LABORATORIUM 10/11

Procesy w systemie Linux/UNIX: tworzenie, kończenie, podmiana wykonywanego programu, oczekiwanie na potomków.

Literatura uzupełniająca

M. K. Johnson, E. W. Troan, *Oprogramowanie użytkowe w systemie Linux*, rozdz. 9.2.1 i 9.4.1-9.4.5.

W. R. Stevens, *Programowanie zastosowań sieciowych w systemie UNIX*, rozdz. 2.5.1-2.5.4.

M. J. Bach, *Budowa systemu operacyjnego UNIX*, rozdz. 7.1, 7.3-7.5.

man do poszczególnych funkcji systemowych

Pliki, z których będziemy korzystać (pliki-LABO10.zip):

Makefile	-	plik Makefile
err.h	-	plik nagłówkowy biblioteki obsługującej błędy
err.c	-	biblioteka do obsługi błędów
proc_fork.c	-	program ilustrujący tworzenie nowego procesu
proc_tree.c	-	program tworzący drzewo procesów – *do wykonania na zajęciach*
proc_exec.c	-	program tworzący nowy proces, który wykona nowe polecenie

---

Scenariusz zajęć

1. **Przypomnij sobie informacje na temat programu make**

   http://www.programuj.com/artykuly/linux/makefile.php

   LABO10 - Kompilacja i scalanie programów C w linii poleceń (LINUX).pdf

   http://www.computerhope.com/unix/umake.htm

   http://linoxide.com/how-tos/linux-make-command-examples/

   Przeczytaj plik Makefile.

   Wykonaj polecenie *make*.

   Zwróć uwagę na kolejność w jakiej kompilują się programy.

2. Identyfikator procesu

Każdy proces w systemie ma jednoznaczny identyfikator nazywany potocznie PID (od angielskiego: *Process ID*). Identyfikatory aktualnie wykonujących się procesów możesz poznać wykonując w Linuksie polecenie **ps**.

Ćwiczenie

Wykonaj polecenie **ps**. Zobaczysz wszystkie uruchomione przez Ciebie procesy w tej sesji. Znajdzie się wśród nich proces **ps** i **bash** (lub inny stosowany przez Ciebie interpreter poleceń), który analizuje i wykonuje Twoje polecenia. Pierwsza kolumna to PID procesu, a ostatnia to polecenie, które dany proces wykonuje. Więcej informacji na temat polecenia **ps** uzyskasz wywołując **man ps**.

Z poziomu programisty, proces może poznać swój PID wywołując funkcję systemową:

```
pid_t getpid();
```

Wartości typu **pid_t** reprezentują PIDy procesów. Najczęściej jest to długa liczba całkowita (**long int**), ale w zależności od wariantu systemu operacyjnego definicja ta może być inna. Dlatego lepiej posługiwać się nazwą **pid_t**.

3. Tworzenie nowego procesu

W Linuksie, tak jak we wszystkich systemach uniksowych, istnieje hierarchia procesów. Każdy proces poza pierwszym procesem w systemie (procesem **init** o PID równym 1) jest tworzony przez inny proces. Nowy proces nazywamy procesem potomnym, a proces który go stworzył nosi nazwę procesu macierzystego.

Do tworzenia procesów służy funkcja systemowa:

```
pid_t fork();
```

Powrót z wywołania tej funkcji następuje dwa razy:

- w procesie macierzystym, w którym wartością przekazywaną przez funkcję **fork** jest PID nowo utworzonego potomka,
- w procesie potomnym, w którym funkcja przekazuje w wyniku 0.

Jak dokładnie działa funkcja systemowa **fork()** ?
Proces w systemie Unix jest wygodnie wyobrażać sobie jako obiekt składający się z trzech części:

Proces Uniksowy. Podział na logiczne części.

