Temat zajęć: Obsługa procesów w systemie.

Czas realizacji zajęć:	*90 min.*
Zakres materiału, jaki zostanie zrealizowany podczas zajęć:	Procesy macierzyste i potomne, tworzenie procesów potomnych, uruchamianie programów, przekierowywanie strumieni standardowych, implementacja przykładowych programów obsługi procesów

I. Obsługa procesów.

Proces (zwany też zadaniem) jest jednostką aktywną, kontrolowaną przez system operacyjny i związaną z wykonywanym programem. Proces ma przydzielone zasoby typu pamięć (segment kodu, segment danych, segment stosu, segment danych systemowych), procesor, urządzenia zewnętrzne itp. Część przydzielonych zasobów jest do wyłącznej dyspozycji procesu (np. segment danych, segment stosu), część jest współdzielona z innymi procesami (np. procesor, segment kodu w przypadku współbieżnego wykonywania tego samego programu w ramach kilku procesów).
W zależności od aktualnie posiadanych zasobów wyróżnia się stany procesu (np. wykonywany, uśpiony, gotowy), które zmieniają się cyklicznie w związku z wykonywanym programem lub ze zdarzeniami zachodzącymi w systemie. Szczegółowy stan procesu, umożliwiający kontynuację jego wykonywania po przerwaniu nazywany jest kontekstem procesu.

W zakresie obsługi procesów system UNIX udostępnia mechanizm tworzenia nowych procesów, usuwania procesów oraz uruchamiania programów. Każdy proces, z wyjątkiem procesu systemowego o identyfikatorze 1, tworzony jest przez 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 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 staję się w ten sposób jego przodkiem. Jeżeli proces potomny zakończył działanie przed wywołaniem funkcji **wait** w procesie macierzystym, potomek pozostanie w stanie *zombi* (proces taki nazywany jest zombi, upiorem, duchem lub mumią). 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.

W ramach istniejącego procesu może nastąpić uruchomienie innego programu w 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**. 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.

Funkcje służące do obsługi procesów opisane są w 2 i 3części pomocy systemowej i w większości zdefiniowane w plikach `sys/types.h` oraz `unistd.h`.

II. Funkcje systemowe i ich argumenty.

- **`int fork( void );`**

  Wartości zwracane:
  poprawne wykonanie funkcji: utworzenie procesu potomnego; W procesie macierzystym funkcja zwraca identyfikator (pid) procesu potomnego (wartość większą od 1), a w procesie potomnym wartość 0.
  zakończenie błędne: -1

  Możliwe kody błędów (*errno*) w przypadku błędnego zakończenie funkcji:
  EAGAIN – błąd alokacji wystarczającej ilości pamięci na skopiowanie stron rodzica i zaalokowanie struktury zadań
  ENOMEM – nie można zaalokować niezbędnych struktur jądra z powodu braku pamięci

  UWAGI:
  W momencie wywołania funkcji (przez proces który właśnie staje się przodkiem) tworzony jest proces potomny, który wykonuje współbieżnie ze swoim przodkiem ten sam program. Potomek rozpoczyna wykonywanie programu od wyjścia z funkcji **fork** i kontynuuje wykonując kolejną instrukcję, znajdującą się w programie po wywołaniu funkcji **fork**. Do funkcji **fork** wchodzi zatem tylko proces macierzysty, a wychodzą z niej dwa procesy: macierzysty i potomny, przy czym każdy z nich otrzymuję inną wartość zwrotną funkcji **fork**. Wartością zwrotną funkcji **fork** w procesie macierzystym jest identyfikator (PID) potomka, a w procesie potomnym wartość 0. W przypadku błędnego wykonania funkcji **fork** potomek nie zostanie utworzony, a proces wywołujący otrzyma wartość -1.

