# 3 Procesy

Pojęcie *procesu* jest kluczowe dla zrozumienia funkcjonowania wielozadaniowych systemów operacyjnych. Trudność w zrozumieniu tego pojęcia i tym samym zgłębienie mechanizmu obsługi procesów systemu operacyjnego wynika z faktu, że pojęcie procesu utożsamiane jest często z pojęciem programu, który wykonywany jest w ramach procesu. Poza tym samo pojęcie procesu jest dość trudne do zdefiniowania, gdyż z punktu widzenie programisty proces jest pojęciem dość abstrakcyjnym.

W celu przybliżenia pojęcia *proces* należy sobie uświadomić, jakie zasoby sprzętowe i programowe systemu komputerowego są niezbędne do wykonania programu. Do zasobów sprzętowych warunkujących wykonanie programu należą:

- procesor — wykonuje instrukcje zawarte w programie,
- pamięć operacyjna — przechowuje dane do przetworzenia, tymczasowe dane pomocnicze, wyniki pośrednie oraz końcowe, a także zgodnie z architekturą von Neumana — program, czyli ciąg instrukcji,
- urządzenia zewnętrzne — służą do komunikacji ze światem zewnętrznym, czyli umożliwiają pobieranie danych do przetwarzania oraz składowanie i prezentację wyników działania programów.

Oprócz wymienionych zasobów sprzętowych, istotne też są zasoby programowe, tworzone i udostępniane najczęściej przez system operacyjny. Do tej grupy zasobów należą np. pliki, mechanizmy komunikacji między procesami, mechanizmy synchronizacji procesów itp.

Zasoby systemu komputerowego, nadzorowanego przez wielozadaniowy system operacyjny, mogą być wykorzystywane na potrzeby współbieżnego wykonania wielu programów (w szczególności może to być kilka różnych wykonań tego samego programu). Na potrzeby każdego wykonania należy przydzielić odpowiednie zasoby, a zadaniem systemu operacyjnego jest rozwiązywanie konfliktów w dostępie do tych zasobów w przypadku, gdy są one jednocześnie potrzebne dla kilku wykonań. W celu właściwej ewidencji użytkowania zasobów, czyli ich przydziału, kontroli ich wykorzystania oraz odzyskiwania, synchronizacji dostępu do nich itp., wprowadzone zostało właśnie pojęcie procesu, oznaczające wykonanie programu.

Program jest zatem rozumiany jako ciąg instrukcji do wykonania przez procesor, do wykonania przez interpreter lub do dalszego przetworzenia na inny ciąg instrukcji (do kompilacji). Proces natomiast jest abstrakcyjnym pojęciem, określającym program w trakcie wykonywania wraz z niezbędnymi do tego wykonania zasobami systemu komputerowego. Proces identyfikuje zatem przetwarzanie danych, realizowane przez system komputerowy.

## 3.1 Identyfikacja procesów w systemie UNIX

Każdy proces w systemie musi mieć swój unikalny identyfikator, odróżniający go od innych procesów i umożliwiający jednoznaczne wskazanie konkretnego procesu, gdy wymagają tego mechanizmy systemu operacyjnego. W środowisku systemu operacyjnego UNIX powszechnie używanym skrótem terminu *identyfikator procesu* jest PID (ang. Process IDentyfier). Jednym z atrybutów procesu jest też identyfikator jego przodka, określany skrótem PPID.

PID jest liczbą całkowitą typu `pid_t` (w praktyce `int`). Każdy nowo tworzony proces otrzymuje jako identyfikator kolejną liczbę typu `pid_t`, a po dojściu do końca zakresu liczb danego typu przydział rozpoczyna się od początku. Przydzielane są oczywiście tylko identyfikatory nie używane w danej chwili przez inne procesy.

PID danego procesu nie może ulec zmianie, można go natomiast łatwo uzyskać wywołując funkcję systemową `getpid`. Identyfikator przodka można uzyskać za pomocą funkcji `getppid`.

`pid_t getpid(void)` — uzyskanie własnego identyfikatora (PID) przez proces. Funkcja zwraca identyfikator procesu lub –1 w przypadku błędu.

`pid_t getppid(void)` — uzyskanie identyfikatora procesu macierzystego (PPID) przez potomka. Funkcja zwraca identyfikator procesu lub –1 w przypadku błędu.

