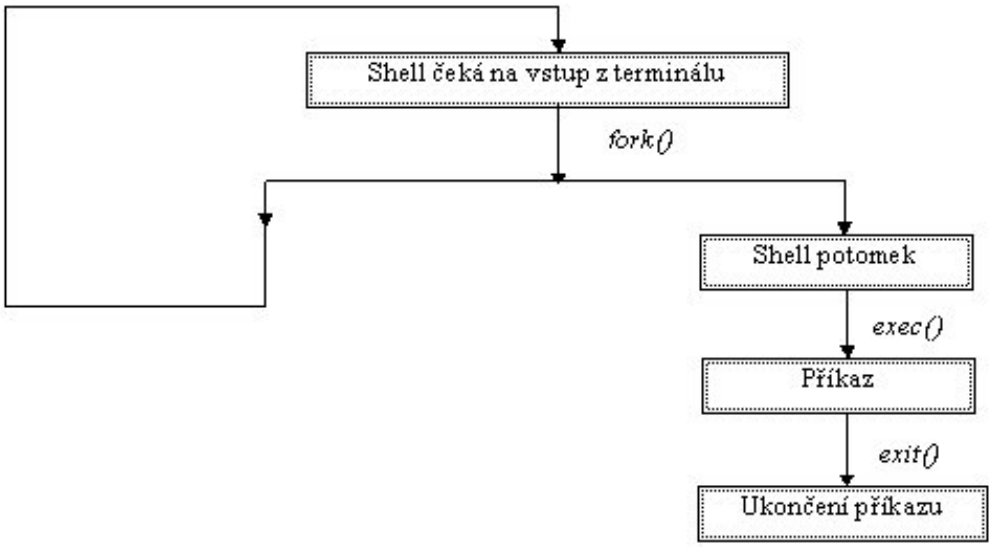
Služby systému se volají pomocí vnitřního přerušení. Protože jazyk C neobsahuje jazykovou konstrukci pro generování vnitřního přerušení, jsou služby systému realizovány pomocí funkcí, které jsou součástí standardní knihovny jazyka C, tj. knihovny **libc.a** . Tyto funkce byly napsány v assembleru, byly přeloženy a zařazeny do této knihovny. Služby systému se volají tak, že se volá ta funkce z knihovny **libc.a**, která službu systému realizuje.

## Spouštění procesu

### Spuštění procesu na popředí



### Spuštění procesu na pozadí




## Vytvoření nového procesu

Proces může vytvořit nový proces voláním služby ***fork()***

```
int fork(void);
pid=fork();
```

Všechny procesy, které v systému existují, vznikly tímto způsobem. Výjimkou je proces **swapper**, který má *PID* rovno 0 a který vytvořilo po spuštění systému jádro systému.

Po zavolání služby *fork()* operační systém vytvoří nový proces, který je kopií původního procesu. Původní proces se nazývá ***proces rodič***, nově vzniklý proces se nazývá ***proces potomek*** nebo ***proces dítě (child process)***.

Systémový kontext obou procesů se liší jen ve dvou položkách:

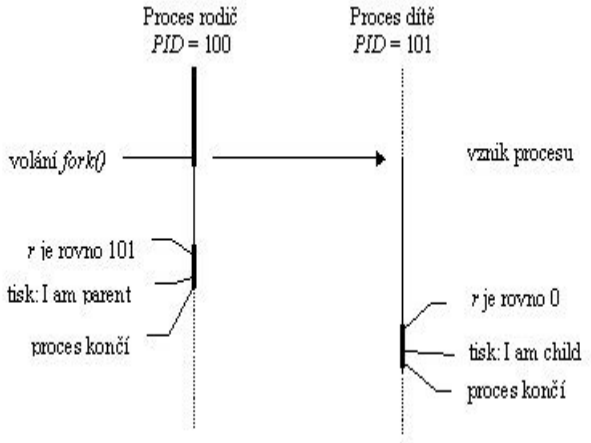
1. Procesu potomka přiřadí operační systém nové identifikační číslo *PID*. Obvykle je to číslo o jedničku vyšší než má proces rodič. Pokud toto číslo již má přiřazen jiný proces, přiřadí potomku první vyšší číslo, které je volné.

