

## 2. Zarządzanie procesami

### 2.1 Funkcje zarządzania procesami

Administrowanie procesami obejmuje następujące czynności:

- Testowanie atrybutów procesu.
- Ustawianie atrybutów procesu.
- Tworzenie procesu.
- Kończenie procesu.
- Synchronizacja zakończenia procesu potomnego z jego procesem macierzystym.
- Testowanie statusu zakończonego procesu potomnego

Ut看wienie kopii procesu bieżącego.	<code>fork()</code>
Zastąpienie procesu bieżącego innym procesem – rodzina funkcji.	<code>exec*()</code>
Ut看wienie procesu potomnego – rodzina funkcji.	<code>spawn*()</code>
Wykonanie programu lub skryptu.	<code>system()</code>
Zakończenie procesu	<code>exit()</code>
Czekanie na zakończenie procesu	<code>wait(),</code> <code>waitpid()</code>

Rysunek 1 Ważniejsze funkcje zarządzania procesami

### 2.2 Tworzenie kopii procesu bieżącego – funkcja `fork`

Funkcja posiada następujący prototyp.

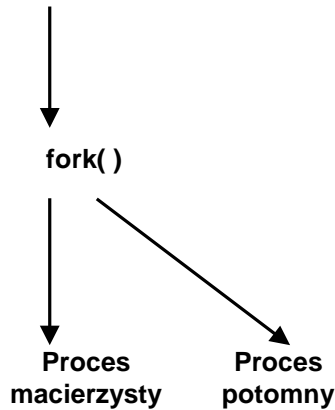
```
pid_t fork(void)
```

Funkcja tworzy kopię procesu bieżącego czyli tego procesu który wykonuje funkcję `fork()`. Utworzony proces potomny różni się od macierzystego pod następującymi względami:

1. Ma inny PID.
2. Ma inny PID procesu macierzystego (ang. parent PID).
3. Proces potomny zachowuje otwarte pliki procesu macierzystego ale tworzy własne kopie ich deskryptorów.

Funkcja `fork ( )` zwraca:

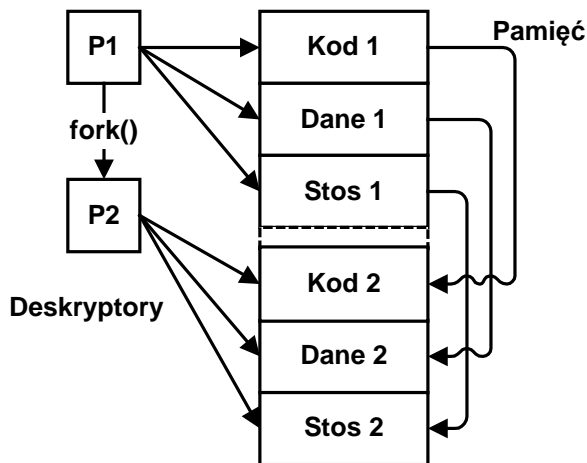
- 0 w procesie potomnym
- > 0 w procesie macierzystym zwracany jest PID procesu potomnego
- 1 błąd



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.

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.



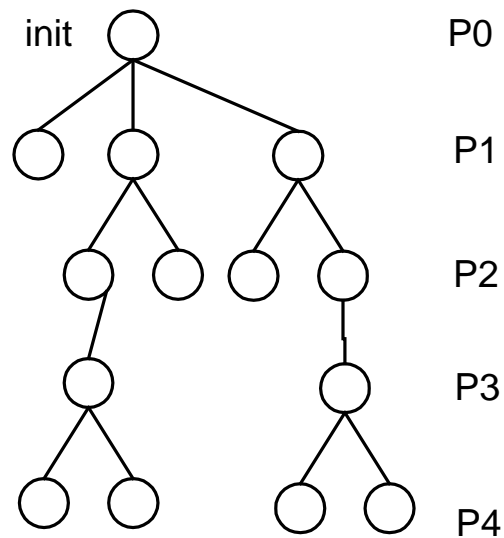
Rys. 2-2 Proces P1 wykonał funkcję `fork` i utworzył proces P2. Procesy P1 i P2 posiadają własne segmenty danych i stosu.

```
#include <stdio.h>
#include <process.h>
main(int argc, char * argv[]) {
    int pid;
    int i,j;
    if((pid = fork()) == 0) { // Proces potomny ---
        printf(" Proces potomny PID:  %d \n", getpid());
        for(i=1;i < 10;i++) {
            printf("Potomny - krok %d\n",i);
            sleep(1);
        }
        exit(0);
    } else { // Proces macierzysty -----
        printf("Proces macierzysty PID:  %d \n", getpid());
        for(j=1;j < 20;j++) {
            printf("Macierzysty - krok %d\n",j);
            sleep(1);
        }
    }
    printf("Koniec programu\n");
}
```