Procesy powstają na potrzeby realizacji zadań, zlecanych przez użytkowników, w związku z czym działają one na rzecz tych użytkowników i z użytkownikiem związana jest domena ochrony w systemie UNIX [3]. Dla autoryzacji dostępu kluczowe są zatem dwa identyfikatory: *identyfikatora użytkownika* i *identyfikator grupy*, do której dany użytkownik należy. Identyfikator te dziedziczone są od przodka i jeśli są to identyfikatory zwykłego użytkownika, to nie mogą być zmieniane. Właściciel procesu nie może więc ulec zmianie.

Trwałe związanie praw dostępu procesów do zasobów systemu (np. do plików) może czasami nadmiernie ograniczać dostępność istotnych dla danego procesu informacji. Typowym przykładem może być proces zmiany hasła użytkownika. Proces taki musi mieć dostęp do pliku `/etc/shadow`, do którego zwykły użytkownik nie ma żadnych praw (nawet prawa odczytu!). Modyfikacja pliku `/etc/shadow` na zlecenie zwykłego użytkownika byłaby więc niemożliwa.

W systemie UNIX wyróżniono zatem dwa rodzaje identyfikatorów: *rzeczywisty* (ang. real) i *obowiązujący* (ang. effective). Rzeczywisty identyfikator użytkownika oraz rzeczywisty identyfikator grupy identyfikują właściciela procesu i w przypadku procesu użytkownika zwykłego nie ulegają zmianie przez cały czas życia procesu. Obowiązujący identyfikator użytkownika oraz obowiązujący identyfikator grupy decydują o prawach dostępu, tzn. te identyfikatory brane są pod uwagę przy autoryzacji dostępu do zasobów. Proces ma więc takie prawa dostępu, jakie ustawione są dla użytkownika o identyfikatorze obowiązującym lub dla grupy o identyfikatorze obowiązującym.

Jak już wcześniej wspomniano, identyfikatory użytkownika i grupy dziedziczone są od przodka i najczęściej identyfikatory obowiązujące równe są rzeczywistym. Zmiana obowiązującego identyfikatora użytkownika następuje w wyniku uruchomienia w procesie programu z pliku wykonywalnego, w którym ustawiono *bit zmiany obowiązującego identyfikatora użytkownika*. Obowiązujący identyfikator użytkownika przyjmuje wartość identyfikatora właściciela pliku zawierającego program. Podobnie, zmiana obowiązującego identyfikatora grupy następuje, jeśli w pliku z programem w którym ustawiono *bit zmiany obowiązującego identyfikatora grupy*, a obowiązujący identyfikator grupy przyjmuje wartość identyfikatora grupy pliku z programem.

Kolejnym atrybutem procesu jest identyfikator grupy procesów, do której dany proces należy. Grupy procesów definiowane są na potrzeby przekazywania sygnałów. Identyfikator grupy procesów jest PID'em lidera tej grupy. Proces uruchamiany w reakcji na wydanie polecenia przez użytkownika należy do grupy procesu powłoki. Proces może odłączyć się od swojej grupy poprzez utworzenie własnej grupy, dla której będzie liderem. Do grupy tej wejdą wszystkie nowo utworzone procesy potomne, o ile któryś z potomków nie utworzy własnej grupy.

## 3.2 Obsługa procesów w systemie UNIX

W zakresie obsługi procesów system UNIX udostępnia mechanizm tworzenia nowych procesów, usuwania procesów oraz uruchamiania programów. Każdy proces, z wyjątkiem procesu systemowego o identyfikatorze 0, tworzony jest przez jakiś inny proces, który staje się jego *przodkiem* zwanym też *procesem macierzystym*, *procesem rodzicielskim* lub krótko *rodzicem* (ang. parent). Nowo utworzony proces nazywany jest *potomkiem* lub *procesem potomnym* (child). Procesy w systemie UNIX tworzą zatem drzewiastą strukturę hierarchiczną, podobnie jak katalogi.

### 3.2.1 Tworzenie procesu

Potomek tworzony jest w wyniku wywołania przez przodka funkcji systemowej `fork`. Po utworzeniu potomek kontynuuje wykonywanie programu swojego przodka od miejsca wywołania funkcji `fork`.

`pid_t fork(void)` — utworzenie procesu potomnego. W procesie macierzystym funkcja zwraca identyfikator (pid) procesu potomnego (wartość większą od 0, w praktyce większą od 1), a

w procesie potomnym wartość 0. W przypadku błędu funkcja zwraca –1, a proces potomny nie jest tworzony.