2. Do u-oblasti, na místo, kde se ukládá výsledek volání systému, uloží operační systém procesu rodiči *PID* potomka a procesu potomka 0. To, že proces potomka nezná ani své *PID* ani *PID* svého rodiče nevadí. Existují volání systému ***getpid()*** a ***getppid()***, pomocí kterých může proces svoje *PID* i *PID* svého rodiče získat.

**Příklad 1:** Volnáním služby *fork()* vytvořte nový proces. Vypište na obrazovku rodičovský a dětský proces.

***Řešení***

```
#include <stdio.h>
main()
{
int r;
if((r=fork())== 0)
 printf("Já jsem potomek");
else
 printf("Já jsem rodič");
}
```




**Příklad 2:** Napište program, který po svém spuštění vytvoří proces potomka. Proces potomek vypíše své *PID* a *PID* svého rodiče a požádá operační systém o převedení do stavu čekání. Proces rodič vypíše své *PID* a *PID* svého potomka a požádá operační systém o ukončení.

***Řešení***

```
#include <stdio.h>
main()
{
int pid;
if  ((pid=fork()) == 0)
  {
     printf("\nProces potomek: PID=%d PID rodice=%d\n",
       getpid(),getppid());
     pause();
  }
printf("\nProces rodic: PID=%d PID potomka=%d\n", getpid(),
  pid);
exit(0);
}
```

# Skupina procesů

Každý proces patří do určité skupiny procesů. Skupina procesů je jednoznačně identifikována svým číslem skupiny procesů *PGRP*. Každá skupina procesů má svého vedoucího. Vedoucí skupiny procesů má *PID* rovno *PGRP*. Je zřejmé, že vedoucím skupiny procesů může být jen jeden proces.

Proces s *PID*=0 má *PGRP*=0. Příslušnost do skupiny procesů se dědí. Každý proces ale může požádat operační systém voláním ***setpgrp()*** o to, aby se stal vedoucím nové skupiny procesů. Po zavolání služby systému

```
int setpgrp(void);
grp=setpgrp();
```

operační systém nastaví procesu číslo skupiny procesů na hodnotu jeho *PID* . Protože pro tento proces potom platí *PID=PGRP* , stal se tento proces vedoucím nově vzniklé skupiny procesů. Při volání operační systém vrátí procesu novou hodnotu jeho skupiny procesů `grp` .

Číslo své skupiny procesů může proces získat zavoláním služby ***getpgrp()***

```
int getpgrp(void);
grp=getpgrp();
```

*Poznámka*. Ve BSD versích Unixu má volání systému *setpgrp()* odlišnou syntaxi.

# Ukončení procesu

Proces končí svou činnost voláním služby ***exit()***

```
void exit(int status);
exit(status);
```

Operační systém ukončí proces a zajistí vrácení 8 nejméně významných bitů *statusu* čekajícímu rodiči.

Proces také ukončí svou činnost po návratu z funkce *main()*, tj. když dojde na konec programu nebo po příchodu signálu, který nebyl procesem zpracován. V těchto případech se provede volání služby *exit()* automaticky.

Výsledkem volání služby *exit()* je, že proces přejde do stavu zombie. V tomto stavu je tak dlouho, dokud není zpracován svým rodičem.

## Zpracování procesů zombie

Proces rodič by měl být řízen programem napsaným tak, aby zpracoval svoje procesy potomky, které ukončí činnost a dostanou se do stavu zombie.

Proces může zpracovat svého potomka ve stavu zombie tak, že zavolá službu systému ***wait( )***

```
int  wait(int *stat_adr);
pid=wait(stat_adr);
```

Jakmile proces volá službu *wait()* , je jádrem převeden do stavu zablokování. Probudí se až po příchodu signálu. Měl by být probuzen signálem `SIGCHLD` , který mu zašle jádro při ukončení činnosti některého z jeho potomků.

Pokud končící proces volal službu *exit(status)* s parametrem `status` , uloží se poslední byte proměnné `status` (tj. bity 0-7) do předposledního bytu proměnné `stat_adr` (tj. do bitů 8-15). Bity v tomto případě číslujeme zprava doleva (tj. od nejméně významného k nejvíce významnému bitu ).

