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 Teoria do laboratorium

1.1. UNIX/Linuks - Podstawowe pojęcia

1.1.1. Pliki

Plik
Odpowiada ze przechowywanie danych, wszystko jest plikiem, pliki istnieją w systemie plików. Są różne rodzaje plików: pliki regularne, foldery, linki symboliczne, pliki specjalne, nazwane potoki (FIFOs) i gniazda (sockets).

Pliki regularne
zawierają bajty danych, dane mogą być dopisywane i czytane z pliku. Pliki nie mają nazw, ale indeksy (i-number), które są indeksami w tablicach **i-node**. Każdy i-node zawiera informacje o pliku takie jak: typ pliku(regular, directory, socket itd.), liczbę dowiązań, właściciela, grupę, rozmiar, czas modyfikacji.

Katalogi i linki symboliczne
nie byłoby wygodnie odwoływać się do plików przez i-number, **katalogi** umożliwiają posługiwanie się nazwami plików. Każdy katalog składa się z tabeli o dwóch kolumnach, w jednej znajduje się i-number, a w drugiej nazwa pliku. Dostęp do pliku jest realizowany przez znalezienie i-number odpowiadającego nazwie pliku w katalogu, a następnie i-node odpowiadającego i-number. Para nazwa/i-node nazywana jest linkiem. **Linkiem symbolicznym** nazywamy plik w systemie plików, który wskazuje odwołując się za pomocą nazwy, na dowolny inny plik lub katalog.

Pliki specjalne
to typowo jakieś urządzenie (np. dysk CD-ROM)

1.1.2. Program, proces, wątek, sygnały

Program
to zbiór instrukcji i danych trzymanych w pliku regularnym. Plik ten jest oznaczony jako executable w swoim i-node. Aby uruchomić program jądro rozpoczyna nowy proces.

Proces
proces jest środowiskiem w którym wykonywany jest program. Kilka procesów może być uruchomione współbieżnie z jednego programu, ale wówczas nie ma pomiędzy nimi żadnego powiązania. Dane systemowe procesu zawierają między innymi takie atrybuty jak obecny katalog, czas CPU itp. Proces tworzony jest przez jądro z obecnie działającego procesu, który staje się rodzicem nowo powstałego procesu. Proces dziecko dziedziczy większość atrybutów systemowych rodzica, w przeciwieństwie do wątków dla których brak takiego dziedziczenia. Każdy proces ma swoje ID, które jest dodatnią liczbą całkowitą. Każdy proces poza jednym (poza init) ma swojego rodzica i przechowuje jego ID w informacjach o procesie.

Grupa procesów
jest to grupa powiązanych ze sobą procesów implementujących pewien podsystem.

Wątek
podczas działania procesu jądro nadzoruje pracę jego wątków, które są oddzielnymi częściami kontrolującymi wykonywanie instrukcji. Proces może mieć wiele wątków. Każdy wątek ma swój własny stos.

Sygnały Jądro może wysyłać sygnały do procesów. Sygnał może być również wysłany do procesu przez proces. Np. sygnał zakończenia wątku może zostać wysłany z jednego procesu do drugiego. Proces może kontrolować co stanie się po otrzymaniu sygnału chyba, że otrzyma sygnał KILL lub STOP. Proces może zaakceptować defaultową akcję, zignorować sygnał lub odebrać sygnał i wykonać funkcję.

1.1.3. Prawa dostępu

userID
to liczba dodatnia powiązana z nazwą użytkownika z pliku */etc/passwd*.

groupID
użytkownicy są zorganizowani w grupy, które także mają przypisane id zwane Group-ID. Grupy są zdefiniowane w */etc/group*. Użytkownik może mieć przypisane wiele grup. Każdy plik w swoim i-node przechowuje id grupy i użytkownika odpowiadające właścicielom pliku.

Prawa dostępu
każdy plik ma przyporządkowane prawa dostępu przechowywane w i-node (odczyt, zapis, wykonywanie). Prawa te często są wyrażane w formacie ósemkowym (liczba od 0 do 7), której suma odpowiada prawom dla kolejno właściciela, grupy oraz innych. r(read) -> 4, w(write) -> 2, x(execute) -> 1, np. 775.

1.1.4. Komunikacja międzyprocesowa

Komunikacja między procesowa może być realizowana przez wiele mechanizmów, m.in. semafory, sygnały, potoki, pamięć współdzieloną, gniazda.

...

1.2. Korzystanie z dokumentacji UNIX - man

Wraz z systemem GNU/Linux dostarczana jest bardzo obszerna dokumentacja systemu i oprogramowania. Man jest zbiorem tradycyjnych podręczników dostępnych w każdym systemie UNIX.

