# 1 Obsługa błędów

W większości przypadków **wywołanie systemowe** (funkcja systemowa) lub **funkcja biblioteczna** kończąc się **błędem** zwraca wartość $-1$ (czasami `NULL`) i przypisuje zmiennej zewnętrznej `errno` wartość wskazującą rodzaj błędu. Informacje o kodach błędów oraz odpowiadajacych im komunikatach można znaleźć w man errno.

- **Funkcja biblioteczna** `perror`

| Pliki włączane | `<stdio.h>` | | |
|---|---|---|---|
| Prototyp | `void perror(const char *s);` | | |
| Zwracana wartość | Sukces | Porażka | Czy zmienia `errno` |
| | | | Nie |

Funkcja `perror` wypisuje komunikat błędu poprzedzony napisem `*s` i znakiem ':'.
→ UWAGA: Pliki nagłówkowe podaje się względem katalogu `/usr/include/` .

# 2 Procesy

**PROGRAM:** Nieaktywny, statyczny zbiór złożonych w odpowiedniej kolejności instrukcji oraz towarzyszących im danych.

**PROCES:** *Podstawowe pojęcie w Uniksie.*
Abstrakcyjny twór składający się z wykonywanego (działającego) programu oraz bieżących danych o jego stanie i zasobach, za pomocą których system operacyjny steruje jego wykonywaniem. Proces jest jednostką dynamiczną. W Uniksie **procesy** mogą być wykonywane **równolegle** (na jednym procesorze – przełączanie kontekstu) — **wielozadaniowość**.

## 2.1 Identyfikatory związane z procesami

Podstawowe identyfikatory związane z procesami oraz funkcje systemowe służące do ich uzyskiwania:

| Nazwa | Funkcja systemowa | Opis |
|---|---|---|
| UID | `uid_t getuid(void);` | identyfikator użytkownika (rzeczywisty) |
| GID | `gid_t getgid(void);` | identyfikator grupy użytkownika (rzeczywisty) |
| PID | `pid_t getpid(void);` | identyfikator procesu |
| PPID | `pid_t getppid(void);` | identyfikator procesu macierzystego (przodka) |
| PGID | `pid_t getpgid(pid_t pid);` | identyfikator grupy procesów (=`PID` lidera grupy) |
| | `pid_t getpgrp(void);` | ≡ `getpgid(0);` PGID procesu bieżącego |

Powyższe identyfikatory przyjmują wartości **liczb całkowitych nieujemnych**. Jedynie funkcja `getpgid` może zakończyć się błędem – wówczas zwraca wartość `-1` i ustawia zmienną errno.

Pliki nagłówkowe niezbędne dla wywołania powyższych funkcji:
`<sys/types.h>`
`<unistd.h>`

Z poziomu powłoki podstawowe informacje o bieżących procesach można uzyskać przy pomocy komendy `ps`, np. `ps -el` podaje wykaz wszystkich bieżących procesów w tzw. długim formacie (więcej szczegółów w podręczniku systemowym `man`). Podgląd najbardziej aktywnych procesów w czasie rzeczywistym można uzyskać za pomocą komendy `top`.

## 2.2 Tworzenie procesów potomnych – funkcja systemowa fork

| Pliki włączane | `<sys/types.h>, <unistd.h>` | | |
|---|---|---|---|
| Prototyp | `pid_t fork(void);` | | |
| Zwracana wartość | Sukces | Porażka | Czy zmienia `errno` |
| | → 0 w procesie potomnym<br>→ PID procesu potomnego w procesie macierzystym | −1 | Tak |

Funkcja systemowa `fork` tworzy proces potomny, który jest kopią procesu macierzystego.

Typowe wywołanie funkcji `fork`

```
switch (fork())
{
   case -1:
      perror("fork error");
      exit(1);
   case  0:
      /* akcja dla procesu potomnego */
   default:
      /* akcja dla procesu macierzystego, np. wywolanie funkcji wait */
};
```

## 2.3 Kończenie działania procesu – funkcje exit i _exit

| Pliki włączane | `<stdlib.h>` | | |
|---|---|---|---|
| Prototyp | `void exit(int status);` | | |
| Zwracana wartość | Sukces | Porażka | Czy zmienia `errno` |
| | | | Nie |