Pokud je signál `SIGCHLD` ošetřen takto:

```
signal(SIGCHLD,SIG_IGN);
```

potom zpracováním potomka služba *wait()* nekončí, ale je proveden její restart.

## Záměna textu procesu – služba *execl*()

V jazyce C existuje celá řada funkcí, které lze k záměně textu procesu použít. Tyto funkce byly realizovány pro pohodlí uživatelů . Všechny ale nakonec volají základní funkci *execve()* . Uveďme nejoblíbenější z nich a sice funkci *execl()* . Funkce se volá následovně:

```
execl(char *path,char *argv0,char *argv1 , . . . , NULL);
ret=execl(path,argv0,argv1 , . . . , NULL);
```

Funkce spustí program uložený v souboru, který je určen cestou `path` a spouštěnému programu předá ukazatele na parametry `argv0, argv1, …` . V argumentu `argv0` je nutno předat ukazatel na název spouštěného programu.

**Příklad 3:** S použitím volání funkce *execl()* napište program, který po svém spuštění vytvoří proces potomka a bude čekat na jeho skončení. Proces potomek požádá operační systém o spuštění programu `/bin/ls` s parametry `-al` a `/etc` . Program `/bin/ls` vypíše adresář `/etc` a skončí. Po ukončení procesu potomka, proces rodič zpracuje zombie potomka a skončí.

***Řešení***

```
#include <stdio.h>
main()
{
  int status;
  if(fork() == 0)
    execl("/bin/ls","ls","-al","/etc",NULL);
  wait(&status);
  printf("\nProces potomek skoncil se statusem %d\n",\
    status);
  exit(0);
}
```

**Příklad 4:** V jazyce C napište program, který postupně spustí programy **date** a **ls -al /etc** a po jejich skončení vypíše hlášení `Programy date a ls skoncily .`

***Řešení***

```
#include <stdio.h>
main()
{
     if (fork() == 0){
          execl("/bin/date","date",NULL);
          printf("date nelze spustit");
          exit(1);
     }
     if (fork() == 0){
          execl("/bin/ls","ls","-al","/etc",NULL);
          printf("ls nelze spustit");
          exit(1);
     }
     while (wait(NULL)>0);
     printf("Programy date a ls skoncily\n");
}
```

# Zasílání signálů procesům

Signály informují procesy o asynchronních událostech. Signály může poslat buď jádro operačního systému nebo jiný proces prostřednictvím volání služby systému ***kill()*** .

Pokud byl procesu zaslán signál, operační systém zajistí zpracování signálu před spuštěním procesu. Způsob zpracování signálu závisí na tom, zda byl příchod signálu ošetřen nebo nebyl.

Pokud příchod signálu nebyl ošetřen, provede se implicitní zpracování signálu. To pro většinu signálů spočívá v ukončení procesu. Výjimkou jsou signály `SIGCHLD` a `SIGCONT`, jejichž příchod znamená pouze odblokování procesu.

## Ošetření signálu

Ošetření signálu je třeba provést před příchodem signálu voláním služby systému ***signal()***.

```
void(*signal(int sig,void(*func)(int)))(int);
last_func=signal(sig,func);
```

kde

`sig` je číslo signálu (signálů je celkem 32)

`func` má jednu z hodnot

`SIG_IGN` Znamená ignorování signálu (Signál 9 a 19 nelze ignorovat).

`SIG_DFL` Obnovení implicitního nastavení.

*Ukazatel na funkci* uvnitř programu, která má být po příchodu signálu provedena.

Volání funkce vrací hodnotu `last_func` , což jest hodnota `func` při posledním volání služby ***signal()*** se stejnou hodnotou `sig`. Funkce `func` , která se provádí po příchodu signálu má jeden parametr typu **int**, pomocí něhož se funkci předává číslo zachyceného signálu.

Příchod signálu u většiny operačních systémů zruší nastavení ošetření signálu. Aby byl ošetřen i další příchod signálu, je třeba ošetření signálu znovu nastavit. To lze nejlépe provést tak, že ve funkci `func`, která se po příchodu signálu vykonává, se znovu volá služba systému *signal()* .