Wykonywany kod	Dane: • w szczególności wszystkie zmienne procesu	Dane systemowe: • PID, • PID ojca • otwarte pliki • itd

Funkcja systemowa **fork** tworzy nowy proces i kopiuje do niego wszystkie powyższe elementy, zmieniając jedynie te elementy, które muszą zostać zmienione (na przykład PID). Zatem nowy proces potomny:

- wykonuje taki sam kod jak proces macierzysty;
- dziedziczy po procesie macierzystym całą historię wykonania, bo stos wykonania jest także kopiowany. Oznacza to w szczególności, że wykonanie w procesie potomnym zaczyna się od następnej instrukcji po **fork()**;
- ma te same zmienne co proces macierzysty i do tego zmienne te mają te same wartości co w procesie macierzystym. Jednak przestrzenie adresowe tych procesów są rozłączne: każdy ma swoją kopię zmiennych. Oznacza to m.in., że zmiana wartości zmiennej w procesie potomnym nie jest odzwierciedlana w procesie macierzystym i na odwrót;
- ma te same uprawnienia, te same otwarte pliki itd. (tym zajmiemy się w kolejnym laboratorium).

A oto przykład ilustrujący wykorzystanie funkcji **fork()** do tworzenia nowego procesu. Przykład ten możesz znaleźć w plikach przygotowanych do zajęć pod nazwą *proc_fork.c*.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "err.h"

int main ()
{
1:  pid_t pid;
2:  /* wypisuje identyfikator procesu */
    printf("Moj PID = %d\n", getpid());
    /* tworzy nowy proces */
3:  switch (pid = fork()) {
4:    case -1: /* błąd */
        syserr("Error in fork\n");
5:    case 0: /* proces potomny */
        printf("Jestem procesem potomnym. Mój PID = %d\n", getpid());
        printf("Jestem procesem potomnym. Wartość przekazana przez fork() = %d\n", pid);
        return 0;
6:    default: /* proces macierzysty */
        printf("Jestem procesem macierzystym. Mój PID = %d\n", getpid());
        printf("Jestem procesem macierzystym. Wartość przekazana przez fork() = %d\n", pid);
7:      /* czeka na zakończenie procesu potomnego */
        if(wait(0) == -1)
          syserr("Error in wait\n");
        return 0;
    } /*switch*/
}
```

Przeanalizujmy ten program.

A. Część z dyrektywami **#include**. Tutaj umieszczamy niezbędne pliki nagłówkowe:
   - standardową bibliotekę wejścia/wyjścia **stdio.h**
   - plik nagłówkowy **unistd.h** zawierający deklaracje standardowych funkcji uniksowych (**fork(), write()** itd.); Należy go dołączyć do programu, w przeciwnym razie kompilator generuje ostrzeżenia takie jak **implicit declaration of function fork**
   - plik nagłówkowy z deklaracją funkcji **wait()** z **sys/wait.h**
   - plik nagłówkowy z deklaracją funkcji **syserr** do obsługi błędów. Więcej na temat błędów za chwilę.

B. Kod programu umieszczamy jak zwykle w funkcji main. Wewnątrz tej funkcji w wierszu **1** znajduje się deklaracja zmiennej **pid**, na której będziemy pamiętać PID procesu.

C. W wierszu **2** program wypisuje swój PID pobrany za pomocą funkcji systemowej **getpid.**

D. W wierszu **3** następuje wywołanie funkcji systemowej **fork** i dochodzi do "rozwidlenia" procesu. Tworzy się nowy proces i w chwili powrotu z funkcji systemowej mamy już dwa procesy. Oba mają w tym momencie jeszcze tę samą wartość zmiennej **pid**, ale natychmiast po powrocie zmienna ta otrzymuje wartość przekazaną przez funkcję systemową **fork().** Będzie to zatem 0 w procesie potomnym oraz wartość większa od 0 w procesie macierzystym.

E. Wiersz **4** zawiera bardzo ważny element programu --- kontrolę błędów. System operacyjny może nie utworzyć nowego procesu z wielu powodów (np. brak pamięci, brak miejsca w tablicy procesów, przekroczenie limitu procesów na użytkownika itp). Każda funkcja systemowa przekazuje swój kod zakończenia. Jest to wartość **>= 0**, jeżeli funkcja zakończyła się pomyślnie, lub w przeciwnym wypadku liczba ujemna oznaczająca kod błędu. Ponadto jeżeli wystąpił błąd, to kod błędu jest przypisywany na globalną zmienną **errno**. Dzięki kodowi błędu uzyskujemy więcej informacji o powodach wystąpienia danego błędu. Przykład wykorzystania zmiennej **errno** można znaleźć w funkcji **syserr** znajdującej się w pliku *err.c*, która korzysta z globalnej tablicy **sys_errlist** zawierającej opisy wszystkich kodów błędów. **Uwaga! Sprawdzanie poprawności wykonania wszystkich funkcji systemowych jest absolutnym obowiązkiem programisty!**

F. Fragment programu od wiersza **5** jest wykonywany jedynie przez proces potomny, który wypisuje swój PID i wartość przekazaną mu przez funkcję systemową **fork(),** po czym kończy się.

G. Fragment programu od wiersza **6** jest wykonywany jedynie przez proces macierzysty, który wypisuje swój PID i wartość przekazaną mu przez funkcję systemową **fork().**

H. Wiersz **7** zawiera wywołanie funkcji systemowej **wait**, którą omówimy za chwilę. W skrócie powoduje ona zatrzymanie wykonania procesu w oczekiwaniu na zakończenie jego potomka.

Uwagi:

i. Jeszcze raz podkreślić należy, jak ważne jest wychwytywanie sytuacji błędnych i reakcja na nie.
ii. Po udanym wykonaniu funkcji **fork()** powyższy program jest wykonywany przez dwa procesy "jednocześnie". Oba wypisują pewne komunikaty na ekranie. Funkcja **printf** daje gwarancję niepodzielności komunikatów wypisywanych na ekran na poziomie pojedynczych wierszy. Oznacza to, że jeśli rozpoczniemy wypisywanie na ekranie, to zanim nie skończymy wiersza, nikt inny nie będzie po ekranie pisał.
iii. Konstrukcja poniżej jest dość charakterystycznym sposobem korzystania z funkcji **fork** i warto ją zapamiętać.:

