Spis treści

Na podstawie:

- *M.J.Rochkind - Programowanie w systemie UNIX dla zaawansowanych (Advanced UNIX Programming)*
- `https://ai.ia.agh.edu.pl/wiki/pl:dydaktyka:so:2017:labs:lab_intro`
- *Graham Glass, King Ables - Linux dla programistów i użytkowników*

1 Pojęcia podstawowe potrzebne do zrozumienia tematu

1.1. Definicje podstawowe

Program
Program to zbiór instrukcji i danych trzymanych w pliku regularnym na dysku, w swoim i-node jest on oznaczony jako executable.

Proces
Proces oznacza po prostu działający program. Np. gdy wpisujemy w konsoli 'pwd' uruchamiamy tym samym program, program ten podczas swojego działania jest nazywany procesem. Proces składa się z: segmentu instrukcji, segmentu danych użytkownika, segmentu danych systemowych. Program jest używany aby zainicjalizować dane użytkownika i segment instrukcji. Dane systemowe to np. deskryptory otwartych przez proces plików, informacje o obszarze pamięci, obecność tych danych systemowych stanowi ważną różnicę pomiędzy programem, a procesem.

Wątek
Wątki to osobne przepływy wykonywania instrukcji dla jednego procesu. Wątki mogą zapisywać i czytać te same dane, każdy z nich ma jednak osobny stos.

Process ID (PID)
Linux zarządza procesami używając tzw. Process ID, w skrócie PID. PID to 5-cyfrowa liczba przyporządkowana do procesu. System gwarantuje, że liczba ta jest unikalna dla danego procesu. Jeśli dany proces się zakończy, to jego PID może być ponownie użyte dla innego procesu, ale w danym punkcie czasu nie mogą istnieć dwa procesy o tym samym PID.

ps
ps to komenda wyświetlająca wszystkie działające procesy. Aby wyświetlić więcej informacji należy dodać flagę -f: *ps -f*. Aby wyświetlić dane procesu o danym id możemy użyć: *ps PID*, przykładowo: *ps 19*.

Zarządca pamięci, planista
Kilka działających równolegle procesów rywalizuje o dostęp do procesora i pamięci RAM. *Planistą* nazywamy część jądra odpowiedzialną za przydzielanie procesora. *Zarządcą pamięci* nazywamy część jądra przydzielającą pamięć RAM.

Potoki
Potok jest to mechanizm umożliwiający przekazywanie wyniku jednego procesu jako dane wejściowe kolejnego. Przykładowo *ls | wc*.

- Czym jest program ?

- Czym jest proces ?
- Czym jest wątek ?
- Czym jest PID ?
- Jak wyświetlić informację o procesach, a jak o pojedynczym procesie ? Co daje flaga -f ?
- Czym jest zarządca pamięci, a czym planista ?
- Czym jest potok ?
- Co odróżnia proces od programu ?

1.2. Procesy i pliki wykonywalne

Jak wiemy aby uruchomić program należy najpierw go skompilować, w wyniku czego powstaje plik wykonywalny. Proces jest działającym programem, a więc odnosi się do wykonywania instrukcji pliku wykonywalnego. Linux uruchamia program tworząc proces, a następnie wiążąc go z określonym plikiem wykonywalnym.
Co ważne, w Linuxie **nie ma wywołania systemowego tworzącego nowy proces**. Zamiast tego powielane są istniejące procesy, a następnie wiązane z plikiem wykonywalnym odpowiednim dla procesu.
Przy takiej operacji kopiowania procesów, proces pierwotny nazywany jest rodzicem nowego procesu - procesu potomnego. W informacjach wyświetlanych przez polecenie *ps -f* można zauważyć pole PPID oznaczającego PID procesu rodzica.

- Jak tworzone są nowe procesy w Linuxie ?
- Czym jest proces potomny, a czym proces rodzic ?
- Czym jest PPID ?

1.3. Pierwsze procesy startowe, sched, init

Pierwszym uruchamianym procesem, który tworzony jest podczas uruchamiania systemu jest *sched*, proces ten ma PID równe 0. Proces ten natychmiast tworzy proces potomny *init*, który ma PID równe 1. Wszystkie inne procesy są potomkami procesu init, tworzą hierarchię na początku której znajduje się init.

- Czym jest sched i init ? Jakie mają PID ?
- Z jakiego procesu wywodzą się wszystkie inne procesy (poza sched) ?

