2. Zarządzanie procesami

W systemie Linux wszelkie aktywności wykonywane są przez procesy które operują na pamięci operacyjnej, plikach i urządzeniach. Procesy tworzą drzewo którego korzeniem jest proces init. Proces ten tworzony jest przy starcie systemu. Czyta on pliki konfiguracyjne (zawarte w pliku /etc/init.d) i uruchamia dalsze procesy. Drzewo procesów otrzymać można pisząc ma konsoli polecenie pstree.

```
init-+-NetworkManager-+-dhclient
     |                `-{NetworkManager}
     |-SystemToolsBack
 ...
     |-gnome-terminal-+-bash---su---bash---su---bash---pstree
     |                |-bash---mc---bash
     |                |-bash
     |                |-gnome-pty-helpe
     |                `-{gnome-terminal}
     |-gnome-vfs-daemo
...
```

Rys. 2-1 Fragment drzewa procesów systemu Linux Debian uzyskanych przy pomocy polecenia **`pstree`**

2.1 Funkcje zarządzania procesami

Administrowanie procesami obejmuje następujące czynności:

- Tworzenie procesu.
- Kończenie procesu.
- Testowanie atrybutów procesu.
- Ustawianie atrybutów procesu.
- Synchronizacja zakończenia procesu potomnego z macierzystym.

Utworzenie kopii procesu bieżącego.	**`fork()`**
Zastąpienie procesu bieżącego innym procesem	**`exec*()`**
Wykonanie programu lub skryptu.	**`system()`**
Zakończenie procesu	**`exit()`**
Czekanie na zakończenie procesu	**`wait(), waitpid()`**

Rysunek 2-1 Ważniejsze funkcje zarządzania procesami

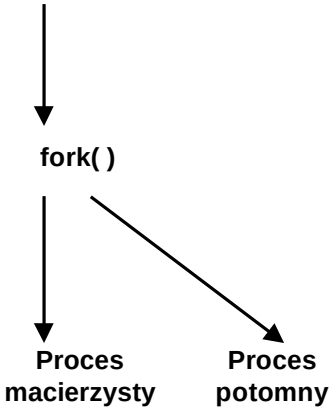
2.2 Tworzenie kopii procesu bieżącego – funkcja fork

Procesy tworzone są za pomocą funkcji **`fork()`**. Funkcja **`fork`** posiada następujący prototyp. Funkcja tworzy kopię procesu bieżącego czyli tego procesu który wykonuje funkcję **`fork( )`**.

```
pid_t fork(void)
```

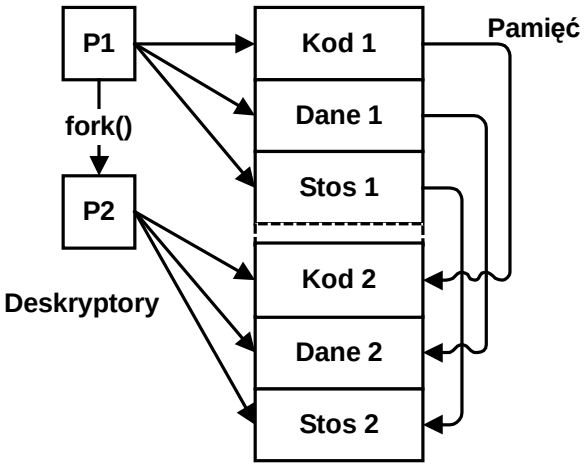
Funkcja **`fork ()`** zwraca:

`0`	w procesie potomnym
`> 0`	w procesie macierzystym zwracany jest PID procesu potomnego
`- 1`	błąd



Rys. 2-1 Działanie funkcji fork – procesy macierzysty i potomny wykonywane są współbieżnie.

Funkcja **`fork`** tworzy deskryptor nowego procesu oraz kopię segmentu danych i stosu procesu macierzystego.



Rys. 2-2 Proces P1 wykonał funkcję **`fork`** i utworzył proces P2. Procesy P1 i P2 posiadają własne segmenty danych i stosu.

Uwagi:

1. Wartości zmiennych w procesie potomnym są takie jak w procesie macierzystym bezpośrednio przed wykonaniem funkcji **`fork`**.
2. Modyfikacje zmiennych danych dokonywane w procesie macierzystym nie są widoczne w procesie potomnym (i odwrotnie) gdyż każdy z procesów posiada własną kopię segmentu danych.
3. Proces potomny zachowuje otwarte pliki procesu macierzystego ale tworzy własne kopie ich deskryptorów.
4. Otwarte kolejki komunikatów są dziedziczone
5. Otwarte strumienie kartotek (funkcja: **`opendir`**) są dziedziczone.

Utworzony proces potomny różni się od macierzystego pod następującymi względami:

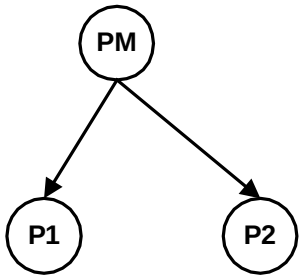
1. Ma inny PID.
2. Ma inny PID procesu macierzystego (ang. *parent PID*).
3. Blokady obszarów pamięci nie są dziedziczone
4. Liczniki zużycia czasu procesora są zerowane
5. Zbiór nie obsłużonych sygnałów jest pusty
6. Nie jest dziedziczony stan liczników semaforów
7. Nie są dziedziczone blokady plików
8. Nie są dziedziczone czasomierze

```
#include <stdio.h>
#include <process.h>
#include <unistd.h>
void main(void){
  int pid,status;
  if((pid = fork()) == 0) { /* Proces potomny ---*/
      printf(" Potomny = %d \n",getpid());
      sleep(30);
      exit(0);
  }
 /* Proces macierzysty */
 printf("Macierzysty = %d \n",getpid());
 pid = wait(&status);
 printf("Proces %d zakończony status:
%d\n",pid,status);
}
```

