Funkcje jądra systemu operacyjnego UNIX w przykładach*

Dariusz Wawrzyniak
darek@cs.put.poznan.pl

5 kwietnia 2001

1 Pliki

1.1 Operacje na plikach zwykłych

Jądro systemu operacyjnego UNIX udostępnia dwie podstawowe operacje na plikach — *odczyt* i *zapis* — realizowane odpowiednio przez funkcje systemowe **`read`** i **`write`**. Z punktu widzenie jądra w systemie UNIX plik nie ma żadnej struktury, tzn. nie jest podzielony na przykład na rekordy. Plik jest traktowany jako tablica bajtów, zatem operacje odczytu lub zapisu mogą dotyczyć dowolnego fragmentu pliku, określonego z dokładnością do bajtów.

Wykonanie operacji wymaga wskazania pliku, na którym operacja ma zostać wykonana. Plik w systemie UNIX identyfikowany jest przez nazwę (w szczególności podaną w postaci ścieżki katalogowej), przy czym podawanie nazwy pliku przy każdym odwołaniu do niego wymagałoby każdorazowego przeszukiwania odpowiednich katalogów w celu ostatecznego ustalenia jego lokalizcji. W celu uniknięcia czasochłonnego przeszukiwania katalogów podczas lokalizowania pliku przy każdej operacji na nim, wprowadzona została funkcja systemowa **`open`**, której zadanim jest zaalokowanie niezbędnych zasobów w jądrze, umożliwiających wykonywanie dalszych operacji na pliku bez potrzeby przeszukiwania katalogów. Funkcja **`open`** zwraca *deskryptor*, który jest przekazywany jako parametr aktualny, identyfikujący plik, do funkcji systemowych związanych z operacjami na otwartych plikach. Przy otwieraniu pliku przekazywany jest tryb otwarcia, określający dopuszczalne operacje, jakie można wykonać w związku z tym otwarciem, np. *tylko zapis*, *tylko odczyt* lub *zapis i odczyt*. Tryb otwarcia może mieć również wpływ na sposób wykonania tych operacji, np. każda operacja zapisu dopisuje dane na końcu pliku.

Jądro systemu operacyjnego dostarcza też mechanizm tworzenie plików. Mechanizm tworzenia plików zwykłych dostępny jest przez funkcję systemową **`creat`**, która tworzy plik o nazwie podanej jako paramtetr aktualny i otwiera utworzony plik w trybie do zapisu, zwracając odpowiedni deskryptor.

1.2 Przykłady zastosowania operacji plikowych

Listing 1.1 przedstawia program do kopiowania pliku. W programie wykorzystano funkcje systemowe **`open`**, **`creat`**, **`read`**, **`write`** i **`close`**. Nazwy plików przykazywane są jako argumenty w linii poleceń przy uruchamianiu programu. Jako pierwszy argument przekazywana jest nazwa istniejącego pliku źródłowego, a jako drugi argument przekazywana jest nazwa pliku docelowego, który może zostać dopiero utworzony.

Listing 1.1: Kopiowanie pliku

```
#include <fcntl.h>
#include <stdio.h>
```

*W przypadku wykrycia jakichkolwiek błędów proszę o mail na podany adres.

```
3  #define MAX 512

   int main(int argc, char* argv[]){
6     char buf[MAX];
      int desc_zrod, desc_cel;
      int lbajt;
9
      if (argc<3){
         fprintf(stderr, "Za malo argumentow. Uzyj:\n");
12       fprintf(stderr, "%s <plik zrodlowy> <plik docelowy>\n", argv[0]);
         exit(1);
      }
15
      desc_zrod = open(argv[1], O_RDONLY);
      if (desc_zrod == -1){
18       perror("Blad otwarcia pliku zrodlowego");
         exit(1);
      }
21
      desc_cel = creat(argv[2], 0640);
      if (desc_cel == -1){
24       perror("Blad utworzenia pliku docelowego");
         exit(1);
      }
27
      while((lbajt = read(desc_zrod, buf, MAX)) > 0){
         if (write(desc_cel, buf, lbajt) == -1){
30          perror("Blad zapisu pliku docelowego");
            exit(1);
         }
33    }
      if (lbajt == -1){
         perror("Blad odczytu pliku zrodlowego");
36       exit(1);
      }

39    if (close(desc_zrod) == -1 || close(desc_cel) == -1){
         perror("Blad zamkniecia pliku");
         exit(1);
42    }

      exit(0);
45 }
```