1.4. Sposoby uruchomienia procesu

Proces może być uruchomiony na dwa sposoby:

- Na pierwszym planie (Foreground Process) - jest to defaultowy sposób, jeśli proces działa w tym trybie, to żaden inny proces nie może wtedy działać lub być uruchomiony. Przykładowo, polecenie: *ls pwd*, działa na pierwszym planie, prompt dla uruchomienia innych procesów jest niedostępny do czasu zakończenia działania procesu.
- W tle (Background Process), tzw. zadania - Działa w tle, może być wykonywany podczas działania innych procesów. Aby uruchomić proces w takim trybie należy dodać do komendy znak ampersand '&', przykładowo: *pwd &*. Procesy takie są zwłaszcza wykorzystywane gdy mamy do dyspozycji tylko terminal tekstowy, bez interfejsu graficznego, wtedy aby móc uruchomić kilka procesów naraz konieczne jest ich uruchamianie w tle. Zadania działające w tle można wyświetlić używając polecenia *jobs*.

1.5. Zarządzanie działaniem procesów pierwszoplanowych i zadań

Procesy działające w tle nie wyświetlają wyników po zakończeniu działania. Należy je przenieść na pierwszy plan aby uzyskać wynik. Zadanie można przenieść na pierwszy plan używając polecenia *fg numer_zadania*.
Proces pierwszoplanowy można przenieść w tło najpierw zawieszając proces przy użyciu klawiszy CTRL+Z, a następnie korzystając z polecenia *& bg numer_zadania*.

- Czym są zadania ? Po co się je wykorzystuje ?
- Jak uruchomić zadanie w tle ?
- Do czego służy polecenie jobs ?
- Jak przenieść proces z pierwszego planu do tła ?
- Jak przenieść proces z tła na pierwszy plan ?

1.6. Polecenia powłoki kill i nice

kill -sygnal pid

kill to polecenie wysyłające podany sygnał do podanego procesu, domyślnie wysyłany jest sygnał TERM, który powoduje przerwania działania procesu. Aby uzyskać listę sygnałów należy użyć kill -l. Aby mieć pewność, że działanie procesu zostało zakończone wyślij sygnał -9 (SIGKILL).

nice option command

Wywołanie samego polecenia *nice* powoduje wyświetlenie defaultowego priorytetu nowo tworzonego procesu. Priorytet ten można zmieniać przy pomocy polecenia nice, co ma wpływ na przydział procesora do procesów. Przykładowo *nice -n13 pico*, ustawia priorytet na 13, po n należy wpisać priorytet. Zakres możliwych priorytetów do ustawienia to -20(najwyższy) do 19(najniższy).
UWAGA: W przypadku nice, im mniejsza liczba tym większy priorytet.

- Do czego służy kill ?
- Do czego służy nice ?
- Jaki sygnał wysłać aby mieć pewność, że zakończymy proces ?
- Jaki jest zakres priorytetów dla nice ? Jaki jest najwyższy możliwy priorytet ?

2 Dane wejściowe do programu - zmienne środowiskowe

2.1. Tablica environ

Jak wiemy, do programu w C/C++ możemy przekazać dane jako argumenty dla funkcji main:

```
int main(
  int argc, /* argument count */
  char *argv[] /* array of argument strings */
)

// lub wywolac program bez argumentow:
int main(void)
```

Programy działające w systemie Linux mogą mieć również dostęp do zmiennych środowiskowych, które przekazywane są do programu w postaci tablicy *environ*. Używamy jej w następujący sposób:

```
extern char **environ;
int main(void) {
  int i;
    for (i = 0; environ[i] != NULL; i++)
      printf("%s\n", environ[i]);
    exit(EXIT_SUCCESS); }
```

2.2. Funkcje standardowe getenv, setenv

2.2.1. getenv

Częściej interesuje nas wartość konkretnej zmiennej środowiskowej, aby ją uzyskać możemy skorzystać z funkcji *getenv*.

```
#include <stdlib.h>
char *getenv(
  const char *var /* variable to find */
); /* Returns value or NULL if not found (errno not defined) */
```

Przykład odczytania wartości zmiennej środowiskowej LOGNAME:

```
int main(void) {
  char *s;
    s = getenv("LOGNAME");
    if (s == NULL)
      printf("variable not found\n");
    else
      printf("value is \"%s\"\n", s);
    exit(EXIT_SUCCESS); }
```

2.2.2. setenv

Funkcja ta służy do ustawiania wartości zmiennej środowiskowej lub tworzenia zmiennej środowiskowej w przypadku gdy zmienna o podanej nazwie jeszcze nie istnieje.

```
#include <stdlib.h>
int setenv(
  const char *var, /* variable to be changed or added */
  const char *val, /* value */
  int overwrite /* overwrite? */
); /* Returns 0 on success or -1 on error (sets errno) */
```

Flaga overwrite ustawiona na cokolwiek innego niż 0 powoduje, że jeśli zmienna o podanej nazwie już istnieje to zostanie nadpisana.
Przykład wywołania

```
1 // set environment variable _EDC_ANSI_OPEN_DEFAULT to "Y"
2    setenv("_EDC_ANSI_OPEN_DEFAULT","Y",1);
```

- Czym jest tablica environ ?
- Do czego służą funkcję getenv, setenv ?

