Tajny Skrypt do Wykładu z UNIX-a
wersja 0.82

Marcin Benke

16 października 1998

Spis rzeczy

Ostrzeżenie

```
#include <std/disclaimer.h> ;-)
```

Ten dokument, zgodnie ze swym tytułem, nie tylko **nie jest oficjalnym skryptem**, ale co więcej — jest tajny.

Wiedza podawana jest tu wyrywkowo i skrótowo, w dużej części jest to po prostu inaczej sformatowana (albo w ogóle nie sformatowana) zawartość slajdów. Siłą rzeczy w wielu miejscach brakuje komentarzy, które wygłaszałem na wykładzie, ale które nie znalazły się na slajdach. Dlatego jego użyteczność dla osób które nie uczęszczały na wykład jest wątpliwa. Ale jeśli ktoś bardzo chce...

Rozdział 1

Wprowadzenie

1.1 Najkrótsza historia komputerów

- I Generacja (1945-55): Lampy i druty — żadnych systemów ani języków programowania, jeden użytkownik łaczacy druty i odczytujący wynik z mrugających lampek; komputery unikalne i bezcenne;
- II Generacja (1955-65): Tranzystory i systemy wsadowe — programy i dane na kartach dziurkowanych; zadania "przepuszczane przez maszyne" kolejno przez operatora, wydruki do odbioru często po kilku godzinach; cena przeciętnego komputera (np. IBM7094) — kilka milionów dolarów;
- III Generacja (1965-80): Układy scalone i współbieżność — wiele zadań wsadowych przetwarzanych równocześnie dla lepszego wykorzystania procesora (OS/360); początki systemów interakcyjnych (MULTICS, UNICS, minikomputery DEC PDP — ca. 120 000$)
- IV Generacja (1980-90): Układy VLSI i komputery osobiste — mikrokomputery jako stacje robocze albo jako tani wariant minikomputerów — ceny 1–20 tys. dolarów; CP/M, MS-DOS, Unix; systemy sieciowe i rozproszone.

1.2 Krótka historia Unixa

- Pierwsza wersja (1969) — Stworzona (samodzielnie) przez Kena Thompsona z Bell Labs na PDP-7 w języku B.
- Pod wrażeniem systemu stworzonego przez Thompsona, Dennis Ritchie stowrzył ulepszoną wersję języka B pod nazwą C, wspólnie z innymi pracownikami Bell Labs stworzyli UNIX na PDP-11; w 1974 otrzymali za to nagrodę Turinga.

- Unix był rozdawany uniwersytetom za symboliczną opłatą (wersja 6); eksplozja popularności w drugiej połowie lat 70-tych (wersja 7).
- W 1984 AT&T wypuszcza Unix System III, potem zaś system V R2, R3. Obecnie mamy R4.
- Równolegle University of California at Berkeley tworzy w oparciu o werję 6 własną wersję pod nazwą 1BSD. Istotne obecnie wersję to 4.3BSD i 4.4BSD
- IEEE wprowadza standardy Unixa pod nazwami Posix 1003.0 do 1003.10; ANSI standaryzuje język C
- W 1991 Linus Torvalds, student z Helsinek, pisze pierwszą werję *Linuxa*

1.3 Krótka historia Linuxa

- Sierpień 1991 — wersja 0.01 — tylko kernel
- 5 października 1991 — pierwsza "oficjalna" wersja (0.02) — bash, kompilator C (gcc)
- 1992 Wersje 0.10–0.12, a wkrótce potem 0.95
- Marzec 1994 Wersja 1.0
- Kwiecień 1995 Wersja 1.1
- Marzec 1995 Wersja 1.2 (aka Linux'95)
- Czerwiec 1995 Wersja 1.3
- Lipiec 1996 Wersja 2.0
- Obecnie (pażdziernik 98): wersje 2.0.36 (stabilna) oraz 2.1.125 (eksperymentalna)

1.4 System operacyjny

- System plików
- Obsługa urządzeń
- Procesy (programy)
- Zarządzanie zasobami
- Użytkownicy i zapewnianie bezpieczeństwa

- Komunikacja z użytkownikiem — shell i okienka
- Komunikacja ze światem — (nie tylko) inne komputery
- środowisko tworzenia aplikacji
- Inne narzędzia

Podstawowe funkcje systemu operacyjnego są w Unixie spełniane przez *jądro*, inne — przez wyspecjalizowane procesy, tzw. *demony*.