1.2.1. Sekcje podręcznika

Podręczniki dzielą się na następujące numerowane kategorie, wszystkich jest 9:

1. programy wykonywalne, polecenia shell'a
2. opis funkcji systemowych udostępnianych przez jądro – ta i następna kategoria jest szczególnie przydatna podczas programowania w języku C na poziomie systemu, a więc i na zajęciach.
3. opis funkcji bibliotecznych udostępnianych przez biblioteki systemowe.

4. opis plików specjalnych, głównie związanych z obsługą urządzeń.
5. formaty plików systemowych i konwencje ich tworze-nia – te informacje są szczególnie przydatne dla administratora systemu.
6. gry - gry
7. makro pakiety i konwencje - makro pakiety i konwencje w nich stosowane – taczęść opisuje pakiety takie jak Man, czy Groff, ich formaty plików i organizację.
8. polecenia do administrowania systemem - polecenia do administrowania systemem – częśćbardzo istotna dla administratora, większość opisanych tu komend dostępna jest tylko dlaniego.
9. procedury jądra - niestandardowe procedury jądra Linuxa – w tej części umieszcza sięopisy niestandardowych rozszerzeń funkcjonalności jądra Linuxa.

1.2.2. Części strony podręcznika

Wszystkie strony man składają się z kilku podstawowych części. Najważniejsze z nich to:

- NAME nazwa i krótki opis strony,
- SYNOPSIS spis wszystkich dostępnych opcji z linii poleceń,
- DESCRIPTION dokładny opis hasła,
- OPTIONS dokładny opis opcji z linii poleceń,
- FILES opis plików używanych przez opisywane polecenie lub system,
- SEE ALSO odnośniki do innych stron podręcznika powiązanych tematycznie z opisywanymhasłem,
- BUGS wykryte błędy w opisywanym poleceniu lub systemie,
- AUTHOR informacje o autorze lub autorach.

1.2.3. flagi dla man

Aby wyświetlić stronę podręcznika na temat danej funkcji należy użyć

man nazwaPolecenia, np. man read

Ze strony podręcznika wychodzimy klawiszem 'q'.

Podręcznik może zawierać kilka stron w zależności od kategorii. Aby zobaczyć wszystkie możliwe kategorie w których występuje polecenie, a także krótkie opisy poleceń wpisujemy:

man -f nazwaPolecenia, np. man -f read
wynik to polecenia wraz z numerami sekcji i krótkimi opisami, przykładowo:
read(2) - read from a file descriptor

Aby wybrać stronę z konkretnej sekcji należy podać jej numer np.:

man 2 read

Aby wyszukać na danej stronie podręcznika fragment tekstu:

- Otwórz stronę podręcznika, np. man gcc
- Naciśnij / i wpisz tekst do wyszukania np. /-Wall i naciśnij enter
- Przejście do następnego znalezionego miejsca to naciśnięcie klawisza 'n'.

1.3. Kompilacja programu - gcc i make

UNIX/Linuks posiadają wbudowane kompilatory języka C.

Program jednomodułowy
to program składający się z jednego pliku z rozszerzeniem .c. Program taki można skompilować przy pomocy polecenia gcc w wyniku którego powstanie plik z kodem maszynowym (plik wykonywalny).

Program wielomodułowy
to program składający się z wielu plików z rozszerzeniem.c oraz z plików z rozszerzeniem .h. Pliki z rozszerzeniem .h to pliki zawierające prototypy, a pliki o takiej samej nazwie z rozszerzeniem.c, to pliki zawierające implementację tych prototypów. Dzięki temu te same funkcje mogą być używane w wielu programach. Plik implementujący prototypy musi inkludować odpowiadający mu plik.h, a także bibliotekę standardową stdio.h. Plik z programem głównym (main) musi inkludować tylko plik z deklaracjami .h.

1.3.1. Kompilacja programu jednomodułowego

W przypadku programów jednomodułowych wystarczy użyć polecenia gcc aby otrzymać plik gotowy do wykonania. Do polecenia gcc można dodawać opcję (flagi) takie jak:

- -Wall - włącza wyświetlanie ostrzeżeń (warnings) podczas kompilacji, ostrzeżenia dotyczą konstrukcji które są względnie poprawne.
- -ansi - włącza standard języka C90, wyłącza pewne cechy języka niekompatybilne z C90, jeśli używamy C++, to analogicznie dla wersji C++98
- -pedantic - wprowadza bardziej restrykcyjny sposób kompilacji w relacji do możliwości wystąpienia błędów w kodzie
- -c - opcja używana przy kompilacji programów wielomodułowych w celu osobnej kompilacji wielu modułów.