Program 3-1 Podstawowy schemat wykorzystania funkcji fork

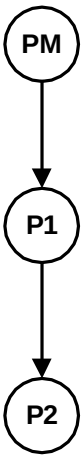
Używając funkcji **`fork`** można tworzyć rozmaite konfiguracje procesów co pokazano poniżej.

```
Kod wspolny
if(fork() == 0) {
    Kod procesu potomnego P1
    exit(0);
}
if(fork() == 0) {
    Kod procesu potomnego P2
    exit(0);
}
Kod procesu macierzystego PM
```



Rys. 3-1 Schemat użycia funkcji fork do utworzenia dwu procesów potomnych P1 i P2. Procesy na usytuowane na jednakowym poziomie hierarchii.

```
Kod wspolny
if(fork() == 0) {
    if(fork() == 0) {
        Kod procesu potomnego P2
    } else {
        Kod procesu potomnego P1
    }
}
Kod procesu macierzystego PM
```



Rys. 3-2 Schemat użycia funkcji fork do utworzenia dwu procesów potomnych P1 i P2. Proces P2 jest procesem potomnym procesu P1.

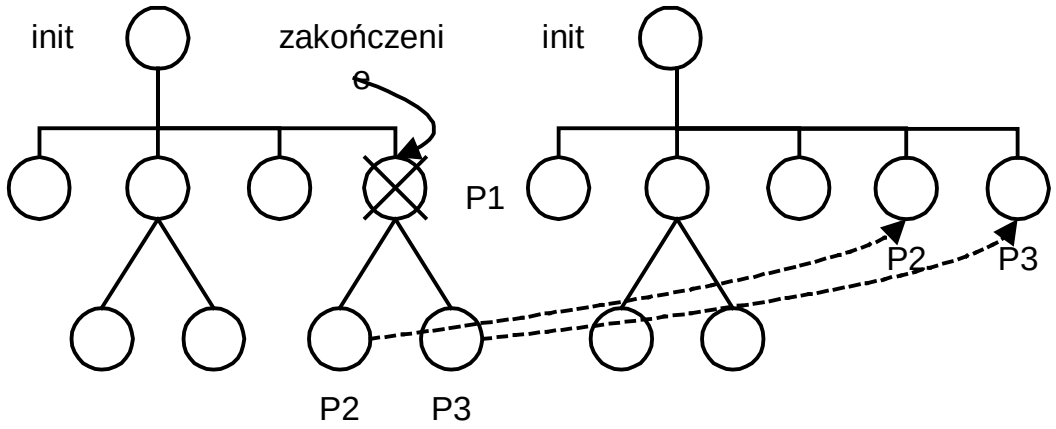
2.3 Obsługa zakończenia procesów

Kończenie procesu

Chcąc prawidłowo zakończyć proces, powinno się wykonać następujące czynności:

1. Zakończyć scenariusze komunikacyjne z innymi procesami.
2. Zwolnić zajmowane zasoby.
3. Zaczekać na zakończenie procesów potomnych.

☞ Przed zakończeniem procesu należy zwolnić zajęte przez ten proces zasoby i zakończyć rozpoczęte z innymi procesami scenariusze komunikacyjne i synchronizacyjne.



Rys. 3-3 Procesy P2 i P3 adoptowane przez proces init

☞ Nie należy kończyć procesu który posiada nie zakończone procesy potomne.

Inicjowanie zakończenia procesu

Zakończenie się procesu następuje w podanych niżej przypadkach:

1. W dowolnym miejscu kodu procesu wykonana zostanie funkcja **`exit`**.
2. Funkcja **`main`** procesu wykona instrukcję **`return`**.
3. Funkcja main procesu wykona ostatnią instrukcję kodu.
4. Proces zostanie zakończony przez system operacyjny lub inny proces.

Preferowanym sposobem zakończenia procesu jest wykonanie funkcji exit której prototyp podany został poniżej.

```
#include <stdlib.h>

void exit(int x)
```

Wykonanie funkcji **`exit(x)`** powoduje zakończenie się procesu bieżącego. Wszystkie zasoby zajmowane przez proces z wyjątkiem jego deskryptora są zwalniane. Dodatkowo wykonywane są następujące akcje:

1. Otwarte pliki i strumienie są zamykane.
2. Najmłodszy bajt (8 bitów) z kodu powrotu x jest przekazywane do zmiennej status odczytywanej przez funkcję **`wait()`** wykonaną w procesie macierzystym. Kod powrotu przechowywany jest w deskryptorze procesu.
3. Gdy proces macierzysty wykonał wcześniej funkcję **`wait()`** albo **`waitpid()`** i jest zablokowany, następuje jego odblokowanie i usunięcie deskryptora.
4. Gdy proces macierzysty nie wykonał wcześniej funkcję **`wait()`** albo **`waitpid()`** kod powrotu przechowywany jest w deskryptorze procesu a proces przechodzi do stanu „zoombie”.
5. Do procesu macierzystego wysyłany jest sygnał SIGCHLD.

Uwaga!