***Nejdůležitější signály:***

| | | |
|---|---|---|
| 1 | SIGHUP | A. Po vypnutí terminálu zašle jádro operačního systému všem procesům, které patří do skupiny řídícího procesu vypnutého terminálu.<br>B. Pokud je ukončen proces, který je vedoucím skupiny a má přiřazen řídící terminál, odešle jádro tento signál všem členům jeho skupiny. |
| 2 | SIGINT | Posílá právě probíhajícímu procesu ovladač terminálu při zmáčknutí klávesy přerušení (obvykle CTRL+C). |
| 9 | SIGKILL | Užívá se ke spolehlivému ukončení procesu. Tento signál nelze zachytit. |
| 14 | SIGALRM | Zasílá jádro po volání systému *alarm(n)*. Parametr *n* určuje, po jaké době (v sekundách) po zavolání této služby, jádro signál procesu zašle. |
| 15 | SIGTERM | Ukončení procesu. Zasílá implicitně program **kill**. |
| 17 | SIGCHLD | Zasílá jádro procesu rodiči při ukončení činnosti některého jeho potomka. Příchod tohoto signálu neznamená ukončení procesu rodiče ani tehdy, když nebylo předem nastaveno ošetření signálu pomocí služby *signal()* . |
| 18 | SIGCONT | Odblokování procesu zastaveného signálem SIGSTOP. |
| 19 | SIGSTOP | Pozastavení procesu. Signál nemůže být zachycen. |

## Zasílání signálů

Procesy mohou zaslat signál jinému procesu pomocí služby systému ***kill()*** :

```
int  kill(int pid, int sig);
ret=kill(pid, sig);
```

kde

`sig` je číslo zasílaného signálu

`pid` má tento význam:

je-li `pid` > 0 , zasílá se signál procesu s *PID* = `pid`

je-li `pid` = 0 , signál se zašle všem procesům stejné skupiny

je-li `pid` = -1 , signál se zašle všem procesům

je-li `pid` < -1 , signál se zašle všem procesům, které patří do skupiny procesů s číslem `-pid`

Navrácená hodnota `ret` je nulová, pokud volání proběhlo úspěšně. V opačném případě je rovna -1 (to nastane například tehdy, když proces, kterému má být signál zaslán neexistuje)

**Příklad 5:** Napište program, který bude každých 5 sekund vypisovat na obrazovku řetězec znaků. Program napište tak, aby jej nebylo možné zrušit signálem 2. Pokud program poběží na popředí, nepůjde zrušit stisknutím kláves `CTRL+C`. Vysvětlete.

***Řešení:***

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

void handler(sig)
int sig;
{
     if (sig == 2)
          printf("Signalem 2 mne nelze zrusit\n");
     signal(2,handler);
     signal(14,handler);
}

main()
{
     int i;
     signal(2,handler);
     signal(14,handler);
     for(i=0;i<100;i++)
     {
          alarm(5);
          pause();
          printf("Ahoj!\n");
     }
}
```

Pokud proces, který program řídí, běží na popředí a pokud stiskneme klávesy `CTRL+C`, ovladač klávesnice procesu zašle signál 2. Protože signál 2 je ošetřen nedojde ke zrušení procesu.

*Poznámka:* Zrušení běžícího procesu je možné provést spuštěním dalšího terminálu (shellu), pomocí příkazu

```
ps -a
```

zjistit PID procesu a pomocí příkazu *kill* zaslat tomuto procesu signál, který jej ukončí:

```
kill -9 PID
```

# Záměna programu řídícího proces

Pro záměnu programu, který řídí proces, za jiný program, existuje služba systému ***execve()***. Tato služba zamění strojový kód běžícího procesu za strojový kód programu, který je uložen v souboru na disku. Při spouštění nového programu mu předá pointer na pole, kde jsou uloženy pointery na hodnoty parametrů a pointer na pole, kde jsou uloženy pointery na proměnné prostředí. Služba systému ***execve()*** se volá následovně:

```
int execve(char *path, char *argv[], char *envp[]);

ret=execve(path, argv, envp);
```

kde

`path` je pointer na cestu identifikující soubor, ve kterém je uložen nově spouštěný program.

`argv` je pointer na pole pointerů, které odkazují na řetězce, které jsou hodnotami předávaných parametrů. První parametr musí obsahovat název programu. Poslední pointer v tomto poli musí být pointer NULL, tj. `(char * ) 0`

`envp` je pointer na pole pointerů, které ukazují na řetězce obsahující definice hodnot proměnných prostředí. Poslední pointer v tomto poli musí být pointer NULL.

## Proměnné prostředí

Řetězce obsahující definice hodnot proměnných prostředí mají většinou tento tvar:

*název_proměnné=hodnota_proměnné*

Aby mohl nově spuštěný program parametry volání a proměnné prostředí použít, musí jeho zdrojový text začínat takto:

```
main(argc,argv,envp)
int argc;
char *argv[];
char *envp[];
{
. . . .
}
```

Druhá možnost, jak může být zdrojový text spouštěného programu napsán je tato:

```
extern char **environ;
main(argc,argv)
int argc;
char *argv[];
{
. . . .
}
```

Proměnná `environ` je globální proměnná obsahující pointer na pole pointerů `envp`. Hodnotu proměnné `environ` naplní standardní systémová startovací subrutina programu, která je k programu automaticky připojena linkage-editorem.

**Příklad 6:** Napište program, který vypíše proměnné prostředí**.** Použijte k tomu ukazatel na pole `envp` .

***Řešení***

```
#include <stdio.h>
main(argc,argv,envp)
int argc;
char *argv[];
chr *envp[];
{
     int i;
     for(i=0;envp[i] != (char *) 0; i++)
          printf(”%s\n”,envp[i]);
     exit(0);
}
```

**Příklad 7:** Napište program, který vypíše proměnné prostředí. Použijte k tomu globální proměnnou `environ` .

***Řešení***

```
#include <stdio.h>
extern char **environ;
main(argc,argv)
int argc;
char *argv[];
{
     int i;
     for (i=0; environ[i] != (char *) 0; i++)
          printf(" %s\n", environ[i]);
     exit(0);
}
```

Programy často používají proměnné prostředí k modifikaci své činnosti. Například editory podle hodnoty proměnné `TERM` posílají terminálu takové řídící znaky, které daný typ terminálu vyžaduje. Aby bylo uživatelům usnadněno zjištění hodnoty proměnné prostředí, existuje ve standardní knihovně jazyka C funkce ***getenv()***, která vrací pointer na hodnotu zadané proměnné.

Funkce *getenv()* se volá takto:

```
char *getenv(char *envvar);
ret=getenv(envvar);
```

Funkce *getenv()* vrací pointer na hodnotu proměnné `envvar`, pokud je tato proměnná definována. Jinak vrací pointer `NULL`.

**Příklad 8:** Použití funkce *getenv()* je zřejmé z následujícího jednoduchého programu, který vypíše hodnotu proměnné TERM.

```
#include <stdio.h>
extern char *getenv();
main()
{
     char *ptr;
     if ((ptr = getenv("TERM"))== (char *) 0)
          printf("promenna TERM neni definovana\n");
     else
          printf("TERM=%s\n",ptr);
     exit(0);
}
```

## Služba *execve()*

Po zavolání služby

```
execve(path,argv,envp);
```

operační systém provede následující akce:

1. Zjistí zda soubor identifikovaný cestou `path` obsahuje spustitelný program a zda proces má právo jej spustit. (To jest zkontroluje přístupová práva). Pokud má spouštěný soubor nastaven s-bit uživatele, změní efektivního individuálního vlastníka procesu (*EUID*) na individuálního vlastníka spouštěného souboru. Pokud má nastaven s-bit skupiny, změní efektivního skupinového vlastníka procesu (*EGID*) na skupinového vlastníka spouštěného souboru.
2. Uloží hodnoty parametrů volání a hodnoty proměnných prostředí na zásobník jádra.
3. Uvolní paměť, kterou proces obsazuje.
4. Pro nový proces vytvoří strukturu tabulky stránek.
5. Překopíruje hodnoty parametrů volání a proměnných prostředí na nový zásobník procesu.
6. Z hlavičky programu vezme startovací adresu a uloží ji do čítače instrukcí.

Po návratu z této služby systému je již proces řízen novým programem.

**Příklad 9:** Následující jednoduchý program po svém spuštění vytvoří proces, který požádá operační systém prostřednictvím volání služby *execve()* o záměnu svého strojového kódu za kód programu `/bin/date`.