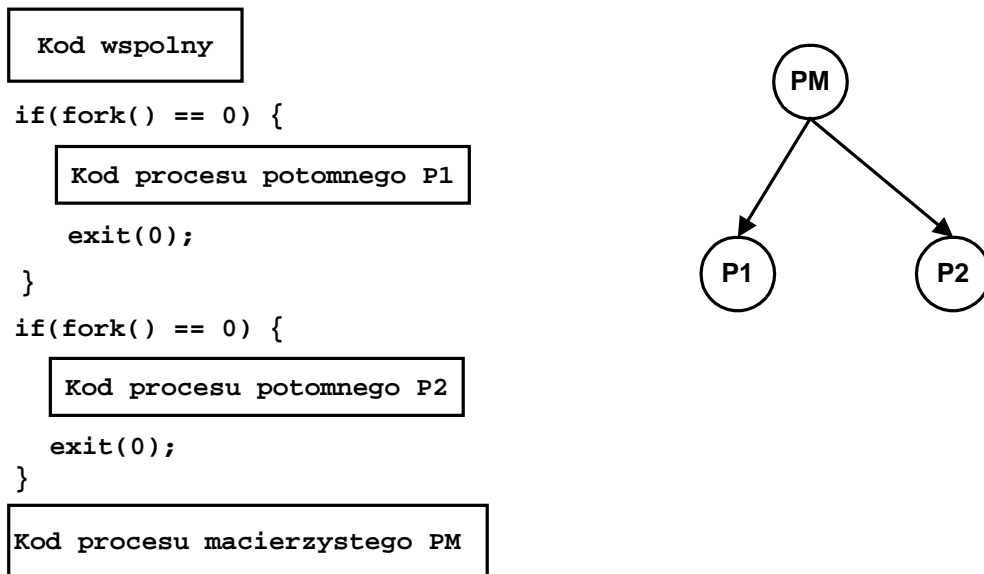
Przykład 2-1 Podstawowy schemat wykorzystania funkcji `fork`

### Drzewo procesów

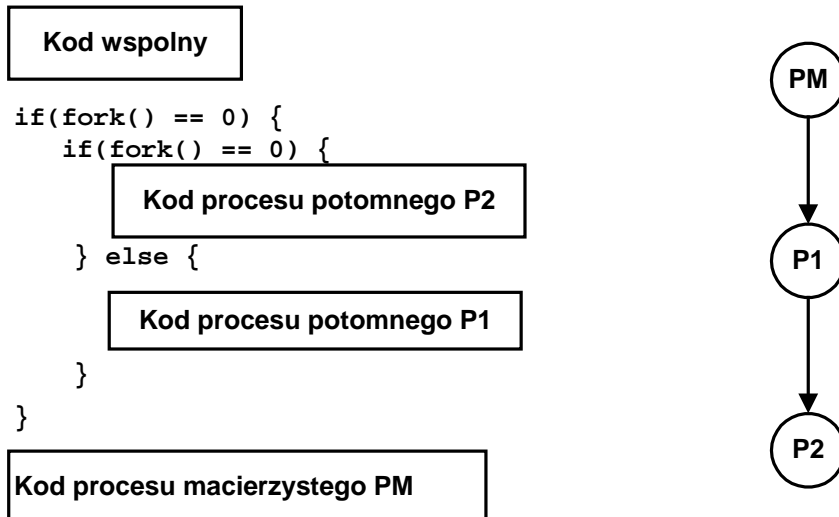
Każdy z procesów ma jeden proces macierzysty ale może utworzyć wiele procesów potomnych. Stąd procesy tworzą drzewo. Jego korzeniem jest pierwszy proces w systemie (init, procnto) który tworzy procesy kolejne.



Rys. 2-3 Drzewo procesów



Rys. 2-4 Schemat użycia funkcji `fork` do utworzenia dwu procesów potomnych P1 i P2. Procesy są utworzone na jednakowym poziomie hierarchii.