W systemie istnieje także funkcja **_exit(int x)** różniąca się tym że nie można w niej zarejestrować funkcji kończącej **on_exit, atexit**.

Rejestracja funkcji kończącej

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

Gdy program zakończy się normalnie wykonana zostanie zarejestrowana funkcja kończąca. Można zarejestrować wiele funkcji kończących, wykonywane są w porządku odwrotnym do rejestracji.

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

 void  bye(void)  {
    printf("Handler - zadzialalem!\n");
 }

int  main(void)  {
  int res,i;
  printf("Start\n");
  // Instalacja handlera zakonczenia --
  res = atexit(bye);
  if (res != 0) {
      perror("atexit");
      exit(EXIT_FAILURE);
  }
  printf("Koniec\n");
  exit(EXIT_SUCCESS);
}
```

Przykład 3-1 Działanie funkcji kończącej

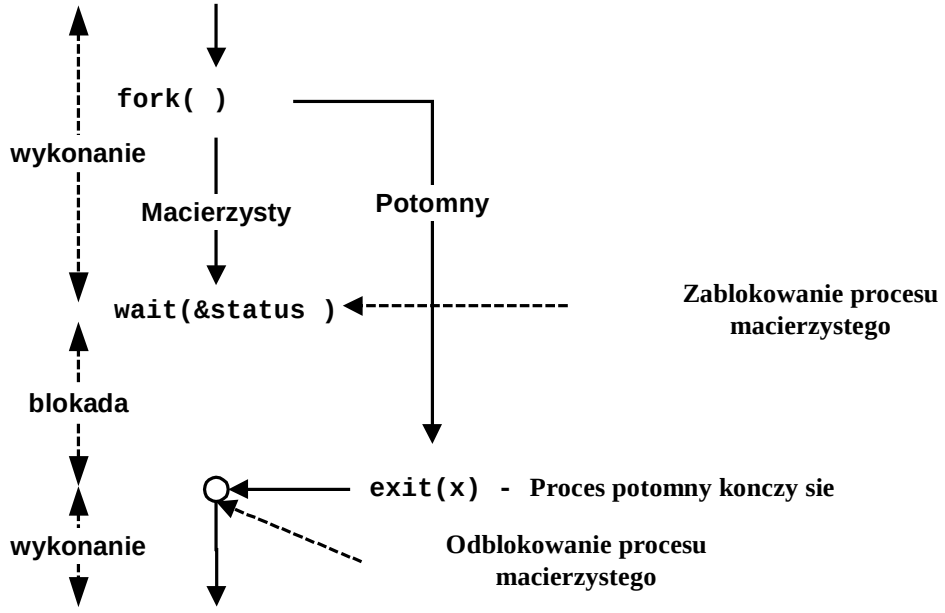
Synchronizacja zakończenia procesów

Funkcja **`wait()`** powoduje że proces macierzysty będzie czekał na zakończenie procesu potomnego. Prototyp funkcji **`wait()`** jest następujący:

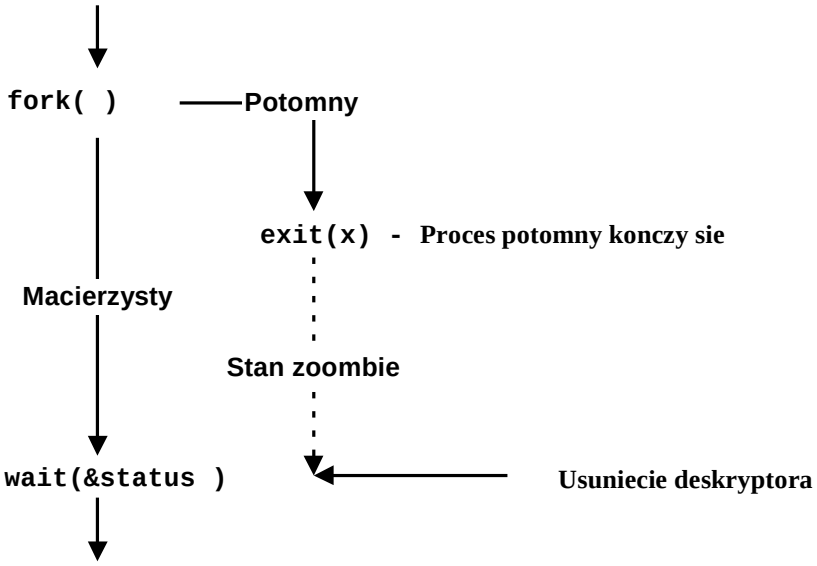
```
pid_t wait(int * status)
```

Działanie funkcji **`wait`** jest następujące:

1. Gdy proces potomny nie zakończył się funkcja **`wait`** powoduje zablokowanie procesu macierzystego aż do zakończenia się procesu potomnego. Gdy ten się zakończy zwracany jest jego PID oraz status.
2. Gdy proces potomny zakończył się zanim wykonano funkcję **`wait`** nie występuje blokada procesu macierzystego. Funkcja zwraca PID zakończonego procesu oraz jego status.
3. Gdy brak jakichkolwiek procesów potomnych funkcja **`wait`** zwraca –1,



Rys. 3-4 Proces macierzysty czeka na zakończenie się procesu potomnego.



Rys. 3-5 Proces potomny kończy się wcześniej niż proces macierzysty