Listingi 1 przedstawia program, który ma zasygnalizować początek i koniec swojego działania przez wyprowadzenia odpowiedniego tekstu na standardowe wyjście.

Listing 1: Przykład działania funkcji `fork`

```
#include <stdio.h>
2
main(){
4    printf("Poczatek\n");
     fork();
6    printf("Koniec\n");
}
```

**Opis programu:** Program jest początkowo wykonywany przez jeden proces. W wyniku wywołania funkcji systemowej `fork` (linia 5) następuje rozwidlenie i tworzony jest proces potomny, który kontynuuje wykonywanie programu swojego przodka od miejsca utworzenia. Inaczej mówiąc, od momentu wywołania funkcji `fork` program wykonywany jest przez dwa współbieżne procesy. Wynik działania programu jest zatem następujący:

```
Poczatek
Koniec
Koniec
```

### 3.2.2 Uruchamianie programu

W ramach istniejącego procesu może nastąpić uruchomienie innego programu wyniku wywołania jednej z funkcji systemowych `execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`. Funkcje te będą określane ogólną nazwą `exec`. Uruchomienie nowego programu oznacza w rzeczywistości zmianę programu wykonywanego dotychczas przez proces, czyli zastąpienie wykonywanego programu innym programem, wskazanym odpowiednio w parametrach aktualnych funkcji `exec`.

```
int execl(const char *path, const char *arg, ...)
int execlp(const char *file, const char *arg, ...)
int execle(const char *path, const char *arg , ..., char *const envp[])
int execv(const char *path, char *const argv[])
int execvp(const char *file, char *const argv[])
int execve(const char *file, char *const argv[], char *const envp[])
```

Uruchomienie programu w ramach procesu. Funkcja zwraca –1 w przypadku błędu lub nic nie zwraca ze względu na zmianę programu w przypadku poprawnego zakończenia.

**Opis parametrów:**

| | |
|---|---|
| *path* | nazwa ścieżkowa pliku z programem (w przypadku podania samej nazwy przyjmuje się, że jest to nazwa w katalogu bieżącym), |
| *file* | nazwa pliku z programem, |
| *arg* | argument łinii poleceń", |
| *argv* | wektor (tablica) argumentów łinii poleceń", |
| *envp* | wektor zmiennych środowiskowych. |

Bezbłędne wykonanie funkcji `exec` oznacza zatem bezpowrotne zaprzestanie wykonywania bieżącego programu i rozpoczęcie wykonywania programu, którego nazwa jest przekazana jako argument. W konsekwencji, z funkcji systemowej `exec` nie ma powrotu do programu, gdzie nastąpiło jej wywołanie, o ile wykonanie tej funkcji nie zakończy się błędem. Wyjście z funkcji `exec` można więc traktować jako jej błąd bez sprawdzania zwróconej wartości.

Listingi 2 przedstawia program, który — podobnie jak program na listingu 1 — ma zasygnalizować początek i koniec swojego działania, przy czym w programie następuje uruchomienie innego programu, znajdującego się w pliku o nazwie `ls`.

Listing 2: Przykład działania funkcji `exec`

```
#include <stdio.h>

main(){
   printf("Poczatek\n");
   execlp("ls", "ls", "-a", NULL);
   printf("Koniec\n");
}
```

**Opis programu:** W wyniku wywołania funkcji systemowej `execlp` (linia 5) następuje zmiana wykonywanego programu, zanim sterowanie dojdzie do instrukcji wyprowadzenia napisu `Koniec` na standardowe wyjście (linia 6). Zmiana wykonywanego programu powoduje, że sterowanie nie wraca już do poprzedniego programu i napis `Koniec` nie pojawia się na standardowym wyjściu w ogóle.

Listing 3 przedstawia program, w którym zastosowano zarówno funkcję `fork` do utworzenia procesu, jak i funkcję `execlp` do uruchomienia innego programu w procesie potomnym.