Jednym ze sposobów zakończenia procesu jest wywołanie funkcji bibliotecznej exit. Funkcja ta wykonuje operacje zakończenia działania procesu i zwraca do procesu macierzystego całkowitoliczbową wartość status, oznaczającą status zakończenia procesu. Zgodnie z konwencją, w przypadku poprawnego zakończenia procesu zwracana jest wartość 0,

a w przypadku błędu wartość niezerowa[1]. Do oznaczania sukcesu czy porażki można użyć stałych: `EXIT_SUCCESS` i `EXIT_FAILURE`, zdefiniowanych w pliku `<stdlib.h>`. Wywołanie funkcji exit powoduje ponadto opróżnienie i zamknięcie wszystkich otwartych strumieni oraz usunięcie wszystkich tymczasowych plików utworzonych przy pomocy funkcji tmpfile. Można zdefiniować własne procedury zakończenia procesu i zerejestrować je przy pomocy funkcji bibliotecznych atexit i/lub on_exit (patrz podręcznik man). Takie procedury zostaną wywołane przez funkcję exit w kolejności odwrotnej do kolejności ich rejestracji. Pozwala to m.in. na opróżnienie wszystkich buforów standardowej bibliotece wejścia-wyjścia.

| Pliki włączane | `<unistd.h>` | | |
|---|---|---|---|
| Prototyp | `void _exit(int status);` | | |
| Zwracana wartość | Sukces | Porażka | Czy zmienia `errno` |
| | | | Nie |

Funkcja _exit różni się od poprzedniej przede wszystkim tym, że jest *wywołaniem systemowym*, a nie funkcją biblioteki języka C. Powoduje ona natychmiastowe zakończenie procesu. Wszystkie otwarte deskryptory plików należące do procesu są zamykane, wszystkie jego procesy potomne są ,,adoptowane"przez proces init, a do procesu macierzystego wysyłany jest sygnał SIGCHLD. Znaczenie parametru status jest takie jak dla funkcji exit. W odróżnieniu od exit, nie wywołuje ona żadnych procedur zarejestrowanych przez funkcje atexit lub on_exit. Natomiast to czy opróżnia standardowe bufory wejścia-wyjścia oraz czy usuwa pliki tymczasowe stworzone przy użyciu funkcji tmpfile, jest zależne od implementacji. Generalnie zaleca się używanie funkcji exit w procesie macierzystym (za wyjątkiem przypadku tworzenia procesów *demonów*), natomiast funkcji _exit w procesach potomnych (by uniknąć efektów ubocznych).

## 2.4 Czekanie na procesy potomne – funkcja systemowa wait

W systemie UNIX na każdy proces, za wyjątkiem procesu init (o identyfikatorze `PID = 1`), powinien czekać jakiś proces macierzysty. Proces, który się zakończył, ale na który nie czekał żaden inny proces nazywa się *zombi*. Proces-zombi nic nie robi, ale zajmuje miejsce w systemowej tabeli procesów. Aby uniknąć powstawania procesów-zombi, w Uniksie procesy ,,sieroty" są ,,adoptowane" przez proces init, który w odniesieniu do nich wykonuje operacje czekania.

| Pliki włączane | `<sys/types.h>, <sys/wait.h>` | | |
|---|---|---|---|
| Prototyp | `pid_t wait(int *stat_loc);` | | |
| Zwracana wartość | Sukces | Porażka | Czy zmienia `errno` |
| | PID procesu potomnego | $-1$ | Tak |

[1]Faktycznie zwracanych jest tylko pierwszych osiem bitów, zatem zwracane wartości należą do przedziału [0,255].

Do oczekiwania na proces potomny służy funkcja systemowa wait. Zawiesza ona działanie procesu macierzystego do momentu zakończenia się *pierwszego* procesu potomnego[2]. Informacje o stanie potomka zwracane są przez parametr stat_loc (tylko dwa młodsze bajty są używane). Jeśli proces potomny zakończył się normalnie, to najmłodszy bajt będzie równy 0, a następny będzie zawierał kod powrotu. W przypadku zakończenia procesu potomnego na skutek sygnału, najmłodszy bajt będzie zawierał numer sygnału, a następny wartość 0 (w przypadku wygenerowania zrzutu pamięci *core*, najstarszy bit najmłodszego bajtu będzie ustawiony na 1). Gdy parametr funkcji wait będzie ustawiony na NULL, to stan procesu potomnego nie zostanie zwrócony. Jeżeli dany proces nie ma procesów potomnych, to funkcja kończy się błędem i ustawia zmienną errno na ECHILD.