```
#include <stdio.h>
#include <process.h>
int main(int argc, char * argv[]){
  int pid,status;
  if((pid = fork()) == 0) { // Proces potomny  ---
     printf(" Proces potomny PID:  %d \n", getpid());
     sleep(10);
     _exit(0);
  }
  // Proces macierzysty -----------------
  pid = wait(&status); // Czekamy na proces potomny
  printf("Proces %d zakończony,status%d\n",
  pid,WEXITSTATUS(status));
  return 0;
}
```

Program 3-2 Schemat wykorzystania funkcji **fork**, **wait**, **exit**.

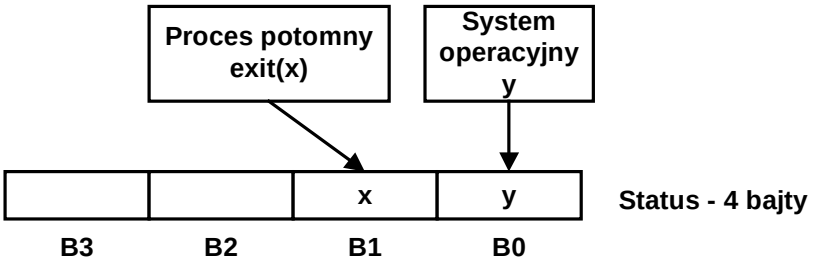
Testowanie statusu zakończonego procesu.

Status zakończonego procesu udostępniany jest przez funkcję **`wait`**

```
pid = wait(&status)
```

Wartość zmiennej status zależy od:

1. Systemu operacyjnego który umieszcza tam informacje o przyczynach i sposobie zakończenie procesu.
2. Zakończonego procesu potomnego który umieszcza tam wartość kodu powrotu – jest to parametr funkcji **`exit`**.



Rys. 3-6 Przekazywanie statusu do procesu potomnego

Znaczenie parametrów x , y jest następujące:
y – informacja o sposobie i przyczynach zakończenia procesu potomnego.
x – parametr x (nazywany kodem powrotu) funkcji **`exit(x)`** wykonanej w procesie potomnym.

Makro	Znaczenie
`WIFEXITED(status)`	zwraca > 0 gdy proces potomny był zakończony normalnie
`WEXITSTATUS(status)`	zwraca kod powrotu y przekazany przez funkcję **`exit(y)`** z procesu potomnego
`WIFSIGNALED(status)`	zwraca > 0 gdy proces potomny był zakończony przez nie obsłużony sygnał
`WTERMSIG(status)`	zwraca numer sygnału gdy proces był zakończony przez sygnał

Tab. 3-1 Makra do testowanie statusu zakończonego procesu potomnego.

```
int pid, status;
…
pid = wait(&status); // Czekamy na proces potomny
if(WEXITED(status))
  printf("%d zakończ, kod %d\n",pid,
  WEXITSTATUS(status));
if(WESIGNALED(status))
  printf("Pro. %d zakończ.sygn:%d\n"  ,pid,
  WTERMSIG(status));
…
```

Program 3-3 Testowanie przyczyny zakończenia procesu

Funkcja **waitpid()** pozwala czekać na konkretny proces