Rys. 2-5 Schemat użycia funkcji `fork` do utworzenia dwu procesów potomnych P1 i P2. Proces P2 jest procesem potomnym procesu P1.

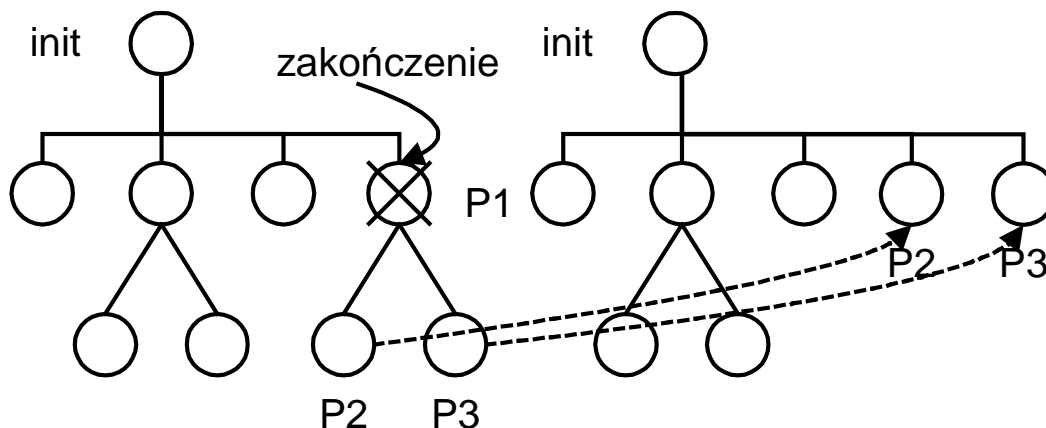
## 2.3 Obsługa zakończenia procesów

### 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.



**Procesy P2 i P3 adoptowane przez proces init**

**!** Nie należy kończyć procesu który posiada nie zakończone procesy potomne.

## 2.4 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.

```
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 deskrytorze 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 deskrytorze procesu a proces przechodzi do stanu „zombie”.
5. Do procesu macierzystego wysyłany jest sygnał **SIGCHLD**.

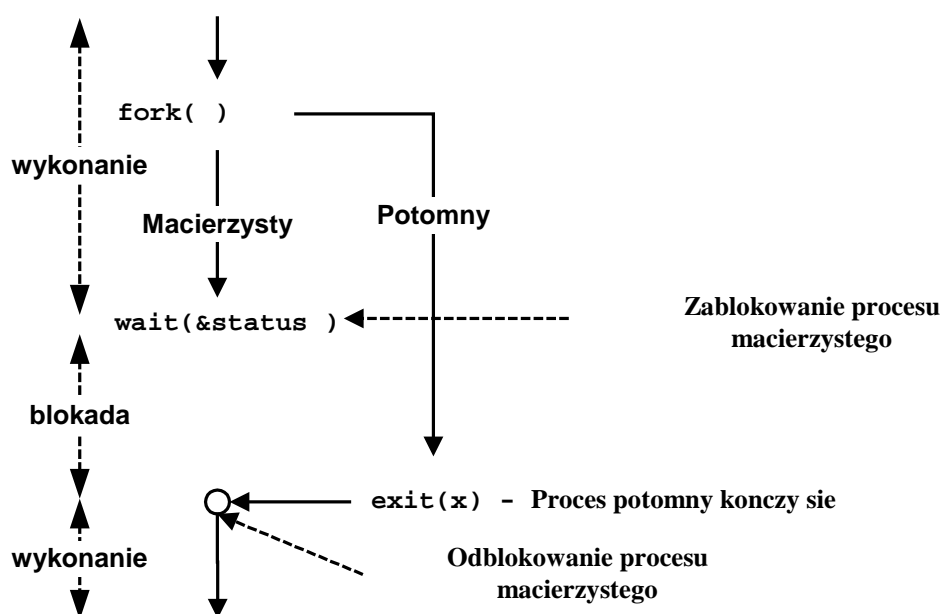
## 2.5 Synchronizacja zakończenia procesów

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

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

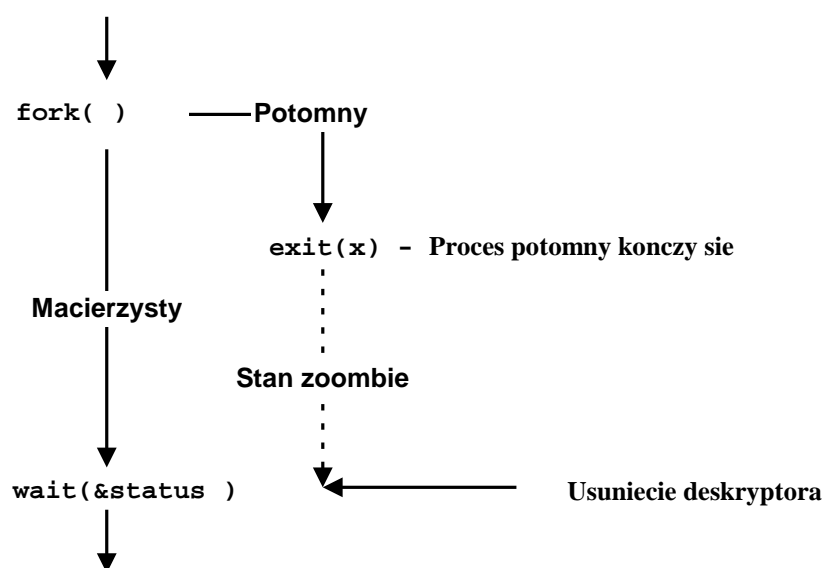
Działanie funkcji `wait` jest następujące:

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 zakończony procesu oraz jego status.
3. Gdy brak jakichkolwiek procesów potomnych funkcja `wait` zwraca `-1`,



Rys. 2-6 Proces macierzysty czeka na zakończenie się procesu potomnego.





Rys. 2-7 Proces potomny kończy się wcześniej niż proces macierzysty