1.5 System plików

- Pliki
- Katalogi
- Pliki specjalne
- Dołączanie innych systemów plików

1.6 Katalogi i pliki specjalne

Katalogi są strukturą odwozorowującą nazwy plików (ścieżki dostępu) na faktyczne pliki. W Unixie katalogi są zrealizowane jako (wyróżnione) pliki. Każdy katalog może zawierać inne katalogi, co sugeruje strukturę drzewiastą, jednak...

...do każdego pliku może być wiele dowiązań, a zatem system plików ma postać dag-u

W Unixie urządzenia są widoczne jako pliki specjalne, co przyczynia się do przejrzystości budowy systemu i aplikacji, gdyż komunikacja z urządzeniami opiera się na tych samych zasadach co czytanie z i pisanie do plików.

Rozdział 2

Podstawy pracy z systemem

2.1 Przywitanie i pożegnanie z systemem

W celu wejścia do systemu należy podać swój identyfikator i hasło:

```
zodiac1 login: benke
Password:
Last login: Fri Mar  8 11:58:19 from bratek.mimuw.edu

zodiac1:~$
```

Aby wyjść z systemu najpewniej użyć polecenia `exit`.

Czasami wystarcza nacisniecie `Ctrl-d`, czasami system reaguje na ten klawisz komunikatem

```
zodiac1:~$ Use "logout" to leave the shell.
zodiac1:~$
```

Znaczy to że trzeba jednak użyć `exit`.

> **Pamiętaj:** nie zostawiaj na dłużej terminala na którym jesteś zalogowany — ktoś może narozrabiać “na Twoje konto”, albo zrobić Ci głupi dowcip.

2.2 Złe Hasła

Wybierz hasło łatwe do zapamiętania, trudne do zgadnięcia dla osoby postronnej. Oto przykłady “złych” haseł:

```
pawel1    EwaEwa   Kowalski
```

Hackerzy do łamania haseł używają oczywiście programów i słowników, więc **nie** są dobrymi hasłami słowa nawet w egzotycznych językach, albo zlepki słów, ich lustrzanych odbić i cyfr:

```
mellon  lewaPawE7   wyborowa   Prozac
```

2.3 Dobre hasła

Najlepszymi hasłami są przypadkowe, łatwe mnemonicznie zlepki liter cyfr i znaków przestankowych, albo pierwsze litery słów jakiegoś fragmentu utworu literackiego. Na przyklad z

```
So long, and thanks for all the fish (D.Adams)
```

Możemy wygenerować hasło `Sl,atfat`

> Należy unikać zapisywania hasła, zwłaszcza razem z nazwą komputera.

2.4 Shell — interpreter poleceń

Popularne są dwie rodziny shelli: wywodzące się od starounixowego Bourne shella (`sh`) oraz od powstałego w Berkeley C-shella (`csh`).

Do pierwszej rodziny należy standardowy na Linuxie `bash` (Bourne Again Shell), do drugiej szeroko rozpowszechniony Tenex shell — `tcsh`.

Aby przekonać się w jakim shellu pracujemy, należy wykonać polecenie `echo $SHELL`

```
[11:19:00] ben@kawa:~> echo $SHELL
/bin/bash
```

Tu jak widać działa `bash`...

```
herbata:/home/zls/benke>echo $SHELL
/bin/tcsh
```

...a tu `tcsh`.