```
pid_t waitpid(pid_t pid, int * status, int opcje)
```

pid >0 – PID procesu na którego zakończenie czekamy,
=0 – czekamy na procesy z tej samej grupy co proces bieżący,
<0 – czekamy na procesy z grupy której numer jest wartością bezwzględną parametru.

status Status kończonego procesu.

opcje 0 lub specyfikacja typu procesu na który czekamy

Funkcja zwraca:

>0 PID zakończonego procesu,

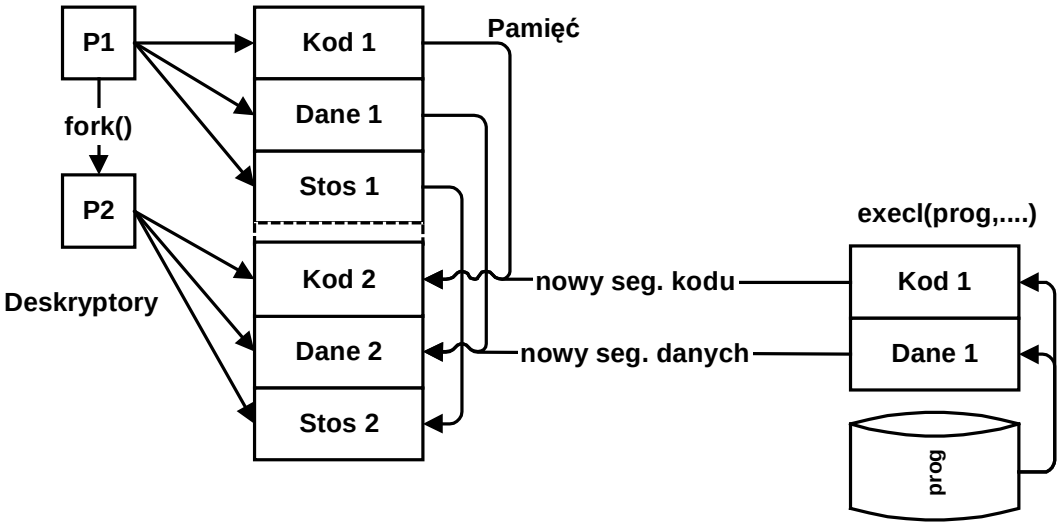
-1 gdy brak jest procesów potomnych.

W odróżnieniu od funkcji **wait()** która odblokuje proces bieżący przy zakończeniu dowolnego procesu potomnego funkcja **waitpid()** odblokuje się gdy zakończy się proces podany jako parametr lub jeden z procesów z podanej grupy procesów.

2.4 Przekształcenie procesu bieżącego w inny proces

Rodzina funkcji **exec** ta zawiera funkcje: **execl, execv, execle, execlp, execvp**

Każda funkcja z rodziny **exec** przekształca bieżący proces w nowy proces tworzony z pliku wykonywalnego będącego jednym z parametrów funkcji exec.



Rys. 3-7 Działanie funkcji exec

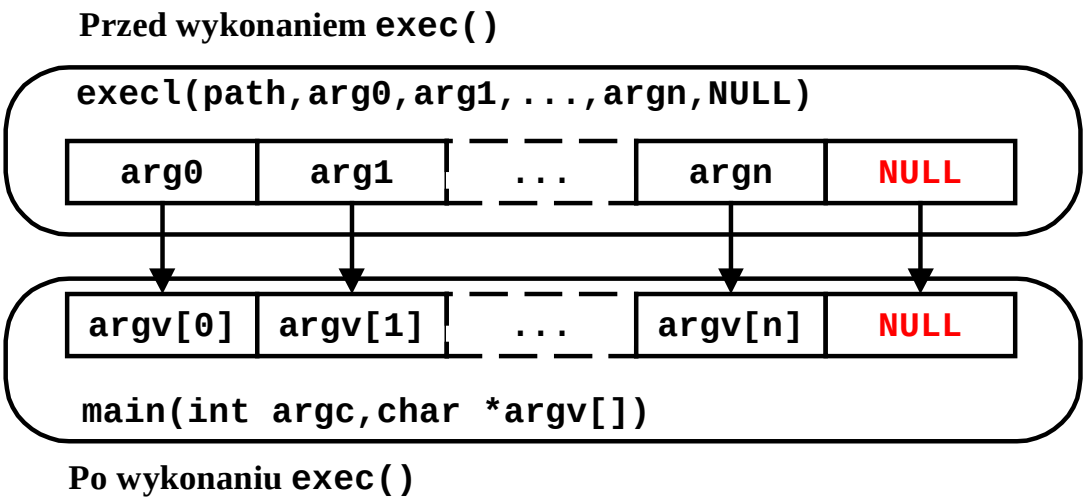
```
pid_t execl(char * path, arg0, arg1,...,argN,NULL)
pid_t execv(char * path, char * argv[])
```

path	Ścieżka z nazwą pliku wykonywalnego.
	Argument 0 przekazywany do funkcji **main** tworzonego procesu. Powinna być to nazwa pliku wykonywalnego ale bez ścieżki.
arg0	Argument 1 przekazywany do funkcji **main** tworzonego procesu
…	…
argN	Argument N przekazywany do funkcji **main** tworzonego procesu
argv[]	Tablica wskaźników do łańcuchów zawierających parametry

Wykonanie funkcji **exec** powoduje zastąpienie starego segmentu kodu, danych i stosu nowymi

Nowy proces dziedziczy ze starego PID, PPID, priorytet, środowisko, katalog bieżący.

Funkcja zwraca - 1 gdy wystąpił błąd.



Rys. 3-8 Przekazywanie argumentów do programu uruchamianego funkcją **execl()**

```
// Uruchomienie ./exec_test kroki_mac kroki_pot
#include <stdio.h>

main(int argc, char * argv[]){
  int pid,i,j,status;
  char kroki[20];
  if((pid = fork()) == 0) { // Proces potomny  ---
      // Uruchomienie programu pot kroki
      execl(”./pot”,”pot”,argv[2] ,NULL);
      perror(„Blad funkcji exec”);

  } else   { // Proces macierzysty ----------------
      for(j=1;j < atoi(argv[1]);j++) {
       printf("Macierzysty - krok %d \n",j);
       sleep(1);
      }
  }
  pid = wait(&status); // Czekamy na proces potomny
  printf(“Proces %d zakończony, status %d\n”,
  pid,WEXITSTATUS(status));
}
```

Program 3-4 Program exec_test - ilustracja działania funkcji **execl**

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

main( int argc,char * argv[]) {
  int j;
  for(j=0;j<atoi(argv[1]);j++) {
       printf("Potomy %s krok %d\n",argv[1],j);
       sleep(1);
  }
  _exit(atoi(argv[1]));
}
```

Listing 3-1 Proces pot2.c używany przez program exec_test

2.5 Wykonanie polecenia systemowego

Jeszcze inną metodą utworzenia procesu jest zastosowanie funkcji system.

```
int system(char * command)
```

command – łańcuch zawierający polecenie do wykonania

Funkcja system powoduje uruchomienie interpretera poleceń shell **/bin/sh** i przekazanie mu do wykonania łańcucha **command**. Wykonane mogą być programy, polecenia systemu lub skrypty.