Opis programu: W liniach 10–14 następuje sprawdzenie poprawności przekazania argumentów z linii poleceń. Następnie otwierany jest w trybie *tylko do odczytu* plik źródłowy i sprawdzana jest poprawność wykonania tej operacji (linie 16–20). Podobnie tworzony jest i otwierany w trybie *tylko do zapisu* plik docelowy (linie 22–26). Właściwe kopiowanie zawrtości pliku źródłowego do pliku docelowego następuje w pętli w linich 28–33. Wyjście z pętli `while` następuje w wyniku zwrócenia przez funkcję **`read`** wartości 0 lub -1. Wartość -1 oznacza błąd, co sprawdzane jest zaraz po zakończeniu pętli w liniach 34–37. Po każdym błędzie funkcji systemowej wyświetlany jest odpowiedni komunikat i następuje zakończenie procesu przez wywołanie funkcji systemowej **`exit`**. Jeśli wywołania funkcji systemowych zakończą sie bezbłędnie, sterowanie dochodzi do linii 39, gdzie następuje zamknięcie plików.

Listing 1.2 przedstawia program do wyświetlania rozmiaru pliku. W programie wykorzystano funkcje systemowe **open**, **lseek** i **close**. Nazwa pliku przykazywana jest jako argument w linii poleceń przy uruchamianiu programu.

Listing 1.2: Wyprowadzanie rozmiaru pliku

```
  #include <fcntl.h>
  #include <stdio.h>
3
  int main(int argc, char* argv[]){
     int desc;
6    long rozm;

     if (argc < 2){
9       fprintf(stderr, "Za malo argumentow. Uzyj:\n");
        fprintf(stderr, "%s <nazwa pliku>\n", argv[0]);
        exit(1);
12   }

     desc = open(argv[1], O_RDONLY);
15   if (desc == -1){
        perror("Blad otwarcia pliku");
        exit(1);
18   }

     rozm = lseek(desc, 0, SEEK_END);
21   if (rozm == -1){
        perror("Blad w pozycjonowaniu");
        exit(1);
24   }

     printf("Rozmiar pliku %s: %ld\n", argv[1], rozm);
27
     if (close(desc) == -1){
        perror("Blad zamkniecia pliku");
30      exit(1);
     }

33   exit(0);
  }
```

Opis programu: W liniach 8–12 następuje sprawdzenie poprawności przekazania argumentów z linii poleceń. Następnie otwierany jest w trybie *tylko do odczytu* plik o nazwie podanej jako argument w linii poleceń i sprawdzana jest poprawność wykonania tej operacji (linie 14–18). Po otwarciu pliku następuje przesunięcie wskaźnika bieżącej pozycji za pomacą funkcji **lseek** na koniec pliku i zarazem odczyt położenia tego wskaźnika względem początku pliku (linia 20). Uzyskany wynik działania funkcji **lseek**, jeżeli nie jest to wartość -1, jest rozmiarem pliku w bajtach. Wartość ta jest wyświetlana na standardowym wyjściu (linia 26), po czym plik jest zamykany (linia 28).

Listing 1.3 zawiera rozbudowaną wersję programu z listingu 1.2, w ten sposób, że wyświetlane są rozmiary wszystkich plków, których nazwy zostały przekazane jako argumenty w linii poleceń.