```
switch (pid = fork()) {
  case -1: /* błąd */
        syserr("Error in fork\n");
  case 0: /* proces potomny */
      ...
      return 0;
  default: /* proces macierzysty */
       ...
```

Ćwiczenia:

Skompiluj i uruchom powyższy program (już zrobione -> **make**)

Czy jesteś w stanie odróżnić wiersze wypisywane przez proces potomny od wierszy wypisywanych przez proces macierzysty?

W jakiej kolejności są wypisywane komunikaty na ekranie? Czy jest to jedyna możliwa kolejność?

4. Oczekiwanie na zakończenie procesu potomnego

Proces macierzysty może zaczekać na zakończenie procesu potomnego za pomocą funkcji **wait()** (lub wait3(), wait4(), waitpid()):

pid_t wait(int *stan);

Funkcja przekazuje w wyniku PID zakończonego procesu. Parametr stan jest wskaźnikiem do zmiennej zawierającej kod zakończonego procesu. Funkcja jest blokująca, co oznacza, że proces macierzysty, który ją wywoła, zostanie wstrzymany aż do zakończenia któregoś z jego procesów potomnych. Jeżeli proces nie miał potomków, to funkcja przekazuje błąd (-1). Jeżeli potomek zakończy się zanim jego rodzic wywoła **wait**, to rodzic nie będzie czekał i wykona się poprawnie od razu dając w wyniku PID zakończonego potomka.

Po co w ogóle oczekiwać na zakończenie swoich potomków? Otóż czasem proces macierzysty chce otrzymać jakieś informacje od kończących się potomków. Jednak nawet jeśli nie ma potrzeby przekazania żadnych informacji, to i tak warto wywołać funkcję systemową **wait** i poczekać na zakończenie potomków. Dlaczego?

System operacyjny przechowuje kody zakończenia procesów potomnych, aż do chwili odebrania ich przez ich procesy macierzyste. Proces potomny, który się zakończył, a którego kod nie został odebrany przez rodzica to tzw. **zombie**. System operacyjny nie może usunąć po prostu informacji o **zombie**, bo być może w przyszłości informacja o jego kodzie zakończenia będzie potrzebna w procesie macierzystym. Z tego powodu **zombie** zajmują miejsce w tablicach systemowych. Oczywiście system operacyjny ma pewien mechanizm usuwania **zombie**, nawet jeśli ich proces macierzysty nie wywoła funkcji **wait** (za zadanie to odpowiada proces init, który po zainicjowaniu systemu "adoptuje" wszystkie osierocone procesy i wykonuje funkcję **wait**), ale wiąże się to z pewnym narzutem. Dlatego ważne jest wywoływanie funkcji **wait**, aby uniknąć niepotrzebnego zajmowania miejsca w tablicy procesów.

Ćwiczenie (**zombie**)

Zmodyfikuj poprzedni program tak, aby można było zaobserwować zombie.

Ćwiczenie sprawdzające

Za chwilę omówimy program tworzący drzewo procesów. Zanim jednak przeczytasz to omówienie spróbuj napisać go sam. Chodzi o stworzenie procesu, który utworzy zadaną z góry (na przykład w postaci stałej) liczbę procesów potomnych. Każdy proces potomny powinien wypisać na ekran jakiś komunikat kontrolny, a następnie zakończyć się. Proces macierzysty ma przed zakończeniem poczekać na zakończenie wszystkich swoich potomków.

Zapisujemy program do pliku: *proc_tree.c*

Ćwiczenie

Skompiluj i uruchom kilkakrotnie ten program. Przeanalizuj PIDy i zwróć uwagę na kolejność wypisywania informacji.

Ćwiczenie do domu na punkty

Przypuśćmy, że usunięto instrukcję **return 0** w wierszu kończącym **case** powstawania procesów potomnych. Co się stanie? Ile procesów powstanie?

Przypuśćmy, że powyższy fragment jest jedynie początkiem dużego programu. Chcemy w nim jedynie utworzyć pięć procesów potomnych, z których każdy ma opuścić pętlę i wykonywać własny kod umieszczony poza nią. Jak zmodyfikować pętlę? Ile procesów opuszcza pętlę? Jak je odróżnić od siebie?