```
#include <stdio.h>
#include <process.h>
main(int argc, char * argv[]) {
    int pid,status;
    int i,j,k1,k2;
    i=0;    j=0;
    k1 = atoi(argv[1]); // Liczba kroków procesu macierzystego
    k2 = atoi(argv[2]); // Liczba kroków procesu potomnego
    if((pid = fork()) == 0) { // Proces potomny ---
        printf(" Proces potomny PID:  %d \n", getpid());
        for(i=1;i < k2;i++) {
            printf("Potomny - krok %d , j= %d \n",i,j);
            sleep(1);
        }
        printf("Proces potomny konczy się\n");
        exit(k2);
    } else { // Proces macierzysty -----
        printf("Proces macierzysty PID: %d \n", getpid());
        for(j=1;j < k1;j++) {
            printf("Macierzysty - krok %d , i= %d\n",j,i);
            sleep(1);
        }
    }
    pid = wait(&status); // Czekamy na proces potomny
    printf("Proces %d zakończony, status
%d\n",pid,WEXITSTATUS(status));
}
```

Przykład 2-2 Schemat wykorzystania funkcji `fork`, `wait`, `exit`

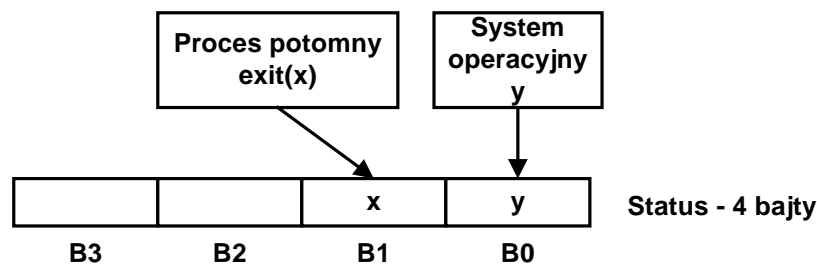
## 2.6 Testowanie statusu zakończonego procesu.

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 zakończenia procesu.
2. Zakończonego procesu potomnego który umieszcza tam wartość kodu powrotu – jest to parametr funkcji `exit`.



Rys. 2-8 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
<code>WIFEXITED(status)</code>	zwraca > 0 gdy proces potomny był zakończony normalnie
<code>WEXITSTATUS(status)</code>	zwraca kod powrotu y przekazany przez funkcję <code>exit(y)</code> z procesu potomnego
<code>WIFSIGNALED(staus)</code>	zwraca > 0 gdy proces potomny był zakończony przez nie obsłużony sygnał
<code>WTERMSIG(status)</code>	zwraca numer sygnału gdy proces był zakończony przez sygnał

Tabela 2-1 Makra do testowanie statusu zakończonego procesu potomnego.