Listing 1.3: Wyprowadzanie rozmiaru wielu plików

```
  #include <fcntl.h>
  #include <stdio.h>
3
```

```
int main(int argc, char* argv[]){
   int desc, i;
   long rozm;

   if (argc < 2){
      fprintf(stderr, "Za malo argumentow. Uzyj:\n");
      fprintf(stderr, "%s <nazwa pliku> ...\n", argv[0]);
      exit(1);
   }

   for (i=1; i<argc; i++) {
      desc = open(argv[i], O_RDONLY);
      if (desc == -1){
         char s[50];
         sprintf(s, "Blad otwarcia pliku %s", argv[i]);
         perror(s);
         continue;
      }

      rozm = lseek(desc, 0, SEEK_END);
      if (rozm == -1){
         perror ("Blad w pozycjonowaniu");
         exit(1);
      }

      printf("Rozmiar pliku %s: %ld\n", argv[i], rozm);

      if (close(desc) == -1){
         perror("Blad zamkniecia pliku");
         exit(1);
      }
   }

   exit(0);
}
```

1.3 Zadania

1.1 Napisz program do rozpoznawania, czy plik o nazwie podanej jako argument w linii poleceń jest plikiem tekstowym.
Wskazówka: wykorzystaj fakt, że plik tekstowy zawiera znaki o kodach 0–127 (można w tym celu użyć funkcji `isascii`).

1.2 Napisz program konwertujący małe litery na duże w pliku podanym jako argument w linii poleceń.
Wskazówka: odczytaj blok z pliku do bufora, sprawdź kody poszczególnych znaków i jeśli odpowiadają małym literom, to dodaj do kodu znaku różnicę pomiędzy kodem litery `A` i `a` (można też użyć funkcji `toupper`), a następnie zapisz zmodyfikowany blok w tym samym miejscu pliku, z którego był odczytany (cofnij odpowiednio wskaźnik bieżącej pozycji).

1.3 Napisz program ustalający liczbę znaków w najdłuższej linii w pliku o nazwie podanej jako argument w linii poleceń.

1.4 Napisz program wyświetlający najdłuższą linię w pliku o nazwie podanej jako argument w linii poleceń.

1.5 Napisz program, który w pliku o nazwie podanej jako ostatni argument zapisze połączoną zawartość wszystkich plików, których nazwy zostały podane w linii poleceń przed ostatnim argumentem.

1.6 Napisz program, który policzy wszystkie słowa w pliku podanym jako argument w linii poleceń, przyjmując, że słowa składają się z małych i dużych liter alfabetu oraz cyfr i znaku podkreślenia, a wszystkie pozostałe znaki są separatorami słów.

1.7 Napisz program do porównywania dwóch plików o nazwach przekazanych jako argumenty w linii poleceń. Wynikiem działania programu ma być komunikat, że *pliki są identyczne*, *pliki różnią się począwszy od znaku nr <numer znaku> w linii <numer linii>* lub — gdy jeden z plików zawiera treść drugiego uzupełnioną na końcu o jakieś dodatkowe znaki — *plik <nazwa> zawiera <liczba> znaków więcej niż zawartość pliku <nazwa>*.

2 Procesy

2.1 Obsługa procesów

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 rodzicielskim* lub krótko *rodzicem*. Nowo utworzony proces nazywany jest *potomkiem* lub *procesem potomnym*. Procesy w systemie UNIX tworzą zatem drzewiastą strukturę hierarchiczną, podobnie jak katalogi.

Potomek tworzony jest w wyniku wywołania przez przodka funkcji systemowej **`fork`**. Po utworzeniu potomek kontynuuje wykonywanie programu swojego przodka od miejsca wywołania funkcji **`fork`**. Proces może się zakończyć dwojako: w sposób naturalny przez wywołanie funkcji systemowej **`exit`** 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. 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.

W ramach istniejącego procesu może nastąpić zmiana wykonywanego programu w wyniku wywołania jednej z funkcji systemowych **`execl`**, **`execlp`**, **`execle`**, **`execv`**, **`execvp`**, **`execve`**. Funkcje te będą określane ogólną nazwą **`exec`**. Bezbłędne wykonanie funkcji **`exec`** oznacza bezpowrotne zaprzestanie wykonywania bieżącego programu i rozpoczęcie wykonywania programu, którego nazwa jest przekazana jako argument. Nowy program zastępuje zatem program, który był wykonywany dotychczas i w którym znajdowała się instrukcja wywołania funkcji systemowej **`exec`**.