- **`int getpid(void) oraz int getppid(void)`**

  Wartości zwracane:
  poprawne wykonanie funkcji: zwrócenie własnego identyfikatora (w przypadku funkcji **getpid**) lub identyfikator procesu macierzystego (dla funkcji **getppid**).
  zakończenie błędne: -1

- **`void exit(int status)`**

  Wartości zwracane:
  poprawne wykonanie funkcji: przekazanie w odpowiednie miejsce tablicy procesów wartości *status*, która może zostać odebrana i zinterpretowana przez proces macierzysty.
  zakończenie błędne: -1

  UWAGI:
  Funkcja 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.`

exit(0) – oznacza poprawne zakończenie wykonanie procesu
exit(dowolna wartość różna od 0) – oznacza wystąpienie błędu

- **`int wait(int *status) oraz int waitpid( int pid, int *status, int options)`**

  Wartości zwracane:
  poprawne wykonanie funkcji: identyfikator procesu potomnego, który się zakończył
  zakończenie błędne: -1 lub 0 jeśli użyto opcji WNOHANG, a nie było dostępnego żadnego potomka

  Możliwe kody błędów (*errno*) w przypadku błędnego zakończenie funkcji:
  ECHILD – jeśli proces o zadanym identyfikatorze *pid* nie istnieje lub nie jest potomkiem procesu wywołującego. (Może się to zdarzyć również w przypadku potomka, który ustawił akcję obsługi sygnału SIGCHLD na SIG_IGN)
  EINVAL – jeśli argument *options* jest niepoprawny.
  EINTR – jeśli opcja WNOHANG nie była ustawiona, a został przechwycony niezablokowany sygnał lub SIGCHLD.

  Argumenty funkcji:
  status – status zakończenia procesu (w przypadku zakończenia w sposób normalny) lub numer sygnału w przypadku zabicia potomka lub wartość NULL, w przypadku gdy informacja o stanie zakończenia procesu nie jest istotna
  pid – identyfikator potomka, na którego zakończenie czeka proces macierzysty
  - *pid* < -1 oznacza oczekiwanie na dowolny proces potomny, którego identyfikator grupy procesów jest równy modułowi wartości *pid*.
  - *pid* = -1 oznacza oczekiwanie na dowolny proces potomny; jest to takie samo zachowanie, jakie stosuje funkcja **wait**.
  - *pid* = 0 oznacza oczekiwanie na każdy proces potomny, którego identyfikator grupy procesu jest równe identyfikatorowi wołającego procesu.
  - *pid* > 0 oznacza oczekiwanie na potomka, którego identyfikator procesu jest równy wartości *pid*.

  options – jest sumą OR zera lub następujących stałych:
  - WNOHANG oznacza natychmiastowe zakończenie jeśli potomek się nie zakończył.
  - WUNTRACED oznacza zakończenie także dla procesów potomnych, które się zatrzymały, a których *status* jeszcze nie został zgłoszony.

  UWAGI:
  Oczekiwanie na zakończenie procesu potomnego. Funkcja zwraca identyfikator (*pid*) procesu, który się zakończył. Pod adresem wskazywanym przez zmienną *status* umieszczany jest status zakończenia, który zawiera albo numer sygnału (najmniej znaczące 7 bitów), albo właściwy status zakończenia (bardziej znaczący bajt).
  Gdy działa parę procesów potomnych zakończenie jednego z nich powoduje powrót z funkcji wait.
  Jeżeli funkcja **wait** zostanie wywołana w procesie macierzystym przed zakończeniem procesu potomnego, wykonywanie procesu macierzystego zostanie zawieszone do momentu zakończenia potomka. Jeżeli proces potomny zakończył działanie przed wywołaniem funkcji **wait**, powrót z funkcji **wait** nastąpi natychmiast, a w czasie pomiędzy zakończeniem potomka, a wywołaniem

funkcji **wait** przez jego przodka potomek pozostanie w stanie *zombi.* Zombi nie jest tworzony, gdy proces macierzysty ignoruje sygnał SIGCLD .
Funkcja **exit** zdefiniowana jest w pliku `sys/types.h` oraz `sys/wait.h`.