Listing 3: Przykład uruchamiania programów bez synchronizacji przodka z potomkiem

```
#include <stdio.h>

main(){
   printf("Poczatek\n");
   if (fork() == 0){
      execlp("ls", "ls", "-a", NULL);
      perror("Blad uruchmienia programu");
      exit(1);
   }
   printf("Koniec\n");
}
```

**Opis programu:** Zmiana wykonywanego programu przez wywołanie funkcji `execlp` (linia 6) odbywa się tylko w procesie potomnym, tzn. wówczas, gdy wywołana wcześniej funkcja `fork` zwróci wartość 0 (linia 5). Funkcja `fork` zwraca natomiast 0 tylko procesowi potomnemu. Ponieważ przodek i potomek działają współbieżnie, nie ma żadnych gwarancji, że przodek wyświetli napis `Koniec` po zakończeniu procesu potomnego. Wynik działania może się zatem nieco różnić od oczekiwań.

### 3.2.3 Zakończenie procesu

Proces może się zakończyć dwojako: w sposób normalny, tj. przez wywołanie funkcji systemowej `exit` lub w sposób awaryjny, czyli przez wywołanie funkcji systemowej `abort` lub w wyniku reakcji na sygnał. Funkcja systemowa `exit` wywoływana jest niejawnie na końcu wykonywania programu przez proces lub może być wywołana jawnie w każdym innym miejscu programu. Funkcja kończy proces i powoduje przekazanie w odpowiednie miejsce tablicy procesów kodu wyjścia, czyli wartości, która może zostać odebrana i zinterpretowana przez proces macierzysty.

`void exit(int` *status*`)` — zakończenie procesu.

**Opis parametrów:**

*status* kod wyjścia przekazywany procesowi macierzystemu.

Zakończenie procesu w wyniku otrzymania sygnału nazywane jest zabiciem. Proces może otrzymać sygnał wysłany przez jakiś inny proces (również przez samego siebie) za pomocą funkcji systemowej `kill` lub wysłany przez jądro systemu operacyjnego.

Proces macierzysty może się dowiedzieć o sposobie zakończenie bezpośredniego potomka przez wywołanie funkcji systemowej `wait`. Jeśli wywołanie funkcji `wait` nastąpi przed zakończeniem potomka, przodek zostaje zawieszony w oczekiwaniu na to zakończenie. Po zakończeniu potomka w procesie macierzystym następuje wyjście z funkcji `wait`, przy czym pod adresem wskazanym w parametrze aktualnym umieszczony zostanie *status zakończenia*, który zawiera albo numer sygnału (najmniej znaczące 7 bitów), albo kod wyjścia (bardziej znaczący bajt), przekazany przez potomka jako wartość parametru funkcji `exit`. Najbardziej znaczący bit młodszego bajtu wskazuje, czy nastąpił zrzut zawartości pamięci, czyli czy został utworzony plik `core`.

`pid_t wait(int *`*status*`)` — oczekiwanie na zakończenie potomka. Funkcja zwraca identyfikator (pid) procesu potomnego, który się zakończył lub –1 w przypadku błędu.

**Opis parametrów:**

*status* adres słowa w pamięci, w którym umieszczony zostanie status zakończenia.

Listing 4 przedstawia program, w którym obok mechanizmu `fork-exec`, zastosowanego w przykładzie na listingu 3 do uruchomienia programu w procesie potomnym, użyto również funkcji `wait` do zsynchronizowania procesu macierzystego z potomnym. Synchronizacja obu procesów polega w tym przykładzie na zablokowaniu procesu macierzystego do momentu zakończenia wykonywania programu przez potomka. W rezultacie poniższy program sygnalizuje koniec swojego działania zgodnie z oczekiwaniami, tzn. napis `Poczatek` pojawia się przed wynikiem wykonania programu (polecenia) `ls`, a napis `Koniec` pojawia się po zakończeniu wykonywania `ls`.