Przykładowe polecenie kompilujące program jednomodułowy do pliku wykonywalnego to:

gcc -Wall -pedantic hello.c

Rezultatem jest utworzenie pliku wykonywalnego a.out. Aby utworzyć plik wykonywalny o określonej nazwie można użyć flagi -o:

gcc -Wall -pedantic hello.c -o hello

Rezultatem jest utworzenie pliku wykonywalnego hello.
Aby uruchomić program należy wpisać w konsolę jego nazwę lub poprzedzić nazwę znakami ./nazwa. Np. ./hello

1.3.2. Kompilacja programu wielomodułowego - bez make

Programy wielomodułowe można kompilować razem dla wszystkich modułów lub oddzielnie dla każdego modułu.

Oddzielnie
w tym przypadku należy zastosować opcję -c, pozwoli to utworzyć osobne moduły wynikowe, które następnie należy połączyć w całość.

gcc -c hello.c main.c ...kompilacja każdego pliku z osobna do hello.o i main.o
gcc hello.o main.o -o hello ...połączenie w jeden plik wykonywalny hello

1.3.3. Kompilacja programu wielomodułowego - make

Make jest narzędziem wspomagającym proces budowania wielomodułowego kodu ze złożonymi zależnościami pomiędzy plikami źródłowymi. W ogólnym przypadku jest przydatny tam, gdzie ma miejsce tworzenie plików wynikowych ze źródłowych, a zmiany w jednych pociągają rekurencyjnie zmiany w kolejnych.

Instrukcje dotyczące kompilacji zawierane są w pliku o nazwie 'makefile' lub 'Makefile', kompilację rozpoczyna się wpisując w konsoli make. Wywołanie tego poleceni powoduje kompilację plików i utworzeniu pliku wykonywalnego zgodnie z instrukcjami w pliku makefile. Instrukcje w pliku makefile mają składnię:

cel: plik1 plik2 ...
<TAB><TAB>*Instrukcja dla kompilatora*

Najpierw kompilujemy każdy plik w osobnej instrukcji do pliku z rozszerzeniem .o, a następnie łączymy je razem.

Przykład pliku makefile:

```
1 hello: main.o hello.o
2     gcc main.o hello.o -o hello -I.
3
4 hello.o: hello.c hello.h
5     gcc -c -Wall -ansi -pedantic hello.c -I.
6
7 main.o: main.c hello.h
8     gcc -c -Wall -ansi -pedantic main.c -I.
```

W wyniku wywołania make powstaje plik wykonywalny hello. Flaga -I. na końcu jest konieczna i oznacza poszukiwanie plików w obecnym katalogu.

1.4. Wywołania (funkcje) systemowe

1.4.1. Pojęcia podstawowe

Wywołania systemowe

aby skorzystać z usług takich jak tworzenie plików, powielanie procesów czy komunikacja międzyprocesowa, aplikacje muszą kontaktować się z systemem operacyjnym. Mogą to czynić poprzez zestaw procedur, zwanych wywołaniami systemowymi, które można potraktować jako interfejs pomiędzy programistą, a jądrem systemu operacyjnego (zasadniczo, są to funkcje biblioteczne, które wywołują odpowiednie wywołania systemowe).
Funkcje systemowe są jedynym możliwym narzędziem, które pozwala na korzystanie z możliwości jądra systemu takich jak system plików, mechanizmy wielozadaniowości, komunikacja między procesami. Funkcje systemowe definiują UNIX, cała reszta - procedury i polecenia - jest zbudowana na ich podstawie.

Wywołania funkcji systemowych

nie różnią się od wywołania innych funkcji, np.

amt = read(fd, buf, numbyte);

Każda funkcja systemowa jest zdefiniowana w pliku nagłówkowym. Należy dołączyć go zatem do programu. Niekiedy trzeba dołączyć więcej niż jeden plik, np. dla funkcji read konieczne jest zainkludowanie <unistd.h>.

Mechanizm wywoływanie funkcji systemowych

wywołanie funkcji systemowej powoduje przekazanie sterowania od funkcji wywołującej do jądra systemu, a następnie wynik jest zwracany do wywołującego ją procesu. Taki mechanizm powoduje, że wywołania funkcji systemowych są zazwyczaj dłuższe niż wywołania przeciętnych funkcji.