2.2 Przykłady użycia funkcji obsługi procesów

Listingi 2.1 i 2.2 przedstawiają program, który ma zasygnalizować początek i koniec swojego działania przez wyprowadzenia odpowiedniego tekstu na standardowe wyjście.

Listing 2.1: Przykład działania funkcji **`fork`**

```
#include <stdio.h>

main(){
  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
```

Listing 2.2: Przykład działania funkcji **exec**

```
#include <stdio.h>
2
main(){
4   printf("Poczatek\n");
    execlp("ls", "ls", "-a", NULL);
6   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 2.3 przedstawia program, który zaznacza początek i 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 2.3: Przykład uruchamiania programów

```
#include <stdio.h>

3 main(){
    printf("Poczatek\n");
    if (fork() == 0){
6      execlp("ls", "ls", "-a", NULL);
       perror("Blad uruchmienia programu");
       exit(1);
9   }
    wait(NULL);
    printf("Koniec\n");
12 }
```

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. 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 momnentu zakończenie potomka.

W powyższym programie (listing 2.3), jak również w innych programach w tym rozdziale założono, że funkcje systemowe wykonują się bez błędów. Program na listingu 2.4 jest modyfikacją poprzedniego programu, plegającą na sprawdzaniu poprawności wykonania funkcji systemowych.

Listing 2.4: 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 2.5 i 2.6 przedstawiają program, którego zadaniem jest zademonstrować wykorzystanie funkcji **wait** do przkazywania przodkowi przez potmka *statusu zakończenia procesu*.

Listing 2.5: 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 przodka o identyfikatorze %d\n", pid1);
   pid2 = wait(&status);
   printf("Status zakonczenia procesu %d: %x\n", pid2, status);
}
```

Listing 2.6: 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 przodka o identyfikatorze %d\n", pid1);
   kill(pid1, 9);
```

```
    pid2 = wait(&status);
15  printf("Status zakonczenia procesu %d: %x\n", pid2, status);
  }
```

2.3 Standardowe wejście i wyjście

Programy systemowe UNIX'a oraz pewne funkcje biblioteczne przakazują wyniki swojego działania na *standardowe wyjście*, czyli do jakiegoś pliku, którego deskryptor ma ustaloną wartość. Podobnie komunikty o błędach przekazywane są na *standardowe wyjście diagnostyczne*. Tak działają na przykład programy `ls`, `ps`, funkcje biblioteczne `printf`, `perror` itp. Programy `ls` i `ps` nie pobierają żadnych danych wejściowych (jedynie argumenty i opcje przekazane w linii poleceń), jest natomiast duża grupa programów, które na standardowe wyjście przekazują wyniki przetwarzania danych wejściowych. Przykładami takich programów są: `more`, `grep`, `sort`, `tr`, `cut` itp. Plik z danymi wejściowymi dla tych programów może być przekazany przez podanie jego nazwy jako jednego z argumentów w linii poleceń. Jeśli jednak plik nie zostanie podany, to program zakłada, że dane należy czytać ze *standardowego wejścia*, czyli otwartego pliku o ustalonym deskryptorze. Przyporządkowanie deskryptorów wygląda następująco:

0 — deskryptor standardowego wejścia, na którym jest wykonywana funkcja **`read`** w programach systemowych w celu pobrania danych do przetwarzania;

1 — deskryptor standardowego wyjścia, na którym wykonywana jest funkcja **`write`** w programach systemowych w celu przekazania wyników;

2 — deskryptor standardowego wyjścia diagnostycznego, na którym wykonywana jest funkcja **`write`** w celu przekazania komunikatów o błędach.

Z punktu widzenie programu nie jest istotne, jaki plik lub jaki rodzaj pliku identyfikowany jest przez dany deskryptor. Ważne jest, jakie operacje można na takim pliku wykonać. W ten sposób przejawia się niezależności plików od urządzeń. Operacje wykonywane na plikach identyfikowanych przez deskryptory 0–2 to najczęściej **`read`** i **`write`**. Warto zwrócić uwage na fakt, że funkcja systemowa **`lseek`** może być wykonywana na pliku o dostępie bezpośrednim (swabodnym), nie może być natomiast wykonana na pliku o dostępie sekwencyjnym, czyli urządzeniu lub łączu komunikacyjnym. Za pomocą `more` można więc cofać się w przeglądanym plik tylko wówczas, gdy jego nazwa jest przekazana jako parametr w linii polceń.

Proces dziedziczy tablicę deskryptorów od swojego przodka. Jeśli nie nastąpi jewne wskazanie zmiany, standarodwym 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. W programie przedstawionym na listingu 2.7 następuje przeadresowanie standardowego wyjścia do pliku o nazwie `ls.txt`, a następnie uruchamiany jest program `ls`, którego wynik trafia właśnie do tego pliku.

Listing 2.7: Przykład przeadresowania standardowego wyjścia