| | | | |
|---|---|---|---|
| Pliki włączane | `<sys/types.h>, <sys/wait.h>` | | |
| Prototyp | `pid_t waitpid(pid_t pid, int *stat_loc, int options);` | | |
| Zwracana wartość | Sukces | Porażka | Czy zmienia `errno` |
| | PID procesu potomnego lub 0 | $-1$ | Tak |

Lepszą funkcjonalność niż funkcja wait dostarcza funkcja waitpid. Funkcji tej można wskazać konkretny proces czy też grupę procesów, na które ma czekać. Jeżeli argument pid > 0 i argument `options = 0`, to funkcja zablokuje wywołujący ją proces do czasu zakończenia procesu potomnego o PID = pid. Znaczenie parametru stat_loc jest takie jak dla funkcji wait. Więcej szczegółów można znaleźć w podręczniku systemowym man.

## ĆWICZENIE 1: Procesy Potomne: fork

(1) Napisać program wypisujący identyfikatory UID, GID, PID, PPID i PGID dla danego procesu.
(2) Wywołać funkcję `fork` 3 razy (np. w pętli) i wypisać powyższe identyfikatory dla wszystkich procesów potomnych.
(3) Wstawić funkcję `sleep` tak, aby procesy pojawiały się na ekranie grupowane pokoleniami od najstarszego do najmłodszego.
(4) Na podstawie wyników programów narysować ,,drzewo genealogiczne" tworzonych procesów (z zaznaczonymi identyfikatorami).
→ *Ile powstaje procesów i dlaczego?*

▷ **Funkcja** sleep:

| | | | |
|---|---|---|---|
| Pliki włączane | `<unistd.h>` | | |
| Prototyp | `unsigned sleep(unsigned seconds);` | | |
| Zwracana wartość | Sukces | Porażka | Czy zmienia `errno` |
| | Liczba nieprzespanych sekund | | Nie |

[2]Uwaga: Funkcja wait czeka na zakończenie tylko jednego procesu potomnego – tego, który zakończy się najwcześniej. W celu oczekiwania na zakończenie kolejnego procesu, trzeba ją wywołać ponownie.

## 2.5 Uruchamianie programów – funkcja systemowa exec

Funkcja systemowa `exec` służy do ponownego **zainicjowania procesu** na podstawie **wskazanego programu**. Jest sześć odmian funkcji `exec` zgrupowanych w dwie rodziny (po trzy funkcje). Rodziny różnią się postacią argumentów: litera `l` w nazwie oznacza argumenty w postaci listy, a litera `v` – w postaci tablicy (ang. *vector*). Poniżej omawiamy po jednym przedstawicielu każdej z rodzin.

| Pliki włączane | `<unistd.h>` | | |
|---|---|---|---|
| Prototyp | `int execl(const char *path, const char *arg0, ..., const char *argn, char *null);`<br>`int execv(const char *path, char *const argv[ ]);` | | |
| Zwracana wartość | Sukces | Porażka | Czy zmienia `errno` |
| | Nic nie zwraca („popełnia samobójstwo”) | $-1$ | Tak |

Argumenty funkcji `exec`:

| | |
|---|---|
| `path` | ścieżkowa nazwa pliku (wykonawczego) zawierającego program; |
| `arg0` | argument zerowy: nazwa pliku (wykon.) zawierającego program; |
| `arg1,...,argn` | argumenty wywołania programu; |
| `null` | wskaźnik `NULL`; |
| `argv[ ]` | adres tablicy wskaźników na ciągi znaków będące argumentami przekazywanymi do wykonywanego programu (ostatnim elementem powinien być `NULL`). |

*Najczęściej funkcję* `exec` *wywołuje się w połączeniu z funkcją* `fork`.

Typowe wywołanie `fork` i `exec`

```
switch (fork())
{
   case -1:
      perror("fork error");
      exit(1);
   case  0:  /* proces potomny */
      execl("./nowy_program.x","nowy_program.x",NULL);
      perror("execl error");
      _exit(2);
   default:  /* proces macierzysty */
};
```

**ĆWICZENIE 2:** Uruchamianie Programów: exec

Zmodyfikować poprzedni program tak, aby komunikaty procesów potomnych były wypisywane przez program uruchamiany przez funkcję `exec`.
→ *Ile teraz powstaje procesów i dlaczego?*