5. Podmiana wykonywanego programu

Procesowi możemy zlecić wykonanie innego programu. Jak widzieliśmy poprzednio, jedną z części procesu jest kod. Po wykonaniu funkcji **fork()** jest on dziedziczony po procesie potomnym, ale w dowolnej chwili można go podmienić. Służy do tego rodzina funkcji **exec.** Jest to tak naprawdę ta sama funkcja, ale mająca sześć różnych postaci wywołania i przekazywanych argumentów:

int execl (const char * sciezka, const char * arg0, ...)

int execlp(const char * plik, const char * arg0, ...)

int execle(const char * sciezka, const char * arg0, ..., const char ** envp)

int execv (const char * sciezka, const char ** argv)

int execvp(const char * plik, const char ** argv)

int execve(const char * sciezka, const ** char argv, const char ** envp)

Dokładny opis wszystkich postaci znajdziesz w **man**. Teraz tylko przedstawimy znaczenie poszczególnych literek w nazwach funkcji i omówimy dwie postaci wywołania tej funkcji:

- **l** oznacza, że argumenty wywołania programu są w postaci listy napisów zakończonej zerem (NULL)
- **v** oznacza, że argumenty wywołania programu są w postaci tablicy napisów (tak jak argument **argv** funkcji main)
- **p** oznacza, że plik z programem do wykonania musi się znajdować na ścieżce przeszukiwania ze zmiennej środowiskowej **PATH**
- **e** oznacza, że środowisko jest przekazywane ręcznie jako ostatni argument

Przyjrzyjmy się przykładowej postaci tej funkcji:

int execlp (const char * sciezka, const char * arg0, ...)

Poszczególne argumenty mają następujące znaczenie:

- ścieżka to pełna ścieżka do pliku zawierającego wykonywalny program lub po prostu nazwa pliku z programem. W tym drugim przypadku plik będzie przeszukiwany w katalogach, które znajdują się na zmiennej środowiskowej **PATH**
- **arg0** to nazwa pliku zawierającego program (już bez ścieżki)
- kolejne argumenty zawierają właściwe argumenty wywoływanego programu.

Jeśli przykładowo chcemy wywołać program, który wykona polecenie **ls -l /bin**, to użyjemy funkcji systemowej w następującej postaci:

execlp ("ls", "ls", "-l", "/bin", 0)

Jeżeli wykonanie funkcji **exec()** się powiedzie, to nigdy nie nastąpi powrót z jej wywołania. Funkcje **exec()** najczęściej wywołuje się zaraz po wykonaniu **fork()** w procesie potomnym. Jak zobaczymy w kolejnym laboratorium, czasem jednak między **fork** a **exec** umieszczamy fragment kodu, który wykonuje pewne czynności przygotowawcze.

A oto pełny przykład (*proc_exec.c*):

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include "err.h"

int main ()

{

  pid_t pid;

  /* wypisuje identyfikator procesu */

  printf("Moj PID = %d\n", getpid());

  /* tworzy nowy proces */

  switch (pid = fork()) {
    case -1: /* blad */
      syserr("Error in fork\n");
    case 0: /* proces potomny */
      printf("Jestem procesem potomnym. Moj PID = %d\n", getpid());
      printf("Jestem procesem potomnym. Wartosc przekazana przez fork() = "%d\n", pid);
      /* wykonuje program ps */
      execlp("ps", "ps", 0);
      syserr("Error in execlp\n");
    default: /* proces macierzysty */
      printf("Jestem procesem macierzystym. Moj PID = %d\n", getpid());
      printf("Jestem procesem macierzystym. Wartosc przekazana przez fork() = %d\n", pid);
      /* czeka na zakonczenie procesu potomnego */
      if (wait(0) == -1)
        syserr("Error in wait\n");
     return 0;
  } /*switch*/
}
```

