ZARZĄDZANIE PROCESAMI

Rozumienie pojęcia proces w informatyce jest, w istocie rzeczy, bliskie jego zwyczajowemu pojmowaniu, czyli jako pewnej sekwencji zmian lub czy zdarzeń zachodzących wg ustalonego schematu (choć niekoniecznie explicite).

> PROCESem nazywamy sekwencją zmian stanu systemu komputerowego, odbywającą się według sformalizowanego zapisu ich algorytmu w postaci programu, stanowiąc instancję tego ostatniego.

Każdy proces może utworzyć jeden lub więcej procesów potomnych (*child*)
stając wobec nich procesem macierzystym (*parent*).
W chwili tworzenia procesu system operacyjny alokuje, celem jego
reprezentacji, strukturę danych w postaci *PCB* (*Process Control Block*)..

W dalszej części niniejszych materiałów będziemy się generalnie odnosić do systemów operacyjnych wyposażonych w *API* standardu *Portable Operating System Interface [for UNIX]* (*POSIX**), chyba że w treści zostanie zaznaczone wyraźnie coś innego.

**POSIX* stanowi standard uniwersalnego *API* systemu operacyjnego ustanowionego normą *IEEE 1003* oraz *ISO/IEC 9945*. Obejmuje specyfikację podsystemów interface użytkownika, obsługi procesów i wątków, czasu rzeczywistego, sieciowego i bezpieczeństwa. Implementowany jest w systemach rodzin takich jak: *UNIX*, *LINX*, *AIX*, *HP-UX*, *BSD*, *IRIX*, *LynxOS*, *Mac OS*, *QNX*, *RTEMS*, *Solaris*. Istnieje także *SFU*, czyli *Microsoft Windows Services for UNIX*, dostępny jako bezpłatne uzupełnienie systemów dla *MsWindows 2000/Server* oraz *XP*, w aktualnej wersji *3.5* zajmuje 217.6 MB.
http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=896c9688-601b-44f1-81a4-02878ff11778

ZARZĄDZANIE PROCESAMI

Każdy system operacyjny oferuje usługi umożliwiające pobranie informacji o aktywności i stanie bieżących procesów. W systemach rodziny ***POSIX*** służą temu m.in. zestaw poleceń konsoli:

`ps top watch time`

`ps [option] -o [format]`

Na liście opcji mamy zasadniczo dwie grupy:
option, co należy wyświetlić a w szczególności

`-e`	wszystko
`user[name]`	użytkownika **`name`**
`group[name]`	grupy **`name`**
`tty [n]`	na terminalu **`n`**

format jak sformatować wyjście, w szczególności

`pid`	identykator PID
`ppid`	identykator parent danego
`tname`	terminal
`state`	stan aktualny

PRZYKŁAD

`ps group users tty 3 -o pid,cmd`

wyświetli dla grupy **`users`** z terminala **`3`** informację o jej procesach podając ***PID*** oraz komendę jaka uaktywniła proces.

ZARZĄDZANIE PROCESAMI

Listing procesów, w postaci struktury drzewiastej, poczynając od procesu init albo pid

pstree [options] [pid|user]

gdzie jako options można użyć w szczególności

-a argumenty linii komend aktywującej proces

user użytkownik którego procesy listować

-p pokazuj *PID* poszczególnych

PRZYKŁAD Tekst niniejszy przygotowywany był w środowisko *LINUX/X Window System*, pracującego pod kontrolą managera okien *KDE*. Inicjującym ten go jest proces ***kdeinit***.

```
$> ps -e |grep kdeinit
 3728 ?        00:00:00 start_kdeinit
 3729 ?        00:00:00 kdeinit
```

czyli, gdyby chcieć prześledzić drzewo procesów potomnych to można użyć komendy

```
$> pstree  3729
kdeinit-+-acroread---{acroread}
        |-firefox---run-mozilla.sh---firefox-bin---5*[{firefox-bin}]
        |-klauncher
        |-2*[konqueror]
        |-konsole---bash---pstree
        |-
```

ZARZĄDZANIE PROCESAMI

Ponieważ informacja odnośnie identyfikacji procesów ma znaczenie kluczowe w kontekście zarządzania nimi, tak i każde API systemowe daje możliwość pobrania tego rodzaju informacji.