3 exec System Calls

Proces może zastąpić swój aktualnie wykonywany kod, dane i stos tymi pochodzącymi z innego pliku wykonywalnego. Do przeprowadzenia takich operacji służą funkcję z rodziny *exec*. exec nie jest funkcją, ale rodziną funkcji, które służą do wykonania takich działań. Funkcje z tej rodziny nie modyfikują PID ani PPID, program się zmienia, podczas gry proces pozostaje ten sam.
Dla porównania, funkcja *fork* omówiona w dalszej części opracowania tworzy nowy proces, który jest klonem istniejącego poprzez skopiowanie instrukcji, danych systemowych i użytkownika.

3.1. Rodzina funkcji exec

Rodzina funkcji exec składa się z 6 funkcji, które są postaci execAB, gdzie A jest zastępowane przez literę 'l' lub 'v', a B przez literę 'p' lub 'e'.

- Litera 'l' oznacza, że argumenty będą przekazane oddzielnie, a litera 'v', że jako tablica.
- Litera 'p' oznacza, że zmienna środowiskowa PATH powinna zostać użyta do znalezienia programu (przydatne przy pisaniu powłoki), a 'e', że określone środowisko powinno zostać użyte.

Wszystkie możliwe funkcje to execl, execv, execlp, execvp, execle, execve.

3.2. Funkcja execl

```c
#include <unistd.h>
int execl(
  const char *path, /* program pathname */
  const char *arg0, /* first arg (file name) */
  const char *arg1, /* second arg (if needed) */
  ..., /* remaining args (if needed) */
  (char *)NULL /* arg list terminator */
); /* Returns -1 on error (sets errno) */
```

Ścieżka musi być do programu który jest executable i do którego użytkownik ma prawa wykonania. Po wykonaniu funkcji proces rozpoczyna wykonywanie instrukcji z przekazanego pliku zaczynając od funkcji main z tego pliku.
execl w przypadku powodzenia nic nie zwraca, a w przypadku niepowodzenia zwraca -1 i ustawia errno.
Ostatnim argumentem musi być NULL, oznacza koniec podawania argumentów.

Kolejnymi argumentami funkcji (arg0, arg1, ..., argN) są dane, które przekazujemy do nowego programu i które mogą być użyte przy pomocy zmiennych argc oraz argv tego programu. Konwencją jest, że jako pierwszy argument podaje się nazwę obecnie wykonywanego programu.

Dane systemowe procesu pozostają nienaruszone więc np. priorytet, deskryptory otwartych plików itp. pozostają takie same.
Przykładowe wywołanie:

```c
execl("/bin/echo", "echo", "the", "lazy", "dogs.", (char *)NULL)
```

Wypisuje *the lazy dogs.*
Inne możliwe deklaracje funkcji z rodziny exec:

```c
#include <unistd.h>
int execv(
  const char *path, /* program pathname */
  char *const argv[] /* argument vector */
```

```
); /* Returns -1 on error (sets errno) */


#include <unistd.h>
int execlp(
  const char *file, /* program file name */
  const char *arg0, /* first arg (file name) */
  const char *arg1, /* second arg (if needed) */
  ..., /* remaining args (if needed) */
  (char *)NULL /* arg list terminator */
); /* Returns -1 on error (sets errno) */
```

- Do czego służą funkcje z rodziny exec ?
- Co zmienia, a czego nie zmienia wywołania funkcji z rodziny exec ?
- Co zwraca funkcja exec ?
- Jakie argumenty przekazywane są do funkcji exec ? Co musi być ostatnim argumentem ?