Listing 4: Przykład uruchamiania programów z synchronizacją przodka z potomkiem

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    if (fork() == 0){
       execlp("ls", "ls", "-a", NULL);
       perror("Blad uruchmienia programu");
       exit(1);
    }
    wait(NULL);
    printf("Koniec\n");
}
```

**Opis programu:** Podobnie jak w przypadku programu w lisingu 3, zmiana wykonywanego programu przez wywołanie funkcji `execlp` (linia 6) odbywa się tylko w procesie potomnym. W tym przypadku jednak, w celu uniknięcia sytuacji, w której proces macierzysty wyświetli napis `Koniec` zanim nastąpi wyświetlenie listy plików, proces macierzysty wywołuje funkcję `wait`. Funkcja ta powoduje zawieszenie wykonywania procesu macierzystego do momentu zakończenie potomka.

W powyższym programie (listing 4), jak również w innych programach w tym rozdziale założono, że funkcje systemowe wykonują się bez błędów. Program na listingu 5 jest modyfikacją poprzedniego programu, polegającą na sprawdzaniu poprawności wykonania funkcji systemowych.

Listing 5: Przykład uruchamiania programów z kontrolą poprawności

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    switch (fork()){
       case -1:
```

```
            perror("Blad utworzenia procesu potomnego");
            break;
        case 0: /* proces potomny */
            execlp("ls", "ls", "-a", NULL);
            perror("Blad uruchmienia programu");
            exit(1);
        default: /* proces macierzysty */
            if (wait(NULL) == -1)
                perror("Blad w oczekiwaniu na zakonczenie potomka");
    }
    printf("Koniec\n");
}
```

Listingi 6 i 7 przedstawiają program, którego zadaniem jest zademonstrować wykorzystanie funkcji `wait` do przekazywania przodkowi przez potomka *statusu zakończenia procesu*.

Listing 6: Przykład działania funkcji `wait` w przypadku naturalnego zakończenia procesu

```
#include <stdio.h>

main(){
    int pid1, pid2, status;

    pid1 = fork();
    if (pid1 == 0) /* proces potomny */
        exit(7);
    /* proces macierzysty */
    printf("Mam potomka o identyfikatorze %d\n", pid1);
    pid2 = wait(&status);
    printf("Status zakonczenia procesu %d: %x\n", pid2, status);
}
```

Listing 7: Przykład działania funkcji `wait` w przypadku zabicia procesu

```
#include <stdio.h>

main(){
    int pid1, pid2, status;

    pid1 = fork();
    if (pid1 == 0){ /* proces potomny */
        sleep(10);
        exit(7);
    }
    /* proces macierzysty */
    printf("Mam potomka o identyfikatorze %d\n", pid1);
    kill(pid1, 9);
    pid2 = wait(&status);
    printf("Status zakonczenia procesu %d: %x\n", pid2, status);
}
```

## 3.3 Dziedziczenie tablicy deskryptorów

Proces dziedziczy tablicę deskryptorów od swojego przodka. Jeśli nie nastąpi jawne wskazanie zmiany, standardowym wejściem, wyjściem i wyjściem diagnostycznym procesu uruchamianego przez powłokę w wyniku interpretacji polecenia użytkownika jest terminal, gdyż terminal jest też standardowym wejściem, wyjściem i wyjściem diagnostycznym powłoki. Zmiana standardowego wejścia lub wyjścia możliwa jest dzięki temu, że funkcja systemowa `exec` nie zmienia stanu tablicy deskryptorów. Możliwa jest zatem podmiana odpowiednich deskryptorów w procesie przed

wywołaniem funkcji `exec`, a następnie zmiana wykonywanego programu. Nowo uruchomiony program w ramach istniejącego procesu zastanie ustawione odpowiednio deskryptory otwartych plików i pobierając dane ze standardowego wejścia (z pliku o deskryptorze 0) lub przekazując dane na standardowe wyjście (do pliku o deskryptorze 1) będzie lokalizował je w miejscach wskazanych jeszcze przed wywołaniem funkcji `exec` w programie. Jest to jeden z powodów, dla których oddzielono w systemie UNIX funkcje tworzenie procesu (`fork`) od funkcji uruchamiania programu (`exec`).

Jednym ze sposobów zmiany standardowego wejścia, wyjścia lub wyjścia diagnostycznego jest wykorzystanie faktu, że funkcje alokujące deskryptory (między innymi `creat`, `open`) przydzielają zawsze deskryptor o najniższym wolnym numerze (patrz 2.3, str. 2.3). W programie przedstawionym na listingu 8 następuje przeadresowanie standardowego wyjścia do pliku o nazwie `ls.txt`, a następnie uruchamiany jest program `ls`, którego wynik trafia właśnie do tego pliku.

