

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 odpowiadających 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 <code>errno</code>
			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 ułożonych 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 wiele **procesów** może być wykonywane **współbieżnie** (na jednym procesorze dzięki podziałowi czasu i przełączaniu kontekstu) – nazywa się to **wielozadaniowością**.

→ UWAGA: **Proces** może wykonywać *co najwyżej jeden* **program**, natomiast ten sam **program** może być wykonywany w *dowolnej liczbie* **procesów**.

2.1 Identyfikatory związane z procesami

Podstawowe identyfikatory związane z procesami oraz funkcje systemowe służące do ich uzyskiwania przedstawione są w tabeli 1. Identyfikatory te przyjmują wartości **liczb całkowitych nieujemnych**. W Uniksie procesy mogą być łączone w grupy – każdy proces należący do takiej grupy posiada ten sam identyfikator grupy procesów `PGID`, który jest równy identyfikatorowi procesu `PID` lidera tej grupy. Spośród podanych funkcji jedynie `getpgid` może zakończyć się błędem – wówczas zwraca wartość `-1` i ustawia zmienną `errno`.

Z poziomu powłoki podstawowe informacje o bieżących procesach można uzyskać przy pomocy komendy `ps`, np. `ps -el` podaje wykaz wszystkich bieżących procesów w tzw.

Pliki włączane		<sys/types.h>, <unistd.h>
Nazwa	Funkcja systemowa	Opis
UID	<code>uid_t getuid(void);</code>	identyfikator użytkownika (rzeczywisty)
GID	<code>gid_t getgid(void);</code>	identyfikator grupy użytkownika (rzeczywisty)
PID	<code>pid_t getpid(void);</code>	identyfikator procesu
PPID	<code>pid_t getppid(void);</code>	identyfikator procesu macierzystego (przodka)
PGID	<code>pid_t getpgid(pid_t pid);</code> <code>pid_t getpgrp(void);</code>	identyfikator grupy procesów (=PID lidera grupy) ≡ <code>getpgid(0)</code> ; PGID procesu bieżącego

Tablica 1: Identyfikatory związane z procesami oraz funkcje służące do ich uzyskiwania

długim formacie (więcej szczegółów w podręczniku systemowym: `man ps`). Podgląd najbardziej aktywnych procesów w czasie rzeczywistym, wraz z wieloma szczegółami na temat używanych przez nie zasobów systemowych, można uzyskać za pomocą komendy `top`.

Pliki włączane	<sys/types.h>, <unistd.h>		
Prototyp	<code>pid_t setpgid(pid_t pid, pid_t pgrp);</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	0	-1	Tak

Identyfikator grupy procesów PGID można zmienić. Do tego celu służy funkcja `setpgid` przedstawiona w powyższej tabeli. Umożliwia ona nadanie identyfikatorowi grupy procesów procesu lidera o identyfikatorze `pid` wartości `pgrp`. Jeżeli przekazana zostanie wartość `pid` równa zero, to zmieniony zostanie identyfikator PGID bieżącego procesu. Wartość `pgrp` odpowiada numerowi grupy, do której proces będzie od tej chwili należał. Jeśli wartością `pgrp` będzie zero, to proces o identyfikatorze równym `pid` stanie się liderem nowej grupy. Tak zwykle się dzieje kiedy uruchamiamy jakieś polecenie z poziomu powłoki Uniksa – wówczas tworzony jest nowy proces i dla niego wywoływana jest funkcja `setpgid` z wartością `pgrp` równą 0 i proces ten staje się liderem nowej grupy procesów. Funkcja ta może być skutecznie wywołana tylko kiedy wywołujący ją proces ma odpowiednie uprawnienia, np. jest procesem macierzystym procesu docelowego. W przypadku sukcesu, funkcja `setpgid` zwraca wartość 0, a w przypadku porażki wartość -1 i ustawia zmienną `errno` na odpowiedni kod błędu.

→ UWAGA: Nie należy mylić identyfikatora „grupy procesów” PGID z identyfikatorem „grupy użytkownika” procesu GID!

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 → 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. wywołanie 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 niezerowa¹. 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 zarejestrować je przy pomocy

¹Faktycznie zwracanych jest tylko osiem mniej znaczących bitów wartości `status`, zatem zwracane wartości należą do przedziału `[0,255]`.

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 biblioteki wejścia-wyjścia.