2.5 Polecenia

Większość poleceń wydawanych systemowi ma postać

```
program argumenty...
```

Na przykład polecenie

```
zodiac1:~$ cat /proc/version
Linux version 1.3.56 (root@zodiac1) (gcc version 2.7.2) #2 Wed Jan 10 19:45:43 MET 1996
```

powoduje wypisanie zawartości pliku `/proc/version` na terminal. Ogólniej,

```
cat plik-1 plik-2 ... plik-n
```

wypisuje kolejno zawartość podanych plików na terminal.

2.6 Strumienie

Tak naprawdę ten mechanizm jest trochę bardziej skomplikowany. Każdy program w Unixie operuje na (co najmniej) trzech strumieniach znaków:

- wejściowym (standard input)
- wyjściowym (standard output)
- komunikatów o błędach (standard error)

Program `cat` wypisuje zawartość podanych plików na wyjście, a jeśli został wywołany bez argumentów, to czerpie dane z wejscia; jeśli wystąpią błędy, to informacja o tym zostanie wypisana na strumień błędów.

2.7 Kierowanie strumieniami

Standardowo wszystkie trzy strumienie są przyłączone do terminala. Shell pozwala jednak na przełączenie ich do innych plików:

- fraza `< f` w dowolnym miejscu polecenia przyłączy strumień wejściowy do pliku `f`
- fraza `> f` uczyni to samo ze strumieniem wyjściowym.

Zagadka: po co wyróżniono strumień błędów?

N.p. `cat plik-1 > plik-2` przepisze zawartośc jednego pliku do drugiego, zaś `cat > f` — zapisze w pliku `f` to, co wpiszemy z klawiatury.

Zagadka: jaki będzie efekt `cat < f`?

2.8 Filtry

Istnieje możliwość połączenia strumienia wyjściowego jednego programu ze strumieniem wejściowym innego przez rurę (*pipe)*, np

```
zodiac1:~$ cat /etc/passwd | grep -i ewa | cut -f 1,5 -d :
ewka:Ewa Kazana
ewojcik:Ewa Wojcik
eostasz:Ewa Ostasz
```

Wiele programów w Unixie jest pomyślanych jako filtry. Klasycznym przykładem jest tu właśnie `grep` — przepuszcza tylko linie zawierające podany wzorzec.

Rozdział 3

System plików

3.1 Zawartość katalogu - polecenie ls

```
zodiac1:~$ ls /
bin/        boot/       bootdsk/    cdrom/      dbin/
dev/        etc/        home/       lib/        linux/
mnt/        mnt2/       mud/        proc/       root/
shlib/      tmp/        usr/        var/        vmlinuz
```

Więcej szczegółów zapewni nam użycie opcji `-al`

```
zodiac1:~$ ls -al
total 7
drwxr-xr-x   2 benke    teachers     1024 Mar 15 11:17 ./
drwxr-xr-x  19 root     teachers     1024 Mar  4 13:01 ../
-rw-r--r--   1 benke    staff          49 Mar  8 16:13 dd
-rw-r--r--   1 benke    staff           5 Mar  8 16:13 ddd
-rw-r--r--   1 benke    staff           7 Mar  8 16:13 f
```

3.2 Inne użyteczne opcje polecenia ls

`-t` — sortuj według daty plików

`-r` — odwróć kolejnosc wypisywania plików

`-s` — podaj rozmiar pliku w blokach (w Linuxie blok=1KB)

`-R` — listuj rekurencyjnie wszystkie napotkane katalogi (**Uwaga:** często oznacza to bardzo długą listę plików)

`-S` — sortuj według rozmiaru plików (tylko Linux)

`-X` — sortuj według rozszerzenia (tylko Linux)

`--help` — pomoc (tylko Linux)

3.3 Gdy wydruk nie mieści się na jednym ekranie...

...może pomóc program `more`. (W Linuxie występuje rozszerzona wersja tego programu pod nazwą `less`). Jest to filtr działający prawie tak samo jak `cat`, z tym, że po przepuszczeniu jednego ekranu tekstu zatrzymuje się i czeka na polecenie użytkownika. Teraz możemy nacisnąć:

`Enter` — `more` przepuści jeszcze jedną linię,

`spacja` — przepuści jeszcze jeden ekran,

`b` — **cofnie** o jeden ekran (tylko `less`),

`/`*napis* — przejdzie do najbliższej linii zawierającej *napis*,

`q` — zakończy wyświetlanie.