- **rodzina funkcji exec**

```
int execl ( char *path, char *arg0, ..., char *argn, char *null )
int execlp( char *file, char *arg0, ..., char *argn, char *null )
int execv ( char *path, char * argv[] )
int execvp( char *file, char * argv[] )
int execle( char * path, char *arg0, ..., char *argn,char *null,char *envp[] )
int execve( char * path, char *argv[],char *envp[] )
```

Wartości zwracane:
poprawne wykonanie funkcji: wywołanie programu podanego jako parametr
zakończenie błędne: -1

Możliwe kody błędów (*errno*) w przypadku błędnego zakończenie funkcji: Każda z tych funkcji może zakończyć się niepowodzeniem i ustawić jako wartość *errno* dowolny błąd określony dla funkcji bibliotecznej **execve**(2).

Argumenty funkcji:
path, file – pełna nazwa ścieżkowa lub nazwa pliku z programem
arg0 ...argn – nazwa i argumenty programu który ma być wywołany

UWAGI:
W ramach istniejącego procesu może nastąpić uruchomienie innego programu w wyniku wywołania jednej z funkcji systemowych **execl**, **execlp**, **execle**, **execv**, **execvp**, **execve**. Funkcje te określane są ogólną nazwą **exec**. Bezbłędne wykonanie funkcji **exec** oznacza bezpowrotne zaprzestanie wykonywania bieżącego programu i rozpoczęcie wykonywania programu, którego nazwa jest przekazana przez argument.
W wyniku wywołania funkcji typu **exec** następuje reinicjalizacja segmentów kodu, danych i stosu. Nie zmieniają się atrybuty procesu takie jak pid, ppid, tablica otwartych plików i kilka innych atrybutów z segmentu danych systemowych
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 **execle** 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.
Wartością zwrotną funkcji typu **exec** jest *status*, przy czym jest ona zwracana tylko wtedy, gdy funkcja zakończy się niepoprawnie, będzie to zatem wartość –1.
Funkcje **exec** nie tworzą nowego procesu, tak jak w przypadku funkcji fork!!!

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)
```

III. Przykłady użycia funkcji obsługi procesów.

Listingi 1 i 2 przedstawiają program, który ma zasygnalizować początek i koniec swojego działania przez wyprowadzenia odpowiedniego tekstu na standardowe wyjście.

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

Listing 1: Przykład działania funkcji **fork**

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. Wynik działania programu jest zatem następujący:

```
Poczatek
Koniec
Koniec
```

```
   #include <stdio.h>
2
   main(){
4     printf("Poczatek\n");
      execlp("ls", "ls", "-a", NULL);
6     printf("Koniec\n");
   }
```

Listing 2: Przykład działania funkcji **exec**

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, 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`.

```
   #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    }
```

Listing 3: Przykład uruchamiania programów

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 momentu zakończenia potomka. W powyższym programie (Listing 3), jak również w innych programach w tym rozdziale założono, że funkcje systemowe wykonują się bez błędów.

Program na listingu 4 jest modyfikacją poprzedniego programu, polegającą na sprawdzaniu poprawności wykonania funkcji systemowych.

```
   #include <stdio.h>

3   main(){
      printf("Poczatek\n");
      switch (fork()){
6     case -1:
            perror("Blad utworzenia procesu potomnego");
            break;
9     case 0: /* proces potomny */
            execlp("ls", "ls", "-a", NULL);
            perror("Blad uruchmienia programu");
12          exit(1);
      default: /* proces macierzysty */
            if (wait(NULL) == -1)
15          perror("Blad w oczekiwaniu na zakonczenie potomka");
      }
      printf("Koniec\n");
18 }
```

Listing 4: Przykład uruchamiania programów z kontrolą poprawności

Listingi 5 i 6 przedstawiają program, którego zadaniem jest zademonstrować wykorzystanie funkcji **wait** do przkazywania przodkowi przez potomka statusu 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 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 */
         sleep(10);
         exit(7);
   }
   /* proces macierzysty */
   printf("Mam przodka o identyfikatorze %d\n", pid1);
   kill(pid1, 9);
   pid2 = wait(&status);
   printf("Status zakonczenia procesu %d: %x\n", pid2, status);
   }
```