```
#include <stdio.h>
2
main(int argc, char* argv[]){
4   close(1);
    creat("ls.txt", 0600);
6   execvp("ls", argv);
}
```

Opis programu: W linii 4 zamykany jest deskryptor dotychczasego standardowego wyjścia. Zakładając, że standarodwe 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 2.7 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.

2.4 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 2.8 tworzy proces-sierotę, który będzie istniał przez około 30 sekund.

Listing 2.8: Utworzenie sieroty

```
#include <stdio.h>
2
main(){
4   if (fork() == 0){
       sleep(30);
6      exit(0);
    }
8   exit(0);
}
```

Opis programu: W linii 4 tworzony jest proces potomny, który wykonuję część warunkową (linie 5–6). Proces potmny 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 2.9 tworzy proces-zombi, który będzie istniał około 30 sekund.

Listing 2.9: Utworzenie zombi

```
  #include <stdio.h>
2
  int main(){
4    if (fork() == 0)
        exit(0);
6    sleep(30);
     wait(NULL);
8 }
```

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 dopier później wywołuje funkcję **wait**, co usuwa proces-zombi.

Zombi nie jest tworzony wówczas, gdy jego przodek ignoruje sygnał SIGCLD (sygnał nr 4, używa się też mnemoniku SIGCHLD). Szczegóły znajdą się w rozdziale 5.

2.5 Zadania

2.1 Które ze zmiennych `pid1` – `pid5` na listingu 2.10 będą miały równe wartości?

Listing 2.10:

```
  #include <stdio.h>

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

6    pid1 = fork();
     if (pid1 == 0){
        pid2 = getpid();
9       pid3 = getppid();
     }
     pid4 = getpid();
12   pid5 = wait(NULL);
  }
```

2.2 Ile procesów zostanie utworzonych przy uruchomieniu programu przedstawionego na listingu 2.11?

Listing 2.11:

```
  #include <stdio.h>

3 main(){
     fork();
```

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

2.3 Ile procesów zostanie utworzonych przy uruchomieniu programu przedstawionego na listingu 2.12?

Listing 2.12:

```
#include <stdio.h>

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

2.4 Jaki będzie wynik działania programu (jaka wartość zostanie wyświetlona jako status), jeśli program przedstawiony na listingu 2.13 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 2.13:

```
#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 przodka o identyfikatorze %d\n", pid1);
    sleep(atoi(argv[1]));
    kill(pid1, 9);
    pid2 = wait(&status);
    printf("Status zakonczenia procesu %d: %x\n", pid2, status);
}
```

2.5 W jaki sposób wynik wykonania programu przedstawionego na listingu 2.14 zależy od katalogu bieżącego, tzn. co pojawi się na standardowym wyjściu w zależności od tego, jaka jest zawartość katalogu bieżącego?

Listing 2.14:

```
#include <stdio.h>

main(){
```

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

3 Łącza

4 Mechanizmy IPC

4.1 Kolejki komunikatów

4.2 Semafory i pamięć współdzielona

5 Sygnały