Listing 8: Przykład przeadresowania standardowego wyjścia

```
#include <stdio.h>

main(int argc, char* argv[]){
    close(1);
    creat("ls.txt", 0600);
    execvp("ls", argv);
}
```

**Opis programu:** W linii 4 zamykany jest deskryptor dotychczasowego standardowego wyjścia. Zakładając, że standardowe wejście jest otwarte (deskryptor 0), deskryptor numer 1 jest wolnym deskryptorem o najmniejszej wartości. Funkcja `creat` przydzieli zatem deskryptor 1 do pliku `ls.txt` i plik ten będzie standardowym wyjściem procesu. Plik ten pozostanie standardowym wyjściem również po uruchomieniu innego programu przez wywołanie funkcji `execvp` w linii 5. Wynik działania programu `ls` trafi zatem do pliku o nazwie `ls.txt`.

Warto zwrócić uwagę, że wszystkie argumenty z linii poleceń przekazywane są w postaci wektora `argv` do programu `ls`. Program z listingu 8 umożliwia więc przekazanie wszystkich argumentów i opcji, które są argumentami polecenia `ls`. Do argumentów tych nie należy znak przeadresowania standardowego wyjścia do pliku lub potoku (np. `ls > ls.txt` lub `ls | more`). Znaki `>`, `>>`, `<` i `|` interpretowane są przez powłokę i proces powłoki dokonuje odpowiednich zmian standardowego wejścia lub wyjścia przed uruchomieniem programu żądanego przez użytkownika. Nie są to zatem znaki, które trafiają jako argumenty do programu uruchamianego przez powłokę.

## 3.4 Sieroty i zombi

Jak już wcześniej wspomniano, prawie każdy proces w systemie UNIX tworzony jest przez inny proces, który staje się jego przodkiem. Przodek może zakończyć swoje działanie przed zakończeniem swojego potomka. Taki proces potomny, którego przodek już się zakończył, nazywany jest *sierotą* (ang. orphan). Sieroty adoptowane są przez proces systemowy `init` o identyfikatorze 1, tzn. po osieroceniu procesu jego przodkiem staje się proces `init`.

Program na listingu 9 tworzy proces-sierotę, który będzie istniał przez około 30 sekund.

Listing 9: Utworzenie sieroty

```
#include <stdio.h>

main(){
    if (fork() == 0){
        sleep(30);
        exit(0);
```

```
7     }
      exit(0);
9  }
```

**Opis programu:** W linii 4 tworzony jest proces potomny, który wykonuję część warunkową (linie 5–6). Proces potomny śpi zatem przez 30 sekund (linia 5), po czym kończy swoje działanie przez wywołanie funkcji systemowej `exit`. Współbieżnie działający proces macierzysty kończy swoje działanie zaraz po utworzeniu potomka (linia 8), osierocając go w ten sposób.

Po zakończeniu działania proces kończy się i przekazuje status zakończenia. Status ten może zostać pobrany przez jego przodka w wyniku wywołania funkcji systemowej `wait`. Do czasu wykonania funkcji `wait` przez przodka status przechowywany jest w tablicy procesów na pozycji odpowiadającej zakończonemu procesowi. Proces taki istnieje zatem w tablicy procesów pomimo, że zakończył już wykonywanie programu i zwolnił wszystkie pozostałe zasoby systemu, takie jak pamięć, procesor (nie ubiega już się o przydział czasu procesora), czy pliki (pozamykane zostały wszystkie deskryptory). Proces potomny, który zakończył swoje działanie i czeka na przekazanie statusu zakończenia przodkowi, określany jest terminem *zombi*.

Program na listingu 10 tworzy proces-zombi, który będzie istniał około 30 sekund.

Listing 10: Utworzenie zombi

```
1 #include <stdio.h>

3 int main(){
      if (fork() == 0)
5        exit(0);
      sleep(30);
7     wait(NULL);
  }
```

**Opis programu:** W linii 4 tworzony jest proces potomny, który natychmiast kończy swoje działanie przez wywołanie funkcji `exit` (linia 5), przekazując przy tym status zakończenie. Proces macierzysty zwleka natomiast z odebraniem tego statusu śpiąc przez 30 sekund (linia 6), a dopiero później wywołuje funkcję `wait`, co usuwa proces-zombi.

Zombi nie jest tworzony wówczas, gdy jego przodek ignoruje sygnał SIGCLD (używa się też mnemonika SIGCHLD). Szczegóły znajdą się w rozdziale 6.