Pliki włączane	<stdlib.h>		
Prototyp	<code>void atexit(void (*function)(void));</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	0	$\neq 0$	Tak

Przedstawiona w powyższej tabeli funkcja `atexit` pozwala zarejestrować funkcję, która będzie wywołana przy normalnym zakończeniu procesu – albo poprzez wywołanie funkcji `exit`, albo przez `return` z funkcji `main`. Tak zarejestrowane funkcje są wywoływane w kolejności odwrotnej do ich rejestracji; żadne argumenty nie są przekazywane. W funkcji zarejestrowanej przez `atexit` nie powinno się wywoływać funkcji `exit` – takie wywołanie jest niezdefiniowane w standardzie POSIX i może dawać różne efekty, zależne od implementacji. Typowo można zarejestrować do 32 funkcji, ale faktyczny limit zależy od implementacji (patrz `man atexit`). W ten sposób można automatycznie wywoływać funkcje, które robią określone porządki w momencie zakończenia procesu. Oto prosty przykład użycia tej funkcji:

```
...
void koniec(void)
{
    printf("Koniec procesu - porzadki zrobione!\n");
}
...
int main()
{
    ...
    if (atexit(koniec) != 0) {
        perror("atexit error");
        exit(EXIT_FAILURE);
    }
    ...
    return 0;
}
```

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

Funkcja `_exit` różni się od `exit` 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 bufor 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 *zombie*. Proces-zombie nic nie robi, ale zajmuje miejsce w systemowej tabeli procesów. Aby uniknąć powstawania procesów *zombie*, w Uniksie procesy „sieroty” są „adoptowane” przez proces `init`, który w odniesieniu do nich wykonuje operacje czekania (wówczas dla takiego procesu `PPID = 1`). W nowszych wersjach systemu Linux osierocone procesy potomne mogą być adoptowane przez specjalny proces *demon* o nazwie `systemd`, który może mieć `PID > 1` (sprawdzić poleceniem `ps tree -p`). Natomiast w systemie macOS proces ten nazywa się `launchd` i ma `PID = 1`.

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

Do oczekiwania na proces potomny służy funkcja systemowa `wait`. Zawiesza ona działanie procesu macierzystego do momentu zakończenia się *pierwszego* procesu potomnego². 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`.

Lepszą funkcjonalność niż funkcja `wait` dostarcza funkcja `waitpid`. Funkcji tej można wskazać konkretny proces czy też grupę procesów, na które ma czekać. Jeżeli argument

²UWAGA: 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*.

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

`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`.

2.5 Usypianie procesu – funkcja systemowa `sleep`

Użyteczną funkcją jest też funkcja `sleep`, która wstrzymuje („usypia”) wywołujący ją proces na wskazaną przez parametr `seconds` liczbę sekund. Kiedy upłynie podany czas usypienia procesu, to funkcja zwraca wartość 0, natomiast kiedy jej wykonanie zostanie przerwane (np. na skutek otrzymania sygnału), to zwraca liczbę nieprzespanych sekund.

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

ĆWICZENIE 1: PROCESY POTOMNE: FORK

- Napisać program wypisujący identyfikatory `UID`, `GID`, `PID`, `PPID` i `PGID` dla danego procesu.
- Wywołać funkcję `fork` trzy razy (najlepiej w pętli `for`) i wypisać powyższe identyfikatory dla procesu macierzystego oraz wszystkich procesów potomnych. Przy pomocy funkcji `wait` sprawić, aby proces macierzysty zaczął na zakończenie wszystkich procesów potomnych.
- Jak w (b), tylko przy użyciu funkcji `sleep` (→ nie używać funkcji `wait`) sprawić by procesy potomne były adoptowane przez proces `init` lub `systemd`; poleceniem `ps tree -p` z poziomu powłoki wyświetlić drzewo procesów w danym systemie i zidentyfikować proces adoptujący osierocone procesy.
- Jak w (b), tylko wstawić funkcję `sleep` w takich miejscach programu, aby procesy pojawiały się na ekranie grupowane pokoleniami od najstarszego do najmłodszego, a proces macierzysty kończył się dopiero po procesach potomnych (→ nie używać funkcji `wait`). Na podstawie wyników programu sporządzić (w pliku tekstowym) „drzewo genealogiczne” tworzonych procesów z zaznaczonymi identyfikatorami `PID`, `PPID` i `PGID`.
- Jak w (b), tylko przy użyciu funkcji `setpgid` sprawić by każdy proces potomny stawał się liderem swojej własnej grupy procesów.
→ Ile procesów powstanie przy n -krotnym wywołaniu funkcji `fork` i dlaczego?

2.6 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	<pre>int execl(const char *path, const char *arg0, ..., const char *argn, char *null); int execv(const char *path, char *const argv[]);</pre>		
Zwracana wartość	Sukces	Porażka	Czy zmienia errno
	Nic nie zwraca („popelnia 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
        execlp("./nowy_program.x", "nowy_program.x", NULL);
        perror("execlp error");
        _exit(2);
    default: // proces macierzysty
};
```

ĆWICZENIE 2: URUCHAMIANIE PROGRAMÓW: EXEC

Zmodyfikować program z **ćwiczenia 1b**, tak aby komunikaty procesów potomnych były wypisywane przez program uruchamiany przez funkcję **execlp**. Nazwę programu do uruchomienia przekazywać przez argumenty programu procesu macierzystego.

→ Ile procesów powstanie przy *n*-krotnym wywołaniu funkcji **fork-exec** jak wyżej i dlaczego?