```
// Testowanie przyczyny zakończenia procesu
int pid, staus;
...
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));
...
```

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.

## 2.7 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`.

```
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.

```
#include <stdio.h>
#include <process.h>
main(void){
    int pid,status,j,i;
    if((pid = fork()) == 0) { // Proces potomny ---
        // Uruchomienie polecenia ls -l
        execl(„/bin/ls“,„ls“,„-l“,NULL);
        perror(„Bład funkcji exec“);

    } else { // Proces macierzysty -----
        for(j=1;j < 10;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));
}
```

Przykład 2-3 Ilustracja działania funkcji execl

## 2.8 Tworzenie nowego procesu za pomocą funkcji spawn

Każda funkcja z rodziny `spawn` tworzy nowy proces potomny na podstawie pliku wykonywalnego określonego w jednym z parametrów funkcji.

```
pid_t spawnl(int mode, char * path, arg0,
arg1, ..., argN, NULL)
pid_t spawnv(int mode, char * path, char * argv[])
```

**mode** tryb wykonania procesu (`P_WAIT`, `P_NOWAIT`, `P_OVERLAY`, `P_NOWAITO`)

**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

... ..

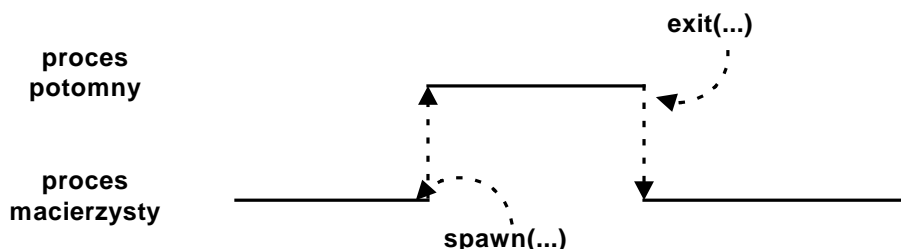
**argN** Argument N przekazywany do funkcji `main` tworzonego procesu

**argv[ ]** Tablica wskaźników do łańcuchów zawierających parametry przekazywane do funkcji `main` tworzonego procesu

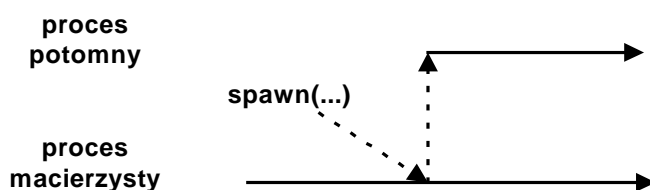
Środowisko (*ang. Enviroment*) jest dziedziczone z procesu macierzystego.

Funkcja zwraca:

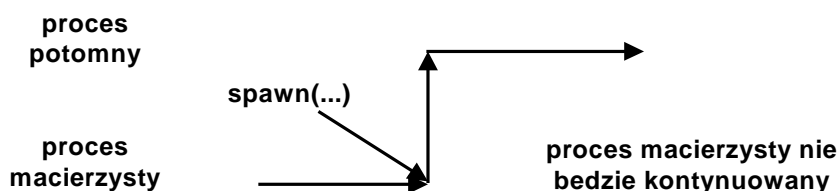
- > 0 - pid utworzonego procesu potomnego
- 1 - błąd gdy proces nie może być utworzony



Rys. 2-9 Tryb `P_WAIT` – Proces macierzysty czeka na zakończenie się procesu potomnego



Rys. 2-10 Tryb P\_NOWAIT – Proces macierzysty i potomny wykonywane są współbieżnie



Rys. 2-11 Tryb P\_OVERLAY – Proces macierzysty zastępowany przez proces potomny

```
// Ilustracja działania funkcji spawnl - uruchomienie programu
my_prog
#include <stdio.h>
#include <process.h>
main(void){
    int pid,i,res;

    res = spawnl(P_NOWAIT,"/home/user/my_prog","my_prog",NULL);
    if(res < 0) perror("SPAWN");
    for(j=1;j < 10;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));
}
```

Przykład 2-4 Proces macierzysty za pomocą funkcji `spawn` tworzy współbieżny proces potomny