Funkcja zwraca: 0 – sukces, - 1 - błąd

```
main( int argc,char * argv[]) {
  int pid, status;
  int i,j ;
  char buf[80];
  int stat;
  if((pid = fork()) == 0) { // Potomny1
      sprintf(buf,"./pot %d %s",i+1,argv[i+2]);
      system(buf);
      _exit(i+1);
  }
  // Macierzysty
  for(i=1;i<atoi(argv[1]);i++) {
       printf("Macierzysty krok %d \n",i);
       sleep(1);
  }
  pid = wait(&stat);
  printf("Proces %d zakoncz, stat=%d\n",
  pid,WEXITSTATUS(stat));
}
```

Przykład 3-2 Tworzenie procesu za pomocą funkcji **system**

2.6 Funkcja clone

Jeszcze inną metodą tworzenia procesów jest funkcja clone. Pozwala ona na dzielenie z utworzonym procesem segmentu kodu i innych obiektów. Używana do tworzenia wątków.

```
int clone(int (*fn)(void *), void *child_stack,
                  int flags, void *arg, ...
                  /* pid_t *ptid, struct user_desc
*tls, pid_t *ctid */ );
```

fn	Funkcja według której tworzony jest kod
child_stack	Wskaźnik na położenie stosu
flags	Flagi
arg	Argumenty przekazywane do funkcji tworzonego procesu
ptid	Identyfikator nowego procesu
tls	Thread local storage

2.7 Atrybuty procesu

Atrybuty procesu są to informacje wykorzystywane przez system do zarządzania procesami a więc do ich identyfikacji, szeregowania, utrzymywania bezpieczeństwa i uruchamiania.

Najważniejsze atrybuty procesu:
- PID - identyfikator procesu,
- PPID - PID procesu macierzystego,
- UID - identyfikator użytkownika
- GID - identyfikator grupy do której należy użytkownik
- SID - identyfikatory sesji
- PGRP - identyfikatory grupy procesów,
- priorytet procesu,
- CWD - katalog bieżącym
- katalog główny
- otoczenie procesu

Każdy proces (z wyjątkiem procnto) posiada dokładnie jeden proces potomny.

Procesy tworzą więc hierarchię która może być przedstawiona jako drzewo.

Identyfikator procesu PID i procesu potomnego PPID

Dla każdego procesu utrzymywany jest identyfikator jego procesu potomnego PPID *(ang. Parent Process Identifier*

`pid_t getpid(void)`	- funkcja zwraca PID procesu bieżącego
`pid_t getppid(void)`	- funkcja zwraca PID procesu macierzystego

Grupa procesów

Grupa procesów jest to taki zbiór procesów który posiada jednakowy parametr PGID. Standardowo PGID jest dziedziczony z procesu macierzystego ale funkcja **`setpgrp`** może go ustawić na PID procesu bieżącego.

Proces w którym tak zrobiono staje się procesem wiodącym grupy (*ang. session leader*).

`pid_t getpgrp(void)` - funkcja zwraca numer grupy procesów dla procesu bieżącego

`pid_t setpgrp(void)` - funkcja ustawia PGID procesu na jego PID

Funkcja **`setpgid`** pozwala na dołączenie do istniejącej grupy procesów lub na utworzenie nowej.

`int setpgid(pid_t pid, pid_t pgid)`

Gdzie:

`pid` 0 albo PID procesu którego PGID chcemy ustawić

`pgid` 0 gdy tworzymy grupę albo PGID istniejącego procesu gdy dołącza my do istniejącej grupy

Funkcja zwraca 0 gdy sukces –1 gdy błąd.

Grupy procesów wykorzystuje się w połączeniu z sygnałami – można wysłać sygnał do całej grupy procesów.

Sesja i identyfikator sesji

Kiedy użytkownik rejestruje się w systemie będzie on należał do sesji związanej z jego bieżącym terminalem (terminalem sterującym).

Sesja identyfikowana jest przez identyfikator sesji SID (*ang. Session Identifier*) i składa się z jednej lub wielu grup procesów.

Proces może uzyskać SID innego procesu lub samego siebie za pomocą funkcji:

`pid_t getsid(pid_t pid)` - funkcja zwraca SID procesu

Gdzie:
`pid` 0 dla procesu bieżącego albo PID procesu którego SID chcemy uzyskać

Demon to proces który nie ma terminala sterującego.

Zwykle procesy pełniące funkcje serwerów są demonami.

Uruchomiony z konsoli proces można przekształcić w demona gdy umieścimy go w sesji nie posiadającej terminala sterującego.

Sesję można zmienić za pomocą funkcji **`setsid`**.

`pid_t setsid(void)` - funkcja tworzy nową sesję i przemieszcza tam proces bieżący

Wywołanie tej funkcji tworzy nową sesję nie powiązaną z żadnym terminalem sterującym i grupę nową procesów.

Proces bieżący zostaje przeniesiony do tej sesji i zostaje procesem wiodącym tej grupy. Jest to jedyny proces w tej sesji i grupie.

Identyfikator użytkownika i grupy

Każdy z użytkowników systemu posiada swój identyfikator i należy do pewnej grupy.

Pliki: **`/etc/passwd`** **`/etc/group`**

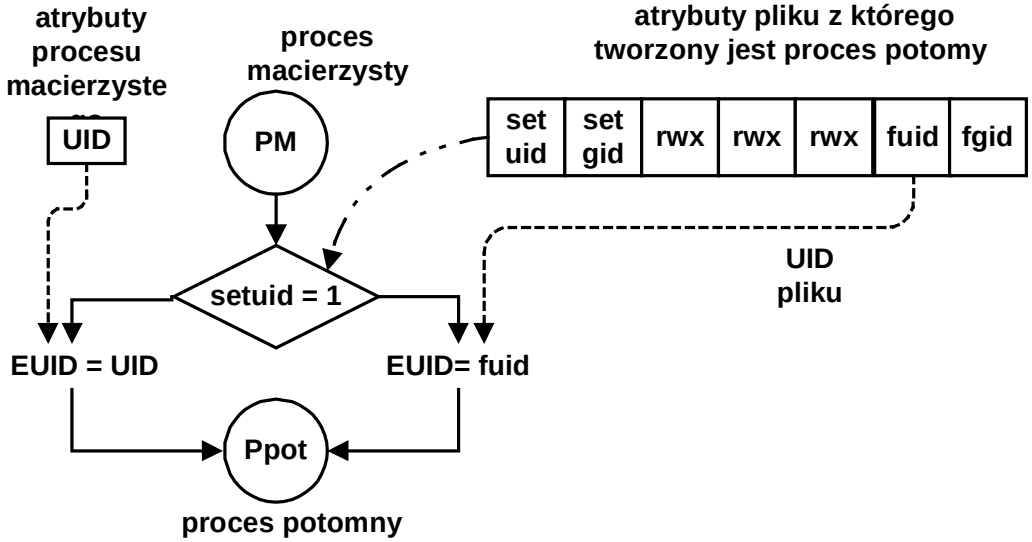
Rzeczywisty identyfikator użytkownika UID (*ang. User Identifier)*
Rzeczywisty identyfikator grupy GID (*ang. Group Identifier)*.