Synopsis

```
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Return

PID, *PPID* procesu

Erorrs

-1

PRZYKŁAD

```
#include<stdio.h>
#include<unistd.h>
int main( void )
{
  printf("Current ID\t%d\n",(int)getpid() );
  printf("Parent  ID\t%d\n",(int)getppid());

  return 0;
}
```

Kompilacja (i konsolidacja)

```
$>gcc pid.c -o pid
```

Wykonanie (z bieżącego katalogu)

```
$>./pid
    Current ID      12087
    Parent  ID      10788
```

Zauważmy że *PPID* jest **10788**, bo

```
$>ps
  PID TTY          TIME CMD
10788 pts/1    00:00:00 bash
12091 pts/1    00:00:00 ps
```

dla naszego programu procesem parent jest powłoka *bash*.

ZARZĄDZANIE PROCESAMI

Nieco inne możliwości śledzenia procesów daje komanda

top

szczególnie ważna w przypadku konieczności monitorowania pracy komputera jako węzła *cluster*'a.

PRZYKŁAD

```
$>top
top - 10:00:47 up 5:34, 3 users, load average: 0.40, 0.29, 0.21
Tasks: 115 total,3 running,112 sleeping, 0 stopped, 0 zombie
Cpu(s): 3.6%us, 1.0%sy,0.0%ni,95.4%id, 0.0%wa,0.0%hi,0.0%si,0.0%st
Mem:   1800860k total,  1738532k used,    62328k free,    82320k buffers
Swap:  2104472k total,       20k used,  2104452k free,   887388k cached
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3345 root      15   0  387m 154m  17m S  2.7  8.8  12:27.57 Xorg
 3748 kmirota   15   0  280m  25m  16m S  0.7  1.5   1:37.95 kicker
 3817 kmirota   15   0 44792 4508 3700 S  0.3  0.3   0:29.89 conky
10787 kmirota   15   0  148m  18m  13m R  0.3  1.1   0:04.38 konsole
```

PID *ID* procesu
USER właściciel
PR priorytet
NI zmiana priorytetu (*nice*)
VIRT pamięć wirtualna
VIRT=SWAP+RES, RES=CODE+DATA
SHR pamięć (współ)dzielona
S status (**D**-uninterruptible sleep, **R**-running, **S**-sleeping, **T**-traced or stopped, **Z**-zombie)
%CPU zużycie czasu procesora
%RES zużycie fizycznej pamięci przez RES
TIME czas pracy procesu
COMMAND komenda uruchomienia

ZARZĄDZANIE PROCESAMI

Mówiąc o usługach systemowych monitorujących, zwłaszcza w kontekście aplikacji clustr'owych warto jeszcze wspomnieć o komendach:

- informacja o zajętości pamięci

  `free [-b|-k|-m] [-s delay ] [-t]`

  `-b|-k|-m` wartości w *B*, *KB*, *MB*

  `-s delay` wyświetla informację co **`delay`** sekund (np. **`free -s 5`**, co 5 sekund)

  `-t` wyświetla podsumowanie *total*

```
$> free -t -m
             total       used       free     shared    buffers     cached
Mem:          1758       1719         38          0         84        880
-/+ buffers/cache:        755       1003
Swap:         2055          0       2055
Total:        3813       1719       2093
```

- statystyki użycia pamięci *VIRTUAL*

  `vmstat [-s] [-S] [-n] [delay [count]]`

  `delay [count]` wyświetla co **`delay`** sekund count krotnie (np. **`vmstat 1 5`**)

  `-n` przy opcji **`delay`** nie jest powtarzany nagłówek

  `-s` wyświetla w podsumowanie, w układzie wierszowym

  `-S` jednostka **`K|M`** czyli *KB* albo *MB*

- statystyki odnośnie czasu wykonania podanego **`cmd`**

  `time cmd`

ZARZĄDZANIE PROCESAMI

W najprostszym, ale i najczęściej występującym przypadku, proces powstaje jako podproces powłoki systemowej *Bourne shell* (**/bin/sh**). Identyczny mechanizm osiągalny jest także z kodu programu za pośrednictwem wywołania funkcji **system()**.

Synopsis