## 2.9 Wykonanie polecenia systemowego

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

**command** – łańcuch zawierający polecenie do wykonania

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

Funkcja zwraca: 0 – sukces, - 1 - błąd

Przykład: `system(„ls -l”)`

## 2.10 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:

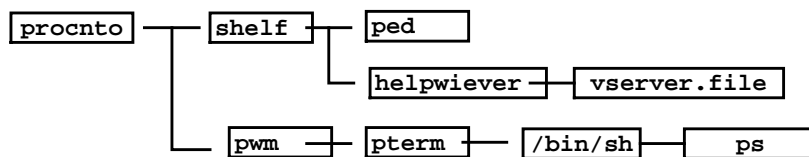
- PID - identyfikator procesu,
- PPID - PID procesu macierzystego,
- UID - identyfikator użytkownika
- GID - identyfikator grupy do której należy użytkownik
- SID - identyfikatory sesji
- PGRP - identyfikatory grupy procesów,
- priorytet procesu,
- CWD - katalog bieżącym
- katalog główny
- otoczenie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

```
$ps -l
UID          PID          PPID  C PRI  NI       SZ     TIME CMD
100      1114130      999464  -  10   0       568K  00:00:00 ps
100       688154           1  -  10   0       684K  00:00:00 pwm
100       733215           1  -  10   0      1368K  00:00:00 shelf
100       823331      733215  -  10   0       872K  00:00:00 ped
100       999463      688154  -  10   0       728K  00:00:00 pterm
100       999464      999463  -  10   0       604K  00:00:00 /bin/sh
```

Przykład 2-1 Uzyskiwanie listy procesów za pomocą polecenia ps

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 dołączamy 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*)

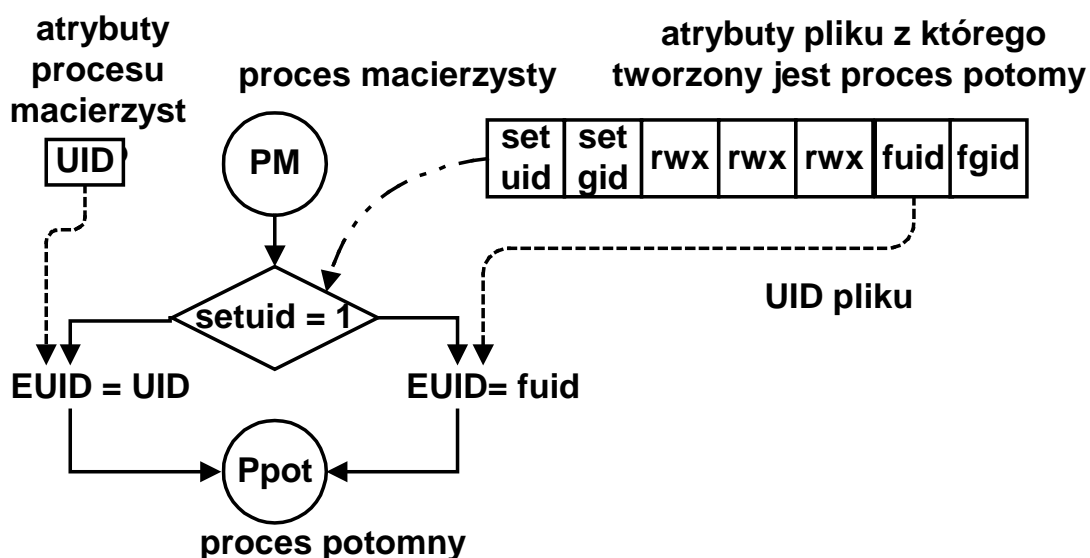
Rzeczywisty identyfikator grupy GID (*ang. Group Identifier*).

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

Efektywny identyfikator użytkownika 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. 2-12 Ustalanie EUID procesu potomnego

Atrybuty testować można z poziomu programu.

<code>int getuid(void)</code>	UID	procesu bieżącego
<code>int geteuid(void)</code>	EUID	procesu bieżącego
<code>int getgid(void)</code>	GID	procesu bieżącego
<code>int getegid(void)</code>	EGID	procesu bieżącego

System oferuje dwie funkcje ustawiania UID i GID.