```
#include <stdio.h>
main()
{
     char *(argv[2]);
     argv[0] = "date";
     argv[1] = ((char *)0);
     execve("/bin/date",argv,(char *)0);
}
```

## Spuštění scriptu

Služba systému *execve()*, respektive další funkce *exec()*, které jí používají, mohou spustit také script. Pokud je ve spouštěném souboru uložen script, první řádek scriptu musí začínat znaky `#!` . Za nimi musí následovat úplná cesta identifikující shell, který má script zpracovat. Například první řádek může začínat takto

```
#!/bin/csh
```

Služba systému *execve()* zajistí vytvoření nového procesu, který je řízen v první řádce scriptu specifikovaným shellem a zajistí, že shell začne script obsažený ve spouštěném souboru zpracovávat. Scripty, které začínají znaky `#!` lze proto považovat za spustitelné soubory a dvojici znaků `#!` za magické číslo.

# Příklady

**Příklad 10:** Pomocí služeb systému *alarm()* a *pause()* implementujte program **sleep**. Program **sleep** se spouští s jedním parametrem

**sleep** *sekundy*

Program nedělá nic jiného než to, že po svém spuštění zajistí, aby byl po dobu *sekundy* zablokován. Po odblokování program skončí. Pro odlišení nazvěte vaši implementaci programu **spi** .

***Řešení***

```
#include <stdio.h>
main(argc, argv)
     int argc;
     char *argv[];
{
     if (argc != 2){
          printf("spi musi mit dva parametry\n");
          exit(1);
          }
     alarm(atoi(argv[1]));
     pause();
}
```

Funkce *alarm(n)* požádá systém o zaslání časového signálu `SIGALRM` po *n* sekundách. Funkce *pause()* požádá operační systém o pozastavení.

**Příklad 11:** Někdy se zdá, že běh určitého programu nelze zrušit. Pokud procesu, který program řídí, zašleme signál, proces sice skončí, ale okamžitě vznikne jiný proces (tj. proces s jiným *PID*), který je řízen stejným programem. To nastane tehdy, když rodič tohoto procesu čeká na jeho skončení a po jeho skončení znovu program spustí. Proto pokud je třeba zamezit tomu, aby byl program neustále spouštěn, je nutné zrušit rodičovský proces.

Napište program **stalespi**, který bude program **sleep 60** spouštět tak dlouho, dokud neskončí normálním způsobem, tj. po uplynutí 60 sekund. To jest pokud bude program **sleep 60** zasláním signálu předčasně ukončen, program **stalespi** jej opět spustí.

***Řešení***

```
#include <stdio.h>
main()
{
     int status;
     while(1){
          if (fork() == 0){
               execl("/usr/bin/sleep","sleep","60",NULL);
               printf("chyba: sleep nelze spustit\n");
               kill(getppid(),9);
               exit(1);

          }
          wait(&status);
          if (!status){  /* spanek nebyl neprerusen */
               printf("souvisly spanek ukoncen\n");
               exit(0);
          }
     }
}
```

**Příklad 12:** Napište program, který vypíše spuštěním programu **ls -al** adresář zadaný hodnotou proměnné prostředí `ADR`. Pokud proměnná `ADR` nebude definována, vypíše obsah aktuálního adresáře.

***Řešení:***

```
#include <stdio.h>
#include <stdlib.h>
main()
{
     int pid,status;
     char *padr;
          if ((pid = fork()) == 0){
               if ((padr=getenv("ADR")) == NULL)
                    padr=".";
               execl("/bin/ls","ls","-al",padr,NULL);
               exit(1);
          }
          wait(&status);
          if(status){
               printf("program ls nelze spustit\n");
               exit(1);
          }
          else
               printf("program ls skoncil O.K. \n");
               exit(0);
}
```