Ćwiczenie

Skompiluj *proc_exec.c* (już zrobione – **make**) i wykonaj go. Spróbuj zmienić **ps** na inny program wywoływany z argumentami.

Zwróć uwagę na sposób obsługi błędow funkcji exec. Dlaczego wywołanie funkcji syserr jest bezwarunkowe?

Ćwiczenie

Wykonaj powyższy program przekierowując wyjście do pliku. Obejrzyj plik z wynikami. Czy zawiera on wszystkie komunikaty wypisywane przez program? Czego nie ma? Dlaczego?

Ćwiczenie

Przeanalizuj *simple_shell.c* i wykonaj **./simple_shell.**
Uwaga: wyjście z programu przez Ctrl-D. Jest to prosty interpreter poleceń. Potrafi wykonać tylko polecenia zewnętrzne, czyli takie, które może zlecić innemu procesowi.
Spróbuj dodać do niego możliwość wykonywania procesów w tle, czyli takich procesów, na które nie czeka interpreter poleceń.

6. Kończenie procesu

Proces może spowodować zakończenie samego siebie przez wywołanie funkcji:

void exit(int kod_zakonczenia);

W przypadku poprawnego zakończenia, kod zakończenia powinien byc równy 0. Jeśli chcemy zasygnalizować błędne zakończenie programu, to używamy wartości różnych od 0.

7. Funkcja system()

Oprócz pary funkcji **fork-exec** można użyć funkcji

system(),

która powoduje wywołanie interpretera poleceń (np. /bin/bash) i przekazanie do niego jako argumentu argumentu funkcji system. Jest to mniej efektywne niż para funkcji fork-exec, bo powoduje powstanie dodatkowego procesu (interpretera poleceń).

INFORMACJE DODATKOWE

O funkcjach systemowych

Każda z funkcji, które tutaj omawiamy wymaga pewnych działań systemu operacyjnego, a dokładniej - wykonania funkcji systemowych.
Pisząc program nie używamy bezpośrednio funkcji systemowych, ale odpowiadających im funkcji bibliotecznych, które wywołują właściwa funkcje i wykonuje pewne dodatkowe czynności.