<code>int setuid(int uid)</code>	UID	procesu bieżącego
<code>int setgid(int gid)</code>	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 2-2 Program info1 podający atrybuty procesu

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

Wynik 2-1 Działanie programu info1

## Środowisko procesu

Środowisko procesu (ang. *environment*) jest to zbiór napisów postaci:  
**NAZWA\_ZMIENNEJ=WARTOŚĆ\_ZMIENNEJ**

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

Gdzie:

**nazwa** Nazwa zmiennej środowiska którego wartość chcemy 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;
}
```

```
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 – 31 wpływająca na szeregowanie procesu.

```
int getprio(pid_t pid)
```

Gdzie:

**pid** PID procesu którego priorytet jest testowany, 0 gdy procesu bieżącego


Funkcja zwraca priorytet procesu, -1 gdy błąd.



## 2.11 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 systemie czasu rzeczywistego powinien istnieć mechanizm limitujący pobieranie zasobów przez procesy.

System QNX6 Neutrino posiada mechanizmy pozwalające na ustanowienie limitu na takie zasoby jak:

- czas procesora,
- pamięć operacyjna,
- 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.

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 ustawiania i testowania limitów zasobów można użyć polecenia `ulimit`

```
ulimit [-acdfHlmpsv] [value]
```

Niektóre opcje:

-n Limit na liczbę deskryptorów plików

- p Ograniczenie liczby procesów
- t Czas użycia procesora w sekundach
- f Liczba bloków pamięci dyskowej która może być zapisana

```
ulimit -a
time(cpu-seconds)      unlimited
file(blocks)           unlimited
coredump(blocks)       unlimited
data(kbytes)           unlimited
stack(kbytes)          unlimited
lockedmem(kbytes)      unlimited
memory(kbytes)         unlimited
nofiles(descriptors)  1000
processes              unlimited
vmemory(kbytes)        unlimited
```

Przykład 0-1 Użycie polecenia ulimit

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.

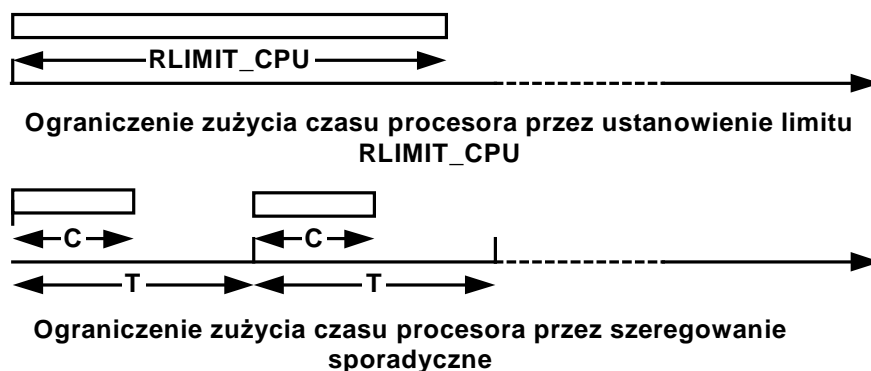
Oznaczenie	Opis	Akcja przy przekroczeniu
<b>RLIMIT_CORE</b>	Pamięć operacyjna	Zakończenie procesu z zapisaniem na dysku obrazu pamięci operacyjnej.
<b>RLIMIT_CPU</b>	Czas procesora	Wysłanie sygnału SIGXCPU do procesu przekraczającego zasób.
<b>RLIMIT_DATA</b>	Wielkość pamięci pobranej ze sterty.	Funkcja pobierająca pamięć kończy się błędem.
<b>RLIMIT_FSIZE</b>	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.
<b>RLIMIT_NOFILE</b>	Maksymalna liczba deskryptorów plików tworzonych przez proces.	Funkcja tworząca ponad limitowe pliki skończy się błędem.
<b>RLIMIT_STACK</b>	Maksymalny rozmiar stosu	Wysłanie sygnału SIGSEGV do procesu przekraczającego stos.
<b>RLIMIT_NPROC</b>	Maksymalna liczba procesów potomnych tworzonych przez proces.	Procesy przekraczające limit nie będą utworzone.

Tabela 0-1 Zestawienie zasobów systemowych podlegających ograniczeniu

Ustanowienie ograniczenia `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.

W szeregowaniu sporadycznym zużycie czasu procesora nie może przekroczyć  $C$  jednostek w każdym z okresów  $T$  podczas gdy ograniczenie `RLIMIT_CPU` dotyczy całego okresu wykonywania się procesu.



Różne sposoby ograniczenia czasu użycia procesora

```
#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;
}
```

Przykład 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)
```