```
#include<stdlib.h>
int system(const char *cmd);
const char *cmd; ...komenda do wykonania
```

Return

kod powrotu procesu **cmd**

Erorrs

-1

Jest to jednak sposób bardzo mało efektywny i rzadko kiedy można znaleźć dla niego uzasadnienie.

PRZYKŁAD

```
//sys.c
#include<stdlib.h>
int main( void )
{
    system( "ps -o pid,ppid,cmd" );
    return 0;
}
```

```
>./sys
  PID  PPID CMD
 4537  4536 /bin/bash
 4942  4537 ./sys
 4943  4942 ps -o pid,ppid,cmd
```

ZARZĄDZANIE PROCESAMI

Tworzeniu procesów służy również rodzina funkcji **exec**()**, przy czym potomny zastępuje macierzysty, dziedzicząc po nim także i *PID*.

Synopsis

```
#include <unistd.h>

int execl(char const *cmd,char const *arg,...);
int execle(char const *cmd,char const *arg,...,char const *const *envp);
int execlp(char const *cmd, char const *arg, ...);
int execlpe(char const *cmd, char const *arg, ...);
int execv(char const *cmd, char const * const * argv);
int execve(char const *cmd,char const * const *argv,char const *const *env );
int execvp(char const *cmd, char const * const *argv);
int execvpe(char const *cmd,char const * const *argv,char const *const *env );
```

Return

de facto funkcje te nie powinny zwracać wartości, chyba że wystąpi błąd

Erorrs

-1

Różnice między funkcjami odzwierciedlają litery w nazwie następujące po słowie "*exec*", i tak pozycja:

- *5* określa sposób przekazywania argumentów: **l** (*command line*), **v** (*tablica wskazań*)
- dalsze są uzależnione od przekazywania zmiennych środowiskowych: **p** (szukać wg **PATH**), **e** (jako ostatni argument).

Zarządzanie Procesami

Załóżmy że kod procesu potomnego przedstawia się w następujący sposób:

```
//child.c
#include <stdio.h>
int main( int argc, char **argv )
{
   int i;
   for( i=0;i<argc;i++ ){ printf( "%4d:... %s\n",i,argv[i] ); }
   return 0;
}
```

Zatem wypisuje on na konsoli on argumenty swego wywołania.

Wywołanie bezparametryczne da wynik

```
$>./child
  0:... ./child
```

zaś wywołanie z listą 3 parametrów "**pierwszy**", "**drugi**", "**trzeci**"

```
$> ./child pierwszy drugi trzeci
   0:... ./child
   1:... pierwszy
   2:... drugi
   3:... trzeci
```

ZARZĄDZANIE PROCESAMI

Kod programu child wykorzystamy do utworzenia procesu potomnego z użyciem funkcji **execl()**, a więc wariantu w przypadku którego zakłada się, że informacje odnośnie środowiska (ścieżki poszukiwania) przekazane będą wprost w pierwszym argumencie wywołania **cmd**. Pozostałe argumenty stanowić będą – załóżmy jak wcześniej - "**pierwszy**", "**drugi**", "**trzeci**". Ponieważ lista argumentów może mieć tutaj zmienną długość, więc trzeba oznaczyć koniec – tutaj odbywa się to za pomocą znaku **NULL** (**'\0'**).