## 3.5 Zadania

3.1. Które ze zmiennych `pid1` – `pid5` na listingu 11 będą miały równe wartości?

Listing 11:

```
1 #include <stdio.h>

  main(){
4     int pid1, pid2, pid3, pid4, pid5;

      pid1 = fork();
7     if (pid1 == 0){
         pid2 = getpid();
         pid3 = getppid();
10    }
      pid4 = getpid();
      pid5 = wait(NULL);
13 }
```

3.2. Ile procesów zostanie utworzonych w wyniku uruchomienia programu przedstawionego na listingu 12?

Listing 12:

```
#include <stdio.h>

main(){
   fork();
   fork();
   if (fork() == 0)
      fork();
   fork();
}
```

3.3. Ile procesów zostanie utworzonych w wyniku uruchomienia programu przedstawionego na listingu 13?

Listing 13:

```
#include <stdio.h>

main(){
   fork();
   fork();
   if (fork() == 0)
      exit(0);
   fork();
}
```

3.4. Jaki będzie wynik działania programu (jaka wartość zostanie wyświetlona jako status), jeśli program przedstawiony na listingu 14 zostanie uruchomiony:

(a) z argumentem 1 (uśpienie przodka na czas 1 sekundy przed wywołaniem funkcji systemowej `kill`),

(b) z argumentem 5 (uśpienie przodka na czas 5 sekund przed wywołaniem funkcji systemowej `kill`)?

Listing 14:

```
#include <stdio.h>

main(int argc, char* argv[]){
   int pid1, pid2, status;

   pid1 = fork();
   if (pid1 == 0) { /* proces potomny */
      sleep(3);
      exit(7);
   }
   /* proces macierzysty */
   printf("Mam potomka o identyfikatorze %d\n", pid1);
   sleep(atoi(argv[1]));
   kill(pid1, 9);
   pid2 = wait(&status);
   printf("Status zakonczenia procesu %d: %x\n", pid2, status);
}
```

3.5. Jaki będzie wynik działania programu (co zostanie wypisane na standardowym wyjściu) w wyniku jego wykonania programu z listingu 15, gdy:

(a) CZAS_POTOMKA >> CZAS_RODZICA
(b) CZAS_POTOMKA << CZAS_RODZICA?

Listing 15:

```
#define CZAS_POTOMKA (?)
#define CZAS_RODZICA (?)

int main() {
    int pid;
    pid = fork();
    if ( pid == 0 ){
       sleep( CZAS_POTOMKA );
       exit(7);
    }
    else {
       int status;
       sleep( CZAS_RODZICA );
       kill( pid, 9 );
       wait( &status );
       printf( "Status = %x\n", status );
    }
}
```

3.6. W jaki sposób wynik wykonania programu przedstawionego na listingu 16 zależy od katalogu bieżącego, tzn. co pojawi się na standardowym wyjściu w zależności od tego, jaka jest zawartość katalogu bieżącego?

Listing 16:

```
#include <stdio.h>

main(){
    printf("Poczatek\n");
    execl("ls", "ls", "-l", NULL);
    printf("Koniec\n");
}
```

## 3.6 Pytania kontrolne

1. Ile dodatkowych procesów tworzonych jest przez jednokrotne wywołanie funkcji systemowej `fork`?
2. Jakie parametry przekazujemy do funkcji systemowej `fork` w celu utworzenia procesu potomnego?
3. Co jest wartością zwrotną funkcji `fork`?
4. Czy proces macierzysty może zakończyć działanie przed swoim potomkiem?
5. Kiedy (w jakich warunkach) nastąpi wykonanie następnej instrukcji po wywołaniu funkcji systemowej `exec`?
6. Na którego potomka oczekuje proces macierzysty w funkcji systemowej `wait`?
7. Co jest wartością zwrotną funkcji systemowej `wait`?
8. Jaką funkcję systemową należy wywołać w celu zakończenia procesu?
9. Co jest parametrem funkcji systemowej `exit`?
10. W jaki sposób może nastąpić zakończenie procesu?

11. Co się dzieje z deskryptorami otwartych plików po zakończeniu procesu?
12. W jaki sposób wywołanie funkcji `exec` wpływa na zawartość tablicy deskryptorów procesu?
13. Jaki program wykonuje proces potomny zaraz po utworzeniu?
14. Jakie pliki otwarte są w procesie potomnym zaraz po jego utworzeniu?
15. Jaki proces nie wykonuje żadnego programu?
16. W jakim celu system operacyjny utrzymuje procesy *zombi*?
17. Co dzieje się z procesem po zakończeniu jego przodka?
18. Kto jest przodkiem procesu-sieroty?