Procesy

Proces (zwany też *zadaniem*) jest jednostką aktywną, kontrolowaną przez system operacyjny i związaną z wykonywanym programem. Proces ma przydzielone zasoby typu pamięć (segment kodu, segment danych, segment stosu, segment danych systemowych), procesor, urządzenia zewnętrzne itp. Część przydzielonych zasobów jest do wyłącznej dyspozycji procesu (np. segment danych, segment stosu), część jest współdzielona z innymi procesami (np. procesor, segment kodu w przypadku współbieżnego wykonywania tego samego programu w ramach kilku procesów).
W zależności od aktualnie posiadanych zasobów wyróżnia się *stany procesu* (np. wykonywany, uśpiony, gotowy), które zmieniają się cyklicznie w związku z wykonywanym programem lub ze zdarzeniami zachodzącymi w systemie. Szczegółowy stan procesu, umożliwiający kontynuację jego wykonywania po przerwaniu nazywany jest *kontekstem procesu*.

Każdy proces może tworzyć inne procesy, które staną się jego *potomkami*, a proces tworzący staje się ich *przodkiem* (zwanym też *procesem macierzystym*, *procesem rodzicielskim* lub krótko *rodzicem*). Powstaje w ten sposób hierarchiczna struktura procesów (podobnie jak katalogów), na czele której stoi proces systemowy init o identyfikatorze 1. Każdy proces, z wyjątkiem procesu o identyfikatorze 1, tworzony jest przez inny proces.

Jeżeli proces macierzysty zakończy działanie przed procesem potomnym, to proces potomny staje się *sierotą* (ang. orphant) i jest „adoptowany" przez proces systemowy init, który staję się w ten sposób jego przodkiem.

Jeżeli proces potomny zakończył działanie przed wywołaniem funkcji `wait` w procesie macierzystym, potomek pozostanie w stanie *zombi* (proces taki nazywany jest zombi, upiorem, duchem lub mumią). Zombi jest procesem, który zwalnia wszystkie zasoby (nie zajmuje pamięci, nie jest mu przydzielany procesor), zajmuje jedynie miejsce w tablicy procesów w jądrze systemu operacyjnego i zwalnia je dopiero w momencie wywołania funkcji `wait` przez proces macierzysty

Poniższe funkcje opisane są w 2 i 3 części pomocy systemowej.

Funkcja FORK

```
#include <sys/types.h>
#include <unistd.h>

PROTOTYPE: int fork( void );
  RETURNS:  success  : utworzenie procesu potomnego; W
                       procesie macierzystym funkcja zwraca
                       identyfikator (pid) procesu potomnego
                       (wartość większą od 1), a w procesie
                       potomnym wartość 0.
            error: -1
```

```
errno = EAGAIN (błąd alokacji wystarczającej ilości
                pamięci na skopiowanie stron rodzica i
                zaalokowanie struktury zadań)
        ENOMEM (nie można zaalokować niezbędnych struktur
                jądra z powodu braku pamięci)
```

UWAGI:
W momencie wywołania funkcji (przez proces który właśnie staje się przodkiem) tworzony jest proces potomny, który wykonuje współbieżnie ze swoim przodkiem ten sam program. Potomek rozpoczyna wykonywanie programu od wyjścia z funkcji `fork` i kontynuuje wykonując kolejną instrukcję, znajdującą się w programie po wywołaniu funkcji `fork`. Do funkcji `fork` wchodzi zatem tylko proces macierzysty, a wychodzą z niej dwa procesy: macierzysty i potomny, przy czym każdy z nich otrzymuję inną wartość zwrotną funkcji `fork`. Wartością zwrotną funkcji `fork`() w procesie macierzystym jest identyfikator (PID) potomka, a w procesie potomnym wartość 0. W przypadku błędnego wykonania funkcji `fork` potomek nie zostanie utworzony, a proces wywołujący otrzyma wartość -1.

Funkcja GETPID;GETPPID

```
#include <unistd.h>

PROTOTYPE: int getpid( void );
           int getppid( void );
 RETURNS:   success   :   zwrócenie odpowiednio własnego
                          identyfikatora, lub identyfikatora
                          procesu macierzystego.
            error: -1
```

Funkcja EXIT

```
#include <stdlib.h>

PROTOTYPE: void fork( int status );
  RETURNS: success : Funkcja powoduje normalne zakończenie
                     programu i zwraca do procesu
                     macierzystego wartość status.
```

UWAGI:
Funkcja kończy działanie procesu, który ją wykonał i powoduje przekazanie w odpowiednie miejsce tablicy procesów wartości status, która może zostać odebrana i zinterpretowana przez proces macierzysty.
Jeśli proces macierzysty został zakończony, a istnieją procesy potomne – to wykonanie ich nie jest zakłócone, ale ich identyfikator procesu macierzystego wszystkich procesów potomnych otrzyma wartość 1 będącą identyfikatorem procesu *init*. (proces potomny staje się *sierotą* (ang. orphant) i jest „adoptowany” przez proces systemowy init)

`exit(0)` – poprawne zakończenie wykonanie procesu
`exit(wartość <> 0 )` – wystąpienie błędu

Funkcja WAIT

```
#include <sys/types.h>
#include <sys/wait.h>

PROTOTYPE: int wait( int *status );
           int waitpid( int pid, int *status, int options);

  RETURNS: success : identyfikator procesu potomnego, który
                     się zakończył
           error: -1
                   0 jeśli użyto WNOHANG, a nie było
                     dostępnego żadnego potomka

  errno = ECHILD (potomek pid nie istnieje)
          EPERM  (efektywny id użytkownika nie odpowiada
                  temu z oczekiwanego procesu i nie jest id
                  root'a)
```