3.4 Jak czytać wyniki `ls -l`

```
drwxr-xr-x  51 ben  zls 5120 Mar 15 13:30 .
lrwxrwxrwx   1 ben  zls   10 May 31  1995 Install -> ../install
```

Pierwsza kolumna: 1 znak określający typ pliku i 3*3 znaki określające prawa dostępu do pliku

`d` — katalog

`l` — link symboliczny

`r` — prawo do czytania

`w` — prawo do pisania

`x` — prawo do wykonywania

Dalej kolejno: ilość dowiązań, właściciel pliku, grupa, rozmiar, data i ew. czas, nazwa.

3.5 Prawo wykonywania dla katalogu

```
zodiac1:~$ ls -ld trash
drwxr-xr-x   2 benke    staff  1024 Mar 22 12:09 trash/
zodiac1:~$ ls -l trash
total 5
-rw-r--r--   1 benke    staff     6 Mar 22 12:09 1
-rw-r--r--   1 benke    staff     6 Mar 22 12:09 2
zodiac1:~$ chmod -x trash
```

```
zodiac1:~$ ls -ld trash
drw-r--r--   2 benke    staff  1024 Mar 22 12:09 trash/
zodiac1:~$ cat trash/1
cat: trash/1: Permission denied
```

3.6 Zmiana uprawnień — polecenie chmod

```
chmod [ -fR ] tryb plik...
chmod [ugoa ]{ + | - | = }[ rwxlsStTugo] plik...
```

Kombinacja liter ugoa określa których użytkowników dotyczyć będzie zmiana (**u**ser, **g**roup, **o**thers, **a**ll)

Operator + powoduje dodanie wybranych uprawnień, - ich usunięcie, zaś = powoduje ustawienie dokładnie podanych uprawnień (kasując pozostałe).

Przykłady:

```
chmod u+x f
chmod g+w f
chmod o-r f
```

Zmiany praw dostępu do pliku może dokonać tylko jego właściciel.

3.7 Jeszcze o uprawnieniach

```
[ben@trawa ben]$ date > plik
[ben@trawa ben]$ cat plik
Tue Oct  6 16:53:37 CEST 1998
[ben@trawa ben]$ ls -l plik
-rw-rw-r--   1 ben      ben            30 Oct  6 16:53 plik
[ben@trawa ben]$ chmod ug-rw plik
[ben@trawa ben]$ ls -l plik
-------r--   1 ben      ben            30 Oct  6 16:53 plik
```

Przy takich ustawieniach właściciel pliku nie może go odczytać…

```
[ben@trawa ben]$ cat plik
cat: plik: Permission denied
```

…może go jednak skasować:

```
[ben@trawa ben]$ rm plik
rm: remove ‘plik’, overriding mode 0004? y
```

3.8 Bieżący katalog i zmiana katalogu

Do wyświetlania bieżącego katalogu służy polecenie `pwd`:

```
[ben@trawa Unix]$ pwd
/home/ben/Zajecia/Unix
```

Do zmiany bieżącego katalogu służy polecenie
`cd` [*katalog*] (bez argumentów przeniesie nas do "domu")

```
[ben@trawa Unix]$ pwd
/home/ben/Zajecia/Unix
[ben@trawa Unix]$ cd
[ben@trawa ben]$ pwd
/home/ben
[ben@trawa ben]$ cd /
[ben@trawa /]$ pwd
/
```

3.9 Polecenie `cd -` czyli tam i z powrotem

Polecenie
`cd -`
przeniesie nas do ostatnio odwiedzanego katalogu:

```
[ben@trawa Unix]$ pwd
/home/ben/Zajecia/Unix
[ben@trawa Unix]$ cd -
[ben@trawa BigCyc]$ pwd
/home/ben/MP3/BigCyc
[ben@trawa BigCyc]$ cd -
[ben@trawa Unix]$ pwd
/home/ben/Zajecia/Unix
```

3.10 Tworzenie katalogów

Polecenie `mkdir d` tworzy w bieżącym katalogu katalog o nazwie `d`.