Listing 6: Przykład działania funkcji **wait** w przypadku zabicia procesu

IV. Standardowe wejście i wyjście

Programy systemowe UNIX'a oraz pewne funkcje biblioteczne przekazują wyniki swojego działania na standardowe wyjście, czyli do jakiegoś pliku, którego deskryptor ma ustaloną wartość. Podobnie komunikaty 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 niezależność plików od urządzeń. Operacje wykonywane na plikach identyfikowanych przez deskryptory 0–2 to najczęściej **read** i **write**. Warto zwrócić uwagę na fakt, że funkcja systemowa **lseek** może być wykonywana na pliku o dostępie bezpośrednim (swobodnym), nie może być natomiast wykonana na pliku o dostępie sekwencyjnym, czyli urządzeniu lub łączu komunikacyjnym. Za pomocą more można wiec cofać się w przeglądanym plik tylko wówczas, gdy jego nazwa jest przekazana jako parametr w linii poleceń.
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. W programie przedstawionym na listingu 7 następuje przeadresowanie standardowego wyjścia do pliku o nazwie `ls.txt`, a następnie uruchamiany jest program **ls**, którego wynik trafią właśnie do tego pliku.

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

Listing 7: Przykład przeadresowania standardowego wyjścia

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 dzia lania 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 7 umożliwia więc przekazanie wszystkich argumentów i opcji, któreś, a 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.

V. 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). Program na listingu 2.8 tworzy proces-sierotę, który będzie istniał przez około 30 sekund.

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

Listing 8: Utworzenie procesu-sieroty

Opis programu: W linii 4 tworzony jest proces potomny, który wykonuje 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 2.9 tworzy proces-zombi, który będzie istniał oko lo 30 sekund.

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

Listing 9: Utworzenie procesu-zombi

Opis programu: W linii 4 tworzony jest proces potomny, który natychmiast kończy swoje działanie przez wywołanie funkcji **exit** (linia 5), przekazując przy tym status zakończenia. 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 (sygnał nr 4, używa się też mnemoniku SIGCHLD).

[ilość modułów: 1]

VI. Zadania do samodzielnego wykonania

1) Napisz program tworzący dwa procesy. Każdy ze stworzonych procesów powinien utworzyć proces – potomka. Należy wyświetlać identyfikatory procesów rodziców i potomków po każdym wywołaniu funkcji fork.
2) Napisz program którego rezultatem będzie wydruk zawartości bieżącego katalogu poprzedzony napisem „*Poczatek*" a zakończony napisem „*Koniec*"
3) Napisz program, którego wynikiem jest sformatowana lista procesów:

```
--------poczatek listy-------------
proces 1
proces2
.....
--------koniec listy---------------
```

4) Napisz program tworzący równocześnie trzy procesy zombi.

VII. Literatura.

[HGS99] Havilland K., Gray D., Salama B., *Unix - programowanie systemowe*, ReadMe, 1999

[Roch97] Rochkind M.J., *Programowanie w systemie UNIX dla zaawansowanych*, WNT, 1997

[NS99] Neil M., Stones R, *Linux. Programowanie*, ReadMe, 1999

[MOS02] Mitchell M., Oldham J., Samuel A., *Linux. Programowanie dla zaawansowanych*, ReadMe,2002

[St02] Stevens R.W., *Programowanie w środowisku systemu UNIX*, WNT, 2002