Prawa dostępu sprawdzane są w oparciu o efektywny identyfikator procesu EUID (*ang. Effective User Identifier*) i grupy EGID *(ang. Effective Group Identifier*).

Efektywny identyfikator uzytkownika może być taki jak UID właściciela pliku z którego tworzony jest nowy proces gdy ustawiony jest specjalny bit **`setuid`** który jest atrybutem pliku.

Gdy **`setuid`** jest ustawiony to tworzony proces będzie miał efektywny identyfikator użytkownika EUID taki jak UID właściciela pliku wykonywalnego z którego tworzony jest proces potomny.

Gdy bit **`setuid`** nie jest ustawiony EUID równy jest UID procesu macierzystego.



Rys. 3-9 Ustalanie EUID procesu potomnego

Atrybuty testować można z poziomu programu.

`int getuid(void)`	UID procesu bieżącego
`int geteuid(void)`	EUID procesu bieżącego
`int getgid(void)`	GID procesu bieżącego
`int getegid(void)`	EGID procesu bieżącego

System oferuje dwie funkcje ustawiania UID i GID.

`int setuid(int uid)`	UID procesu bieżącego
`int setgid(int gid)`	EUID procesu bieżącego

Gdy proces wykonujący funkcję należy do użytkownika root może on ustawić dowolny UID i EUID (będą one takie same).

Gdy proces nie należy do użytkownika root może on tylko ustawić efektywny identyfikator użytkownika EUID taki jak rzeczywisty UID.

```
// Program:  info1.c  Atrybuty procesu
#include <stdio.h>
main(int argc, char * argv[]) {
  int pid,status;
  pid = getpid();
  printf("UID: %d GID: %d EUID: %d EGID: %d\n",
  getuid(),getgid(),geteuid(), getegid());
  printf("PID: %d PPID: %d PGRP: %d SID: %d \n",
  pid,getppid(), getpgrp(), getsid(0));
  return pid;
}
```

Przykład 3-1 Program info1 podający atrybuty procesu

```
$./info1
UID: 100 GID: 100 EUID: 0 EGID: 100
PID: 2539559 PPID: 1142819 PGRP: 2539559 SID: 1142819
```

Wynik 3-1 Działanie programu info1

Środowisko procesu

Środowisko procesu (ang. *enviroment*) jest to zbiór napisów postaci:
NAZWA_ZMIENNEJ=WARTOŚĆ_ZMIENNEJ

```
char *getenv(char * nazwa)
```

Gdzie:
nazwa Nazwa zmiennej środowiska którego wartość chacemy uzyskać

Funkcja zwraca wskaźnik do wartości zmiennej środowiska lub NULL gdy zmiennej nie znaleziono.

```
int putenv(char * nazwa)
```

Gdzie:
nazwa Nazwa zmiennej środowiska i jej nowa wartość

Funkcja zwraca 0 gdy sukces, -1 gdy błąd.

```
// Program:  info2.c  Srodowisko procesu
#include <stdio.h>
main(int argc, char * argv[], char *envp[]) {
  char *ptr;
  int res;
  while(*envp)
    printf("%s\n",*envp++);
  res = putenv("MOJPAR=5");
  if(res == 0) {
     ptr = getenv("MOJPAR");
     if(ptr != NULL)
        printf("Parametr MOJPAR=%s\n",
     getenv("MOJPAR"));
  }
  return 0;
}
```

Przykład 0-1 Program info2 podający atrybuty procesu

```
PATH=/bin:/usr/bin:/usr/photon/bin
SHELL=/bin/sh
HOSTNAME=qumak
TMPDIR=/tmp
...
HOME=/home/juka
TERM=qansi-m
LOGNAME=juka
MOJPAR=5
```

Wynik 0-1 Wyniki działania programu info2

Priorytet i strategia szeregowania

Priorytet – liczba z zakresu 0 – 31wpływająca na szeregowanie procesu.

```
int sched_getparam(pid_t pid,struct sched_param
*param);

    struct sched_param {
       ...
       int sched_priority;
       ...
    };
```

Gdzie:

pid	PID procesu którego priorytet jest testowany, 0 gdy procesu bieżącego
sched_priority	Priorytet procesu

Funkcja zwraca 0 gdy sukces, -1 gdy błąd.

2.8 Ustanawianie ograniczeń na użycie zasobów

W każdym systemie komputerowym zasoby potrzebne do tworzenia i wykonywania procesów są ograniczone.

W przypadku gdy w systemie działa wiele procesów ważną rzeczą jest zabezpieczenie systemu przed wyczerpaniem zasobów spowodowanym przez nadmierne zużycie zasobów przez procesy wchodzące w skład aplikacji.