Możemy też stworzyć katalog w innym, istniejącym już katalogu, np. `mkdir /tmp/d`.

Nie można stworzyć kilku poziomów na raz, np:

```
zodiac1:~$ mkdir nie/ma/mnie
mkdir: cannot make directory 'nie/ma/mnie': No such file or directory
```

3.11 Usuwanie katalogów

Pusty katalog `d` można usunąc przy pomocy polecenia `rmdir d`.

> DOS posiada identyczne polecenia `mkdir` i `rmdir` oraz skróty dla nich — `md` i `rd`. W Unixie tych skrótów nie ma.

3.12 Usuwanie plików

Polecenie `rm plik-1 ... plik-n` usuwa podane pliki. W tym poleceniu, podobnie jak we wszystkich innych, lista plików może być efektem rozwinięcia przez shell wzorca

Elementy wzorców:

- `*` pasuje do dowolnego napisu
- `?` pasuje do jednego, dowolnego znaku
- `[...]` pasuje do dowolnego znaku wymienionego w miejsce ...

Na przykład `?[A-Z]*` pasuje do wszystkich nazw, których drugi znak jest wielką literą.

`rm -r d` usuwa cale poddrzewo zaczynające sie od katalogu `d`

`rm -i pliki` pyta o zgode przed usunięciem każdego pliku

3.13 Kopiowanie plików - `cp`

Polecenie `cp` ma dwie postacie:
`cp [-iv]` *skąd dokąd*
tworzy kopię pliku *skąd* pod nazwą *dokąd*

`cp [-iv]` *pliki... dokąd*
umieszcza kopie wymienionych plików w katalogu *dokąd*

3.14 Tworzenie nowego dowiązania do pliku — ln

`ln` *s d*
tworzy nowe dowiązanie do pliku *s* pod nazwą *d*. Odtąd plik ten jest dostępny pod obydwoma nazwami. Usunięcie *s* lub *d* powoduje zniknięcie tylko dowiązania — plik dalej istnieje i jest dostępny pod drugą z nazw:

```
[ben@trawa Unix]$ ls -il plik
  24638 -rw-rw-r--   1 ben      ben             0 Oct  7 19:35 plik
[ben@trawa Unix]$ cp plik kopia
[ben@trawa Unix]$ ls -il plik kopia
  24639 -rw-rw-r--   1 ben      ben             0 Oct  7 19:36 kopia
  24638 -rw-rw-r--   1 ben      ben             0 Oct  7 19:35 plik
[ben@trawa Unix]$ ln plik linka
[ben@trawa Unix]$ ls -il plik linka
  24638 -rw-rw-r--   2 ben      ben             0 Oct  7 19:35 linka
  24638 -rw-rw-r--   2 ben      ben             0 Oct  7 19:35 plik
[ben@trawa Unix]$ rm plik
[ben@trawa Unix]$ ls -l linka
-rw-rw-r--   1 ben      ben             0 Oct  7 19:35 linka
```

3.15 Przesuwanie i zmiana nazwy plików — `mv`

`mv [-iv]` *skąd dokąd*
Zmienia nazwę pliku *skąd* na *dokąd*

`mv [-iv]` *pliki... dokąd*
przesuwa wymienione pliki do katalogu *dokąd*

Jeśli przesunięcie odbywa się w obrębie jednego systemu plików (partycji,urządzenia), koszt wykonania `mv` nie zależy od rozmiaru pliku. W takim wypadku `mv` *s d* jest równoważne
`ln` *s d* `;` `rm` *s*

3.16 Modyfikacje linii poleceń

Często zdarza się, że pomylimy się wpisując polecenie, albo też chcemy wykonać polecenie podobne do jednego z uprzednio wykonanych. Zamiast wpisywać wszystko od nowa możemy przywołać właściwe polecenie i wykonać je po dokonaniu zmian.

