

Systemy operacyjne 2021-2022

[Strona główna](#) / [Moje kursy](#) / [SQ2021-2022](#) / [Laboratorium 3](#) / [Procesy - materiały pomocnicze](#)

Procesy - materiały pomocnicze

Proces jest pojedynczą instancją wykonującego się programu. Możemy w nim wyróżnić:

- **segment kodu** - zawiera kod binarny aktualnie wykonywanego programu. Znajduje się w nim kod zaimplementowanych przez nas funkcji oraz funkcji dołączanych z bibliotek. Zapisane w tym segmencie adresy funkcji pozwalają na ich lokalizację.
- **segment danych** - zawiera zainicjalizowane zmienne globalne zdefiniowane w programie. Adres segmentu danych można ustalić na podstawie adresu zmiennej globalnej.
- **segment BSS - Block Started by Symbol** - zawiera niezainicjalizowane zmienne globalne
- **segment stosu** - zmienne lokalne oraz adresy powrotu wykorzystywane podczas powrotu z wykonywanej funkcji. Ponieważ proces może pracować w trybie użytkownika lub trybie jądra, każdy z tych trybów ma do dyspozycji oddzielny stos.

Każdemu procesowi przydzielane są zasoby czas procesora, pamięć, dostęp do urządzeń we/wy oraz plików etc). Część tych zasobów jest do wyłącznej dyspozycji procesu, zaś część jest współdzielona z innymi procesami.

Na proces nakładane są pewne ograniczenia dotyczące zasobów systemowych, możemy do nich uzyskać dostęp następującymi funkcjami z **sys/resource.h**:

int getrlimit (int resource, struct rlimit *rlptr) Resource to jedno z makr określające rodzaj zasobu

int setrlimit (int resource, const struct rlimit *rlptr)

struct rlimit {

rlim_t rlim_cur; //bieżące ograniczenie

rlim_t rlim_max; //maksymalne ograniczenie

}

Identyfikatory procesów

Każdy proces w systemie UNIX ma przypisany unikalny identyfikator - **PID**. Jest to 16-bitowa, nieujemna liczba całkowita przypisywana do każdego procesu podczas jego tworzenia. Niektóre identyfikatory są odgórnie zarezerwowane dla specjalnych procesów w systemie, (swapper – 0, *init* – 1 etc).

System UNIX pamięta także identyfikator procesu macierzystego - ta informacja jest zapisywana jako **PPID** (Parent PID).

Do każdego procesu przypisane są również (rzeczywiste) identyfikatory użytkownika (**UID**) oraz grupy (**GID**), określające kto dany proces utworzył. Istnieją również efektywne UID i GID przechowujące informacje o identyfikatorze właściciela oraz grupy właściciela programu.

Do pobrania informacji o identyfikatorach procesu możemy posłużyć się funkcjami z biblioteki `unistd.h`, takimi jak:

- **pid_t getpid(void)** - zwraca PID procesu wywołującego funkcję
- **pid_t getppid(void)** - zwraca PID procesu macierzystego
- **uid_t getuid(void)** - zwraca rzeczywisty identyfikator użytkownika UID
- **uid_t geteuid(void)** - zwraca efektywny identyfikator użytkownika UID
- **gid_t getgid(void)** - zwraca rzeczywisty identyfikator grupy GID
- **gid_t getegid(void)** - zwraca efektywny identyfikator grupy GID

Definicje niezbędnych typów znajdziemy w **sys/types.h**.

Tworzenie procesów

W systemie Unix każdy proces, za wyjątkiem procesu o numerze 0 jest tworzony przez wykonanie przez inny proces funkcji *fork*. Proces ją wykonujący nazywa się **procesem macierzystym**, zaś nowoutworzony - **procesem potomnym**. Procesy, podobnie jak katalogi, tworzą drzewiastą strukturę hierarchiczną - każdy proces w systemie ma jeden proces macierzysty, lecz może mieć wiele procesów potomnych. Korzeniem takiego drzewa w systemie UNIX jest proces o PID równym 1, czyli *init*.

Mechanizm tworzenia procesu w systemach unixowych przedstawiono poniżej:

Funkcje systemowe

Funkcje *fork* oraz *vfork*

- `pid_t fork(void)`

W momencie jej wywołania tworzony jest nowy proces, będący potomnym dla tego, w którym właśnie została wywołana funkcja *fork*. Jest on kopią procesu macierzystego - otrzymuje duplikat obszaru danych, sterty i stosu (a więc nie współdzieli danych). Funkcja *fork* jest wywoływana raz, lecz zwraca wartość dwukrotnie - proces potomny otrzymuje wartość 0, a proces macierzysty PID nowego procesu. Jest to konieczne nie tylko ze względu na możliwość rozróżnienia procesów w kodzie programu: proces macierzysty musi otrzymać PID nowego potomka, ponieważ nie istnieje żadna funkcja umożliwiająca wylistowanie wszystkich procesów potomnych. W przypadku procesu potomnego nie jest konieczne podawanie PID jego procesu macierzystego, ponieważ ten jest określony jednoznacznie (i można go wydobyć np. za pomocą funkcji `getppid`). Z kolei 0 jest bezpieczną wartością, ponieważ jest zarezerwowana dla procesu demona wymiany i nie ma możliwości utworzenia nowego procesu o takim PID.