☞ W bezpiecznym systemie operacyjnym powinien istnieć mechanizm limitujący pobieranie zasobów przez procesy.

System Linux posiada mechanizmy pozwalające na ustanowienie limitu na takie zasoby jak:

- czas procesora,
- pamięć operacyjna,
- pamięć wirtualna
- wielkość pamięci pobranej ze sterty,
- wielkość segmentu stosu,
- maksymalna liczba deskryptorów plików,
- maksymalna wielkość pliku utworzonego przez proces
- maksymalna liczba procesów potomnych tworzonych przez proces.
- Maksymalna liczba blokad plików i obszarów pamięci operacyjnej

Dla każdego z tych zasobów istnieje:
- ograniczenie miękkie (*ang. soft limit*)
- ograniczenie twarde (*ang. hard limit*).

Ograniczenie miękkie może być zmieniane przez proces bieżący ale nie może przekroczyć twardego.
Ograniczenie twarde może być zmieniane przez proces o statusie administratora.

Do testowania limitów zasobów służy funkcja **getrlimit**.

getrlimit – pobranie aktualnego limitu zasobów

```
int getrlimit(int resource, struct rlimit *rlp)
```

Gdzie:

resource	Określenie zasobu.
rlp	Wskaźnik na strukturę zawierającą bieżące i maksymalne ograniczenie.

Funkcja zwraca **0** gdy sukces a **–1** gdy błąd.

Jako pierwszy parametr funkcji podać należy numer testowanego zasobu które podaje tabela. Funkcja powoduje skopiowanie do struktury **rlp** aktualnych ograniczeń.

Struktura ta zawiera co najmniej dwa elementy:
rlim_cur - zawiera ograniczenie miękkie
rlim_max zawierający ograniczenie twarde.

Do ustawiania limitów zasobów służy funkcja **setrlimit**.

setrlimit – ustanowienie nowego limitu zasobów

```
int setrlimit(int resource, struct rlimit *rlp)
```

Funkcja zwraca **0** gdy sukces a **–1** gdy błąd.

Gdy proces próbuje pobrać zasoby ponad przydzielony limit system operacyjny może:

1. Zakończyć proces.
2. Wysłać do niego sygnał .
3. Zakończyć błędem funkcję pobierającą dany zasób.

Oznaczenie	Opis	Akcja przy przekroczeniu
RLIMIT_AS	Pamięć wirtualna	Wysłanie sygnału SIGSEGV do procesu przekraczającego zasób
RLIMIT_CORE	Pamięć operacyjna	Zakończenie procesu z zapisaniem na dysku obrazu pamięci operacyjnej.
RLIMIT_CPU	Czas procesora	Wysłanie sygnału SIGXCPU do procesu przekraczającego zasób.
RLIMIT_DATA	Wielkość pamięci pobranej ze sterty.	Funkcja pobierająca pamięć kończy się błędem.
RLIMIT_FSIZE	Maksymalna wielkość pliku utworzonego przez proces. Gdy 0 to zakaz tworzenia plików.	Wysłanie sygnału SIGXFSZ do procesu przekraczającego zasób. Gdy sygnał jest ignorowany to plik nie zostanie powiększony ponad limit.
RLIMIT_NOFILE	Maksymalna liczba deskryptorów plików tworzonych przez proces.	Funkcja tworząca ponad limitowe pliki skończy się błędem.
RLIMIT_STACK	Maksymalny rozmiar stosu	Wysłanie sygnału SIGSEGV do procesu przekraczającego stos.
RLIMIT_NPROC	Maksymalna liczba procesów potomnych tworzonych przez proces.	Procesy przekraczające limit nie będą utworzone.
RLIMIT_MSGQUEUE	Pamięć zajmowana przez kolejki	

	komunika tów POSIX	

Tab. 0-1 Zestawienie niektórych zasobów systemowych podlegających ograniczeniu

Ustanowienie ograniczenie **RLIMIT_CPU** na czas zużycia procesora w systemach działających nieprzerwanie nie ma dużego zastosowania. Powodem jest fakt że jeżeli proces ma działać w nieskończoność to limit ten musi być znaczny. Tak więc system operacyjny zareaguje dopiero wtedy gdy ten limit zostanie przekroczony a w tym czasie inne procesy mogły nie uzyskać potrzebnego im czasu procesora.

Odpowiednim rozwiązaniem tego problemu jest szeregowanie sporadyczne które narzuca limit na zużycie czasu procesora w przesuwającym się do przodu oknie czasowym.

```
#include <stdlib.h>
#include <sys/resource.h>
int main(int argc, char *argv[]) {
  int res, i, num = 0;
  struct rlimit rl;
  printf("       CUR          MAX \n");
  getrlimit(RLIMIT_CPU,&rl);
  printf("CPU    %d   %d \n",rl.rlim_cur, rl.rlim_max);
  getrlimit(RLIMIT_CORE,&rl);
  printf("CORE   %d   %d \n",rl.rlim_cur, rl.rlim_max);
  rl.rlim_cur = 2;
  setrlimit(RLIMIT_CPU,&rl);
  while (1);
  return 0;
}
```

Program 0-1 Program **rlimit.c** testujący i nakładający ograniczenia na pobierane przez proces zasoby

Gdy przydzielony czas procesora ulegnie wyczerpaniu proces zakończy się z komunikatem:

```
$CPU time limit exceeded (core dumped)
```