```
//parent.c
#include<unistd.h>
#include<stdio.h>
int main( void )
{
    char *arg1="pierwszy", *arg2="drugi",*arg3="trzeci";
    printf( "- wywołanie (samobójcze) potomka ----------\n" );
    execl( "./child",arg1,arg2,arg3,'\0' );
    return 0;
}
```

Efektem wywołanie (zakładamy że w katalogu bieżącym jest child)

```
> ./parent
- wywołanie (samobójcze) potomka ----------
   0:... pierwszy
   1:... drugi
   2:... trzeci
```

Zarządzanie Procesami

Trudność w przypadku funkcji

`execl*()`

polega na tym, iż konieczna jest znajomość listy parametrów wywołanie. Jeżeli – w momencie powstawania programu **`(!)`** - nie są znane lub ich ilość jest zmienna, wówczas nie można użyć tego typu funkcji. Wówczas możliwe jest zastosowanie wyłącznie funkcji grupy

`execv*()`

którego argumentem jest tablica wskazań do tablic znakowych.

Zmodyfikujmy w takim razie kod źródłowy parent

```
#include<unistd.h>
#include<stdio.h>
int main( void )
{
   char *arg[4];
   arg[0]="pierwszy"; arg[1]="drugi"; arg[2]="trzeci"; arg[3]='\0';
   printf( "- wywołanie (samobójcze) potomka ----------\n" );
   execv( "./child",arg );
   return 0;
}
```

Oczywiście efekt finalny będzie identyczny jak wcześniej.

Zauważmy, że tablicę przekazywaną do **`execv*()`** można utworzyć dynamicznie, o dowolnym rozmiarze i zawartości.

ZARZĄDZANIE PROCESAMI

O ile wywołanie funkcji **system()** czy **exec*()** mogą wygenerować proces potomny, to trudno byłoby między nimi osiągnąć jakąś koordynację działań.
Jeżeli między procesami występują jakieś formy uzależnienia, to w systemach rodziny *POSIX* – celem utworzenia procesu potomnego wykorzystuje się funkcję **fork()**.

Synopsis

```
#include <unistd.h>
int fork( void );
```

Return

PID w *parent*

Erorrs

```
-1
```

Ponieważ proces potomny, jest z punktu widzenia systemu nowym i niezależnym procesem ale jego kod jest dokładna kopią procesu parent (dlatego na liście parametrów **fork()** mamy **void** !), więc zwykle stosuje się – celem zróżnicowania działania *parent* i *child* – następującą konstrukcję jak niżej.

```
switch( fork() )
{
    case -1:
    //...kod dla procesu parent, w przypadku niepowodzenia
    break;
    case 0:
    //...kod dla procesu child
    break;
  default:
    //...kod dla procesu parent
}
```

ZARZĄDZANIE PROCESAMI

Zazwyczaj zachodzić będzie potrzeba zsynchronizowania działań ***parent*** i ***child*** (jeżeli parent uruchomił ***child***, to przypuszczalnie ***child*** powinien coś dla niego wykonać).
Takiej synchronizacji służy funkcja **`wait()`**.

Synopsis

```
#include <unistd.h>
int wait( int *status );
int *status; ...kod powrotu z child
```

Return

PID child

Erorrs

`-1`

Funkcja ma charakter blokujący: ***parent*** będzie czekał na ***child*** póki nie skończy a jeżeli skończy zanim pojawi się wywołanie **`wait()`**, to nie spowoduje to zatrzymania parent (dostanie on – od systemu – ***PID*** zakończonego wcześniej procesu ***child***).

Równocześnie – zwłaszcza w przypadku – potomka mogą zdarzyć się sytuacje kiedy jego wykonanie będzie musiało być natychmiastowo zakończone. Można tego dokonać za pomocą rodziny funkcji **`exit()`**.

Synopsis

```
#include <unistd.h>
void exit( int status );
int status; ...kod powrotu jaki będzie przekazany do parent
```

zaś poprzez wartość parametru formalnego można poinformować proces nadrzędny o przyczynie.

Zarządzanie Procesami

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

int main( void )
{
   int status;

   switch( fork() )
   {
      case -1: //... kod na wypadek błędu dla PARENT
         printf( "<parent> oj niedobrze, niedobrze\n" );
         break;
      case  0: //...kod dla CHILD
         printf( "<child> pozdrowienia od potomka\n" );
         break;
      default: //...kod dla parent
         printf( "<parent> ja jestem PARENT\n" );
         wait( &status );
         printf("<parent> potomek skończył, zwrócił :%d\n",status);
   }

   return 0;
}
```

ZARZĄDZANIE PROCESAMI

Oczywiście kod zwykle dla parent i child będziemy bardziej różnicować.