Po wywołaniu *forka* oba procesy (macierzysty i potomny) kontynuują swoje działanie (od linii następnej po wywołaniu *forka* czyli efektem kodu:

```
#include <stdio.h>

main(){

    printf("Początek\n");

    fork();

    printf("Koniec\n");

}
```

Będzie:

```
Początek //z macierzystego przed wywołaniem forka
Koniec  // z macierzystego lub potomnego po forku
Koniec  //z macierzystego lub potmnego po forku
```

Powyższy komentarz *// z macierzystego lub potomnego po forku* wynika z faktu że nie można przewidzieć, który z procesów będzie wykonywać swoje instrukcje jako pierwszy, dlatego w przypadku gdy wymaga się od nich współpracy, należy zastosować jakieś metody synchronizacji komunikacji międzyprocesowej.

vfork

- `pid_t vfork(void)`

Funkcji tej używa się w przypadku gdy głównym zadaniem nowego procesu jest wywołanie funkcji *exec*. *vfork* „odblokuje” proces macierzysty dopiero w momencie wywołania funkcji *exec* lub *exit*. Inną ważną cechą tej funkcji jest współdzielenie przestrzeni adresowej przez obydwie procesy.

Identyfikacja procesu macierzystego i potomnego

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

int main() {
    pid_t child_pid;
    printf("PID glownego programu: %d\n", (int)getpid());
    child_pid = fork();
    if(child_pid!=0) {
        printf("Proces rodzica: Proces rodzica ma pid:%d\n", (int)getpid());
        printf("Proces rodzica: Proces dziecka ma pid:%d\n", (int)child_pid);
    } else {
        printf("Proces dziecka: Proces rodzica ma pid:%d\n",(int)getppid());
        printf("Proces dziecka: Proces dziecka ma pid:%d\n",(int)getpid());
    }
}
```

```
    return 0;
}
```

Przykładowy wynik działania programu:

```
PID glownego programu: 2359
Proces rodzica: Proces rodzica ma pid:2359
Proces rodzica: Proces dziecka ma pid:2360
Proces dziecka: Proces rodzica ma pid:2359
Proces dziecka: Proces dziecka ma pid:2360
```

Funkcje rodziny `exec`

Funkcje z rodziny `exec` służą do uruchomienia w ramach procesu innego programu.

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

- `int execl(char const *path, char const *arg0, ...)`
funkcja jako pierwszy argument przyjmuje ścieżkę do pliku, następne są argumenty wywołania funkcji, gdzie `arg0` jest nazwą programu
- `int execlp(char const *path, char const *arg0, ..., char const * const *envp)`
podobnie jak `execl`, ale pozwala na podanie w ostatnim argumentcie tablicy ze zmiennymi środowiskowymi
- `int execlp(char const *file, char const *arg0, ...)`
również przyjmuje listę argumentów ale, nie podajemy tutaj ścieżki do pliku, lecz samą jego nazwę, zmienna środowiskowa `PATH` zostanie przeszukana w celu zlokalizowania pliku
- `int execlp(char const *path, char const * const * argv)`
analogicznie do `execl`, ale argumenty podawane są w tablicy
- `int execlp(char const *path, char const * const *argv, char const * const *envp)`
analogicznie do `execl`, również argumenty przekazujemy tutaj w tablicy tablic znakowych
- `int execlp(char const *file, char const * const *argv)`
analogicznie do `execlp`, argumenty w tablicy

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 **execl** 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`.

Znaczenie poszczególnych literek w nazwach funkcji z rodziny `exec`:

- l oznacza, że argumenty wywołania programu są w postaci listy napisów zakończonej zerem (`NULL`)
- v oznacza, że argumenty wywołania programu są w postaci tablicy napisów (tak jak argument `argv` funkcji `main`)
- p oznacza, że plik z programem do wykonania musi się znajdować na ścieżce przeszukiwania ze zmiennej środowiskowej `PATH`
- e oznacza, że środowisko jest przekazywane ręcznie jako ostatni argument

Wartością zwrótną funkcji typu **exec** jest *status*, przy czym jest ona zwracana tylko wtedy, gdy funkcja

zakończy się niepoprawnie, będzie to zatem wartość `-1`.

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)
```

Funkcje `exec` **nie tworzą nowego procesu**, tak jak w przypadku funkcji `fork`. Należy pamiętać, że jeśli w programie wywołamy funkcję `exec`, to kod znajdujący się dalej w programie nie zostanie wykonany, chyba że wystąpi błąd.