1.4.2. Obsługa błędów

Wartości zwracane

Funkcje systemowe i biblioteczne mają standardowy sposób komunikowania o błędzie, jednak nie każda funkcja sygnalizuje błąd tak samo! Zwracane wartości to np. wartość -1, NULL, czy stała symboliczna SIG_ERR. Wartości zwracane przez funkcje opisane są szczegółowo w manualu na stronie danej funkcji (części RETURN VALUE oraz ERRORS na danej stronie podręcznika).

errno

Gdy program sygnalizuje błąd, ważna jest także dla nas konkretna przyczyna tego błędu, a nie sam fakt jego zajścia abyśmy wiedzieli jak go obsłużyć. Sprawdzanie przyczyny błędu możliwe jest przy pomocy zmiennej **errno**. errno jest zmienną globalną, która przechowuje kod numeryczny ostatniego błędu funkcji systemowej. Pierwotna, ustawiana w momencie tworzenia procesu wartość errno wynosi 0. Kody poszczególnych błędów zdefiniowane są w pliku errno.h przy pomocy makr, można je też sprawdzić używając *man errno* - strony podręcznika nie odnoszą się jednak do numerów, a do nazw symbolicznych błędów jak np. E2BIG, EACCES. Aby móc korzystać z errno należy zawrzeć plik nagłówkowy errno.h. Np.

```
#include <errno.h>
if ((amt = read(fd, buf, numbyte)) == -1) {
 fprintf(stderr, "Read failed! errno = %d\n", errno);
 exit(EXIT_FAILURE);
}
```

Jeśli np. podamy do odczytu plik, który nie istnieje to errno będzie miało określoną wartość, a inną wartość w przypadku innego błędu.

Uwaga na errno

Nie wszystkie błędy funkcji systemowych ustawiają wartość errno, należy zawsze sprawdzić dokumentację funkcji. Nie powinno się sprawdzać wystąpienia błędu przez errno, ponieważ zmienna ta jest ustawiana dopiero po wystąpieniu błędu i tylko przez niektóre funkcje. Są wyjątki od tej zasady takie jak funkcje sysconf czy readdirm, które sygnalizację błędu opierają na zmianie wartości errno, ale nawet one zwracają specyficzną wartość, która każe Ci sprawdzić errno.

perror()

To zwykła funkcja biblioteczna języka C, która zmienia bieżącą wartość errno na łańcuch i wypisuje opis tekstowy błędu.

```
if ((amt = read(fd, buf, numbyte)) == -1) {
  perror("Read failed!");
  exit(EXIT_FAILURE);
}
\\ Now the output is:
\\ Read failed!: Bad file number
```

exit()

powoduje zakończenie wywołującego ją procesu. Jej argumentem jest status, który informuje o tym, czy program zakończył się poprawnie czy nie.

1.4.3. Funkcje read i open

open

*int open (char * nazwa_pliku, int tryb [, int prawa_dostępus]*
Funkcja open pozwala otworzyć lub utworzyć plik do odczytania i zapisu. tryb jest określany gdy tworzymy plik, jest to flaga określająca uprawnienia, np. O_RDONLY (tylko odczyt) lub O_RDWR (odczyt i zapis).

read

ssize_t read (int fd, void buf, size_t liczba)*
Funkcja read pozwala na odczyt określonej liczby bajtów z pliku. fd jest deskryptorem pliku z którego chcemy czytać, buf to wskaźnik bufora do zapisania danych, a liczba to liczba bajtów którą chcemy odczytać. Funkcja zwraca liczbę odczytanych bajtów lub 0 gdy napotka koniec pliku, gdy wystąpi błąd zwraca -1 i ustawia odpowiednio zmienną errno.

2 Rozwiązania

2.1. 1. Organizacja przedmiotu

-

2.2. 2. Korzystanie z dokumentacji systemu UNIX

Na podstawie strony manuala o manualu lub i tego artykułu proszę odpowiedzieć na pytania:

- Na jakie sekcje dzieli się manual?
  1. programy wykonywalne, polecenia shell'a
  2. opis funkcji systemowych udostępnianych przez jądro – ta i następna kategoria jest szczególnie przydatna podczas programowania w języku C na poziomiesystemu.
  3. opis funkcji bibliotecznych udostępnianych przez biblioteki systemowe.
  4. opis plików specjalnych, głównie związanych z obsługą urządzeń.
  5. formaty plików systemowych i konwencje ich tworze-nia – te informacje są szczególnie przydatne dla administratora systemu.
  6. gry - gry
  7. makro pakiety i konwencje - makro pakiety i konwencje w nich stosowane – taczęść opisuje pakiety takie jak Man, czy Groff, ich formaty plików i organizację.
  8. polecenia do administrowania systemem - polecenia do administrowania systemem – częśćbardzo istotna dla administratora, większość opisanych tu komend dostępna jest tylko dlaniego.
  9. procedury jądra - niestandardowe procedury jądra Linuxa – w tej części umieszcza sięopisy niestandardowych rozszerzeń funkcjonalności jądra Linuxa.
- Jakie są typowe części pojedynczej strony?
  1. NAME nazwa i krótki opis strony,
  2. SYNOPSIS spis wszystkich dostępnych opcji z linii poleceń,