PARAMETRY:

1. `status`- status zakończenia procesu (w przypadku zakończenia w sposób normalny) lub numer sygnału w przypadku zabicia potomka lub wartość NULL, w przypadku gdy informacja o stanie zakończenia procesu nie jest istotna
2. `pid` – identyfikator potomka, na którego zakończenie czeka proces macierzysty
   - **pid < -1** oznacza oczekiwanie na dowolny proces potomny, którego ID grupy procesów jest równy modułowi wartości `pid`.
   - **pid = -1** oznacza oczekiwanie na dowolny proces potomny; jest to takie samo zachowanie, jakie stosuje funkcja `wait`.
   - **pid = 0** oznacza oczekiwanie na każdy proces potomny, którego ID grupy procesu jest równe ID wołającego procesu.
   - **pid > 0** oznacza oczekiwanie na potomka, którego ID procesu jest równy wartości `pid`.
3. `options` – jest sumą OR następujących stałych:
   - **WNOHANG** oznacza natychmiastowe zakończenie jeśli potomek się nie zakończył.
   - **WUNTRACED** oznacza zakończenie także dla dzieci, które się zatrzymały, a których status jeszcze nie został zgłoszony.

UWAGI:

Oczekiwanie na zakończenie potomka. Funkcja zwraca identyfikator (`pid`) procesu, który się zakończył. Pod adresem wskazywanym przez status umieszczany jest status zakończenia, który zawiera albo numer sygnału (najmniej znaczące 7 bitów), albo status właściwy(bardziej znaczący bajt).

Jeżeli funkcja `wait` zostanie wywołana w procesie macierzystym przed zakończeniem potomka, wykonywanie proces macierzystego zostanie zawieszone do momentu zakończenia potomka. Jeżeli proces potomny zakończył działanie przed wywołaniem funkcji `wait`, powrót z funkcji `wait` nastąpi natychmiast, a w czasie pomiędzy zakończeniem potomka, a wywołaniem funkcji `wait` przez jego przodka potomek pozostanie w stanie *zombi* (proces taki nazywany jest zombi, upiorem, duchem lub mumią). Jak już wcześniej wspomniano, zombi jest procesem, który zwalnia wszystkie zasoby (nie zajmuje pamięci, nie jest mu przydzielany procesor), zajmuje jedynie miejsce w tablicy procesów w jądrze systemu operacyjnego i zwalnia je dopiero w momencie wywołania funkcji `wait` przez proces macierzysty. Zombi nie jest tworzony, gdy proces macierzysty ignoruje sygnał SIGCLD .
Gdy działa parę procesów potomnych zakończenie jednego z nich powoduje powrót z funkcji `wait`.

Rodzina funkcji EXEC

```
#include <unistd.h>

  PROTOTYPE:
    int execl ( char *path, char *arg0, ..., char *argn,
                char *null );

    int execlp( char *file, char *arg0, ..., char *argn,
                char *null );

    int execv ( char *path, char * argv[] );

    int execvp( char *file, char * argv[] );

    int execle( char * path, char *arg0, ..., char
               *argn,char *null,char *envp[] );

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

  RETURNS:  success  :  wywołanie  programu  podanego  jako
                        parametr
            error: -1
   errno = ECHILD (potomek pid nie istnieje)
           EPERM  (efektywny id użytkownika nie odpowiada
                  temu z oczekiwanego procesu i nie jest id
                  root'a)
```

PARAMETRY:

1. `path, file` – pełna nazwa ścieżkowa lub nazwa pliku z programem
2. `arg0 ...argn`– nazwa i argumenty programu który ma być wywołany

UWAGI:

W ramach istniejącego procesu może nastąpić uruchomienie innego programu w wyniku wywołania jednej z funkcji systemowych `execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`. Funkcje te określane są ogólną nazwą `exec`. Bezbłędne wykonanie funkcji `exec` oznacza **bezpowrotne zaprzestanie wykonywania bieżącego programu i rozpoczęcie wykonywania programu, którego nazwa jest przekazana przez argument.**

W wyniku wywołania funkcji typu `exec` następuje reinicjalizacja segmentów kodu, danych i stosu. Nie zmieniają się atrybuty procesu takie jak pid, ppid, tablica otwartych plików i kilka innych atrybutów z segmentu danych systemowych

Różnice pomiędzy wywołaniami funkcji `exec` wynikają głównie z różnego sposobu budowy ich listy argumentów: w przypadku funkcji `execl` i `execlp` są one podane w postaci listy, a w przypadku funkcji `execv` i `execvp` jako tablica. Zarówno lista argumentów, jak i tablica wskaźników musi być zakończona wartością NULL. Funkcja `execle` dodatkowo ustala środowisko wykonywanego procesu.

Funkcje `execlp` oraz `execvp` szukają pliku wykonywalnego na podstawie ścieżki przeszukiwania podanej w zmiennej środowiskowej PATH. Jeśli zmienna ta nie istnieje, przyjmowana jest domyślna ścieżka `:/bin:/usr/bin`.

Wartością zwrotną funkcji typu `exec` jest status, przy czym jest ona zwracana tylko wtedy, gdy funkcja zakończy się niepoprawnie, będzie to zatem wartość –1.

Funkcje `exec` nie tworzą nowego procesu, tak jak w przypadku funkcji `fork`!!!

PRZYKŁADY

```
execl(„/bin/ls”, „ls”, „-l”,null)
execlp(„ls”, „ls”, „-l”,null)

char* const av[]={„ls”, „-l”, null}
execv(„/bin/ls”, av)

char* const av[]={„ls”, „-l”, null}
execvp(„ls”, av)
```