Przykład połączenia funkcji `fork` i `exec`

`main.c:`

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

int main() {
    pid_t child_pid;
    child_pid = fork();
    if(child_pid!=0) {
        printf("Ten napis został wyświetlony w programie 'main'!\n");
    } else {
        execvp("./child", NULL);
    }

    return 0;
}
```

`child.c:`

```
#include <stdio.h>

int main() {
    printf("Ten napis został wyświetlony przez program 'child'!\n");
    return 0;
}
```

Wynikiem działania programu jest:

```
Ten napis został wyświetlony w programie 'main'!
Ten napis został wyświetlony przez program 'child'!
```

Funkcje `wait` oraz `waitpid`

Proces macierzysty może się dowiedzieć o sposobie zakończenia bezpośredniego potomka przez wywołanie funkcji systemowej `wait`. Jeśli wywołanie funkcji `wait` nastąpi przed zakończeniem potomka, przodek zostaje zawieszony w oczekiwaniu na to zakończenie. Jeżeli proces macierzysty zakończy działanie przed procesem potomnym, to proces potomny nazywany jest sierotą (ang. orphan) i jest „adoptowany” przez proces systemowy `init`, który staje się w ten sposób jego przodkiem. Jeżeli proces potomny zakończył działanie przed wywołaniem funkcji `wait` w procesie macierzystym, potomek pozostanie w stanie *zombi*. 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, lub w momencie zakończenia procesu macierzystego.

Aby pobrać stan zakończenia procesu potomnego należy użyć jednej z dwóch funkcji (plik nagłówkowy `sys/wait.h`):

```
• pid_t wait ( int *statloc )
```

```
• pid_t waitpid( pid_t pid, int *statloc, int options )
```

Wywołując `wait` lub `waitpid` proces może:

- ulec zablokowaniu (jeśli wszystkie procesy potomne ciągle pracują)
- natychmiast powrócić ze stanem zakończenia potomka (jeśli potomek zakończył pracę i oczekuje na pobranie jego stanu zakończenia)
- natychmiast powrócić z komunikatem awaryjnym (jeśli nie ma żadnych procesów potomnych)

Funkcja `wait` oczekuje na zakończenie dowolnego potomka (do tego czasu blokuje proces macierzysty). Funkcja `waitpid` jest bardziej elastyczna, posiada możliwość określenia konkretnego PID procesu, na który ma oczekiwać, a także dodatkowe opcje (np. nieblokowanie procesu w sytuacji gdy żaden proces potomny się nie zakończył). Argument `pid` należy interpretować w następujący sposób:

- `pid == -1` Oczekiwanie na dowolny proces potomny. W tej sytuacji funkcja `waitpid` jest równoważna funkcji `wait`.
- `pid > 0` Oczekiwanie na proces o identyfikatorze równym `pid`.
- `pid == 0` Oczekiwanie na każdego potomka, którego identyfikator grupy procesów jest równy identyfikatorowi grupy procesów w procesie wywołującym tę funkcję.

- `pid < -1` Oczekiwanie na każdego potomka, którego identyfikator grupy procesów jest równy wartości absolutnej argumentu `pid`.

W obydwu przypadkach statloc to wskaźnik do miejsca w pamięci, gdzie zostanie przekazany status zakończenia procesu potomnego (można go zignorować, przekazując wartość `NULL`).

Kończenie procesów

Istnieje kilka możliwych sposobów na zakończenie procesu:

- zakończenie normalne
 - wywołanie instrukcji **return** w funkcji `main`
 - wywołanie funkcji **exit** - biblioteka `stdlib`
 - wywołanie funkcji **_exit** - biblioteka `unistd`
- zakończenie awaryjne
 - wywołanie funkcji **abort** - generuje sygnał `SIGABORT`
 - **odebranie sygnału**

Funkcje `exit` i `_exit`

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

```
void _exit( int status )
```

Funkcja `_exit` natychmiast kończy działanie programu i powoduje powrót do jądra systemu. Funkcja `exit` natomiast, dokonuje pewnych operacji porządkowych - 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. orphan) i jest „adoptowany” przez proces systemowy `init`). Funkcja `exit` zdefiniowana jest w pliku `stdlib.h`.

Polecenie kill

Polecenie `kill` przesyła sygnał do wskazanego procesu w systemie. Standardowo wywołanie programu powoduje wysyłanie sygnału nakazującego procesowi zakończenie pracy. Proces zapisuje wtedy swoje wewnętrzne dane i kończy pracę. `Kill` może przysyłać procesom różnego rodzaju sygnały. Są to na przykład:

- `SIGTERM` – programowe zamknięcie procesu (15, domyślny sygnał)
- `SIGKILL` – unicestwienie procesu, powoduje utratę wszystkich zawartych w nim danych (9)
- `SIGSTOP` – zatrzymanie procesu bez utraty danych
- `SIGCONT` – wznowienie zatrzymanego procesu

Czasami może zdarzyć się sytuacja, iż proces nie chce się zamknąć sygnałem `SIGTERM`, bo jest przez coś blokowany. Wtedy definitywnie możemy go unicestwić sygnałem `SIGKILL`, lecz spowoduje to utratę danych wewnętrznych procesu.

Ostatnia modyfikacja: wtorek, 19 marca 2019, 11:40

[◀ Zadania laboratorium 2 \(HTML\)](#)

Przejdź do...

[Zadanie - zestaw 3 ▶](#)



Platforma e-Learningowa obsługiwana jest przez:
Centrum e-Learningu AGH oraz Centrum Rozwiązań Informatycznych AGH

[Pobierz aplikację mobilną](#)