4 Funkcja fork

fork jest w pewien sposób przeciwieństwem exec - tworzy nowy proces, ale nie zmienia wykonywanego programu. fork jest w pewnym sensie dziwnym wywołaniem, ponieważ jest wywoływane przez jeden proces (macierzysty), a powstają z niego dwa procesy (macierzysty i potomny). Obydwa procesy równolegle wykonują ten sam kod, ale mają zupełnie oddzielne stosy i przestrzenie danych.
Jeżeli fork zakończy się powodzeniem to zwraca PID procesu potomnego do rodzica, a do procesu potomnego zwraca 0. Jeżeli zawiedzie, to zwraca -1 do procesu rodzica, a proces potomny nie jest tworzony.

```
#include <unistd.h>
pid_t fork(void);
/* Returns child process-ID and 0 on success or -1 on error (sets errno) */
```

Przykład działania fork:

```
#include<stdio.h>
main()
{
  int pid;
  printf("Jestem procesem pierwotnym o PID rownym %d i PPID rownym %d.\n", getpid(),
    getppid());

  pid = fork(); // powiel. Proces macierzysty i potomny kontunuuja od tego miejsa

  if(pid !=0) // wartosc zmiennej pid jest niezerowa, wiec musze byc rodzicem
  {
    printf("Jestem procesem macierzystym o PID rownym %d i PPID rownym %d.\n", getpid(),
     getppid());
  }
  else
  {
    printf("Jestem procesem pierwotnym o PID rownym %d i PPID rownym %d.\n", getpid(),
    getppid());
  }

}
```

- Jak działa funkcja fork ?
- Co zwraca funkcja fork ?

5 Command chaining

Pozwala na warunkowe wykonywanie poleceń shella w zależności od powodzenia innych. Np.

- cmd1 && cmd2 - działa analogicznie jak operator && w C, jeśli stwierdzi, że cmd1 daje 0, to nie sprawdza cmd2 (cmd2 nie jest wołane), bo wiadomo, że wyjdzie 0. cmd2 jest wołane tylko w przypadku gdy cmd1 da 1.
- cmd1 || cmd2 - Jeśli cmd1 da 1, to nie jest wołane cmd2 (bo wiadomo, że całość da 0), jeśli cmd1 da 0, to jest wołane cmd2.

- Jak działają operatory && oraz || przy chainowaniu komend shella ?

6 Funkcje wait, waitpid

Często w programie zdarza się, że chcemy poczekać na zakończenie działania procesu dziecka w procesie rodzicu. Np. obsługujemy znak prompt w powłoce, chcemy wyświetlać prompt po wykonaniu polecenia, a w celu wykonania polecenia tworzymy nowy proces. Wtedy w procesie rodzicu, który wyświetla znak prompt musimy poczekać na zakończenie procesu dziecka, aby przypadkiem nie wyświetlić znaku prompt przed zakończeniem działania polecenia. W tym celu możemy użyć funkcji wait lub waitpid.

6.1. Funkcja waitpid

```
#include <sys/wait.h>

pid_t waitpid(
  pid_t pid, /* process or process-group ID */
  int *statusp, /* pointer to status or NULL */
  int options /* options (see below) */
); /* Returns process ID or 0 on success or -1 on error (sets errno) */
```

Funkcja ta wywołana w procesie rodzicu będzie czekać na proces/y dzieci.
Argument pid służy do konfiguracji tego na które procesy chcemy czekać. Jeśli chcemy czekać na proces o konkretnym id, to wystarczy przekazać do funkcji to pid.
Możemy też przekazać jako pierwszy argument:

- -1 - czekaj na jakikolwiek proces potomny
- 0 - czekaj na jakikolwiek proces z tej samej grupy procesów
- Mniejsze od -1 - czekaj na jakikolwiek proces dziecko w grupie procesów którego pid jest równe -pid procesu macierzystego

Argument status, to po prostu wskaźnik na zmienną int. W zmiennej tej zostanie zapisany status zakończenia procesu dziecka.
Ostatni argument używany jest do wskazania, które zmiany w stanie procesu dziecka mają być zgłaszane do procesu rodzica. Tj. proces rodzic może oczekiwać na określoną zmianę procesu dziecka, np. zawieszenie dziecka, ponowne wznowienie działania dziecka itp. - nie tylko na jego zakończenie. Argument ten ustawiany jest często przy użyciu flag:

- WCONTINUED - zgłaszaj kontynuację i zakończenie działania dziecka
- WUNTRACED - zgłaszaj zastopowane procesy oraz te zakończone

6.2. Funkcja wait

Funkcja wait jest jedynie skrótem dla wywołania waitpid z pid ustawionym na -1(czekaj na jakikolwiek proces dziecko) i bez żadnych opcji, w związku z czym przekazujemy tylko wskaźnik na zmienną int gdzie chcemy zapisać status procesu dziecka.

```
#include <sys/wait.h>
pid_t wait(
  int *statusp, /* pointer to status or NULL */
); /* Returns process ID or -1 on error (sets errno) */
```

7 Inne pytania z wejściówek

- Jakie są warunki usunięcia pliku ?
- Czy można utworzyć dowiązanie do pliku znajdującego się na innym urządzeniu ?