3. DESCRIPTION dokładny opis hasła,
4. OPTIONS dokładny opis opcji z linii poleceń,
5. FILES opis plików używanych przez opisywane polecenie lub system,
6. SEE ALSO odnośniki do innych stron podręcznika powiązanych tematycznie z opisywanym-hasłem,
7. BUGS wykryte błędy w opisywanym poleceniu lub systemie,
8. AUTHOR informacje o autorze lub autorach.

- Jakie są konwencje stosowane w manualu? Co oznacza tekst pogrubiony, podkreślony, a co [argumenty w nawiasach]?

  1. Pogrubiony - konieczne
  2. Podkreślony - do zamiany na konkretny argument
  3. nawiasy kwadratowe - opcjonalne

- Jak przewijać strony manuala? Jak wyszukiwać konkretne słowo/kolejne wystąpienie tego słowa/poprzednie wystąpienie tego słowa?
  Wyszukiwanie słowa: man polecenie -> /słowo -> n - następne
- Co oznaczają opcje -a, -k, -f dla polecenia man?
- Co znajduje się w części ENVIRONMENT a co w FILES stron manuala?
- Co oznacza zapis nazwa_funkcji(nr)? Co należy wpisać, aby przeczytać opis takiej funkcji?
  Oznacza numer sekcji podręcznika w której występuje funkcja. Aby otworzyć ten opis wpisujemy komendę man (nr) nazwa_funkcji
- Jeżeli szukane polecenie/funkcja stanowi temat w kilku sekcjach manuala, a nie wyspecyfikujemy którą stronę chcemy czytać, to która strona zostanie wyświetlona?
- Jaki jest wynik poleceń apropos(1) i whatis(1)?

2.3. 3. Wykorzystanie funkcji systemowych i sprawdzanie błędów

Ćwiczenia

- Jaki plik nagłówkowy należy dołączyć do programu aby móc odczytać wartość zmiennej errno?
  <errno.h>
- Czym różnią się funkcje perror i strerror?
  Działanie jest takie samo, ale różnią się wywołaniem. strerror(errno), zwraca wskaźnik na wiadomość z błędem, podczas gdy perror(ęrror: ") wypisuje od razu taką wiadomość z dodatkowym przedrostkiem jako argument.
- Jakie dwie stałe języka C podaje się jako argument funkcji exit?
  EXIT_SUCCESS lub EXIT_FAILURE
- Proszę pobrać plik:

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

#define BUFSIZE 1024

int main (int argc, char **argv) {
    int f1, c;
    char b[BUFSIZE], *n1;

    c = 10;
    n1 = argv[1];

    f1 = open (n1, O_RDONLY);
    read (f1, b, c);
    printf("%s: Przeczytano %d znakow z pliku %s: \"%s\"\n",
     argv[0], c, n1, b);
    close(f1);

    return(0);
}
```

- Proszę zmodyfikować program tak, aby sprawdzana była wartość zwracana przez funkcje open(2) i read(2), program sprawdzał czy i jaki błąd wystąpił oraz informował o tym użytkownika, w przypadku błędu program wywoływał funkcję exit z odpowiednio ustawionym parametrem.

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include<stdlib.h>

#define BUFSIZE 1024

int main (int argc, char **argv) {
    int f1, c;
    char b[BUFSIZE], *n1;
  int bytesRead;

    c = 10;
    n1 = argv[1];

    f1 = open (n1, O_RDONLY);
  if(f1 == -1){
    perror("open failed: ");
    exit(EXIT_FAILURE);
  }

    bytesRead = read (f1, b, c);
  if(bytesRead == -1){
    perror("read failed: ");
    exit(EXIT_FAILURE);
  }
  else{
    if(bytesRead == 0){
      printf("read EOF");
    } else{
    printf("%s: Przeczytano %d znakow z pliku %s: \"%s\"\n",
       argv[0], c, n1, b);
    }
  }
    close(f1);

  exit(EXIT_SUCCESS);
```

```
38
39      return(0);
40  }
```

2.4. 4. Kompilowanie programów: gcc i make

2.5. 5. Daty i czasy w Unixie (o ile czas pozwoli)

2.6. 6. Standardy systemu (o ile czas pozwoli)

2.7. 7. Przykłady używane na laboratoriach

2.8. 8. Pytania kontrolne