Funkcja systemowa nie jest zwykłym wywołaniem funkcji. Jest wykonanie oznacza przejście w tryb uprzywilejowany i zlecenie systemowi operacyjnemu wykonania pewnych czynności na rzecz procesu.
Ze względu na przekraczania granic ochrony jest to droższe niż zwykle wywołanie funkcji. Pamiętaj o tym pisząc programy!

Każda funkcja systemowa przekazuje swój kod zakończenia. Jest to 0, jeżeli funkcja zakończyła się pomyślnie lub liczba ujemna oznaczająca kod błędu w przeciwnym przypadku. Funkcja z biblioteki C, która wywołuje funkcje systemowa sprawdza, czy nie nastąpił błąd i jeżeli tak, to przypisuje wartość blefu na globalna zmienna errno i przekazuje w wyniku -1.
Dzięki kodowi blefu uzyskujemy więcej informacji o powodach wystąpienia danego blefu. Przykład wykorzystania zmiennej errno można znaleźć w funkcji syserr w pliku err.c, która korzysta z globalnej tablicy sys_errlist zawierającej opisy wszystkich kodów błędów.

W dalszym ciągu zajęć będziemy używać pewnego skrótu myślowego, a mianowicie będziemy nazywać funkcja systemowa odpowiednia funkcje biblioteczna - na przykład funkcja systemowa fork().

Zalecamy używania funkcji syserr do obsługi błędów funkcji systemowych.

O plikach nagłówkowych

a) Plik nagłówkowy `unistd.h' zawiera deklaracje standardowych funkcji uniksowych (fork(), write(), etc.). Warto go dołączyć do programu, aby kompilator nie generował ostrzeżeń takich jak, "implicit declaration of function `fork'".
b) Deklaracja funkcji wait() znajduje sie w `sys/wait.h'.

ZADANIA LABO10/11 DO DOMU NA PUNKTY

1. Napisz program tworzący "linię" 5 procesów, w której każdy proces potomny jest przodkiem następnego procesu (a więc pierwszy proces jest ojcem drugiego, dziadkiem trzeciego itd). Każdy proces macierzysty przed zakończeniem swojego działania powinien zaczekać na zakończenie swojego potomka. Pamiętaj o obsłudze błędów! Możesz wykorzystać do tego celu funkcje z pliku err.c

2. Zapoznaj się z poleceniami ps, job, fg i bg, oraz kill i killall i top oraz htop.

   - Napisz skrypt, który unicestwi 3 wypisany proces.
   - Napisz skrypt, który unicestwi 2 losowe procesy.
   - Napisz skrypt, który policzy ile procesów jest w stanie sleep (zapoznaj się ze wszystkimi rodzajami stanów procesów w systemie LINUX).

   Przydatne mogą być polecenia między innymi: head, tail, cut, ps, sort -r, seq,.

3. Napisz program, który wyświetli swój komunikat "hello world" oraz swój pid, oraz pid swojego procesu macierzystego.

4. Wykorzystując funkcję fork napisz program, który utworzy 1 proces potomny. Proces macierzysty ma wyświetlić swój pid, pid własnego procesu macierzystego, oraz pid procesu potomnego (skąd go wziąć?). Proces potomny ma wyświetlić swój pid, pid własnego procesu macierzystego, oraz wartość zwracaną przez funkcję fork.
5. Tak zmodyfikuj program z zadania 4, aby proces macierzysty czekał, aż zakończy się proces potomny, i wyświetli informację o tym, że proces zostanie zakończony. Wykorzystaj funkcję wait.
6. Zmodyfikuj program z zadania 4, aby proces macierzysty utworzył zadaną liczbę procesów (podaną z klawiatury) i zaczekał, aż każdy z procesów wypisze swój pid i zakończy się.
7. Napisz wielowątkową aplikację w języku JAVA, która utworzy 10 wątków o numerach N od 1 do 10, i każdy z tych wątków śpi przez N sekund a następnie zakończy się.Uzyskaj informacje o utworzonych wątkach polecenia ps oraz top lub htop.
8. Napisz program, który co sekundę zmienia swój pid na inny.