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

int main( void )
{
   int status;
   switch( fork() )
   {
      case -1: printf( "...błąd uruchomienia procesu potomnego\n" ); break;
      case  0:
         printf( "*** URUCHOMIENIE PROCESU POTOMNEGO ***\n" );
         execl( "/usr/bin/free","-m",'\0' );
         //... w gruncie rzeczy nie może się tu znaleźć, ale
         printf( "...błąd w procesie potomnym\n" );  exit( -1 );
         break; //...oczywiście break w tym miejscu nie ma znaczenia
      default: //...kod dla parent
         wait( &status );
         if( !status ){printf( "potomek zakończył działanie (prawidłowo)\n" )}
         else{ printf( "... coś nie tak z potomkiem\n" ); }
   }
   return 0;
}
```

Zauważmy że child wyzwala – samobójczo – komendę systemową ps.

ZARZĄDZANIE PROCESAMI

Efektem wykonania kodu będzie

```
*** URUCHOMIENIE PROCESU POTOMNEGO ***
             total       used       free     shared    buffers     cached
Mem:       1800860    1702540      98320          0      70124    1116780
-/+ buffers/cache:     515636    1285224
Swap:      2104472          0    2104472
potomek zakończył działanie (prawidłowo)
```

Gdyby zmodyfikować kod następująco (w funkcji **execl()**)

```
switch( fork() )
{
  case -1: printf( "...błąd uruchomienia procesu potomnego\n" ); break;
  case  0:
     printf( "*** URUCHOMIENIE PROCESU POTOMNEGO ***\n" );
     execl( "free","-m",'\0' ); //... tutaj zmiana !!!
     printf( "...błąd w procesie potomnym\n" );  exit( -1 );
     break;
  default:
     wait( &status );
     if( !status ){printf( "potomek zakończył działanie (prawidłowo)\n" )}
     else{ printf( "...coś nie tak z potomkiem\n" ); }
}
```

to rezultat będzie nieco inny

```
*** URUCHOMIENIE PROCESU POTOMNEGO ***
...błąd w procesie potomnym
...coś nie tak z potomkiem
```

ZARZĄDZANIE PROCESAMI

Może się zdarzyć, iż z jakiegoś powodu pewien proces będzie chciał natychmiastowo zakończyć działanie. Służy temu funkcja **exit()**.

Synopsis

```
#include <stdlib.h>
void exit( int status );
int status; ...kod powrotu jaki będzie zwrócony przez proces
```

Funkcja kończy działanie procesu wywołującego tę funkcję i przekazanie status , w szczególności jednej z predefiniowanych stałych symbolicznych (w **stdlib.h**)

EXIT_SUCCESS
EXIT_FAILURE

do procesu macierzystego. Jeżeli dany proces posiadał będzie potomków, to nie są one zakończone ale procesem parent dla nich staje się *INIT* (czyli *PPID* będzie **1**).

W tym samym pliku nagłówkowym (**stdlib.h**) zawarta jest deklaracja funkcji **atexit()**, umożliwiającej zarejestrowanie akcji dla wywołania **exit()** (lub **return**);

Synopsis

```
#include <unistd.h>
int atexit( void (*func)( void ) )
```

Return

0, ... *jeżeli sukces*

Erorrs

non-zero

Zarządzanie Procesami

Na koniec prosty przykład użycia funkcji exit() oraz atexit().

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main( void )
{
   int status;
   void bye( void );

   (void) atexit( bye );

   switch( fork() )
   {
      case -1: printf( "...błąd uruchomienia potomnego\n" ); break;
      case  0: exit( EXIT_SUCCESS ); break;
      default:
        wait( &status );
        if(status){ printf("...bład powrotu\n"); exit(EXIT_FAILURE); }
   }
   return EXIT_SUCCESS;
}

void bye( void )
{ printf( "...i to wszystko w [%d]\n",(int)getpid() ); return; }
```