Do poruszania się po liście wykonanych poleceń służą strzałki $\uparrow$, $\downarrow$, lub kombinacje `Ctrl-p` (poprzednia) i `Ctrl-n` (następna)

Do zmian można używać kursorów lub klawiszy

`Ctrl-a` — początek linii, `Ctrl-e` — koniec linii

`Ctrl-b` — znak w tył, `Ctrl-f` — znak w przód

`Esc-b` — słowo w tył, `Esc-f` — słowo w przód

`Ctrl-d` — usuń znak, `Esc-d` — usuń słowo

3.17 Edytory

`vi` — jest na każdym Unixie, ale skrajnie niewygodny. Dla fanatyków: `man vi`.

`joe` — Standardowy w Lniuxie, poza tym rzadko spotykany. Mały, wygodny, klawiszologia oparta na Wordstarze. Po wywołaniu `Ctrl-k h` wyświetli pomoc.

`emacs` — więcej niż edytor, uniwersalne narzędzie do prawie wszystkich czynności w Unixie. Obiekt religijnego niemal uwielbienia bądź niechęci tysięcy użytkowników Unixa. Po wywołaniu polecam `Ctrl-h t` — tutorial, oraz `Ctrl-h i` — `info`, mnóstwo informacji nie tylko o emacsie.

Rozdział 4

Procesy

4.1 Pojeęcie procesu

- proces jest *abstrakcją* działającego programu;
- żaden proces nie może zakłócić dzialania innego procesu, ani jądra systemu;
- komunikacja miedzy procesami może zachodzić tylko za obopólną zgodą;
- proces może powołać do życia dziecko — proces potomny, który dziedziczy kod, dane i otwarte pliki rodzica;
- rodzic nie może wpływać na działanie swojego dziecka, ale może czekać (lub nie) na jego zakończenie;

Pytanie dla hackerów: czym się robi dzieci (w UNIX-ie)?

4.2 Lista procesów - polecenie ps

```
zodiac1:~$ ps -u
USER       PID %CPU %MEM SIZE  RSS TTY STAT START   TIME COMMAND
benke    11053  0.1  2.0 1136  644 pp0 S    10:55   0:00 -bash
benke    11262  0.0  0.8  800  280 pp0 R    11:01   0:00 ps -u
```

USER — właściciel procesu
PID — identyfikator (numer) procesu
%CPU,%MEM — wykorzystanie procesora i pamięci
SIZE — całkowity rozmiar procesu

RSS — faktyczny rozmiar w pamięci
TTY — terminal związany z procesem
START — kiedy proces został uruchomiony
TIME — ile czasu procesora zużył
COMMAND — jak został uruchoniony (argv)

4.3 Kolejny przykład listy procesów

```
USER       PID  TTY STAT START   TIME COMMAND
root         1   ?  S    19:14   1:09 init [5]
root         2   ?  SW   19:14   0:00 (kernel bdflush)
root         8   ?  S    19:14   0:01 update (bdflush)
root        43   ?  S    19:20   0:02 /usr/sbin/crond -l8
root        60   ?  S    19:20   1:56 /usr/sbin/syslogd
root      1407  v01 S    20:05   0:00 (agetty)
sosnowsk  6294   ?  S    08:54   0:00 (tcsh)
rmilczew  8735  pp6 S    10:01   0:00 -sh
mgruszcz  9906   ?  S    10:23   0:11 netscape
pkozlows 10443  pp2 S    10:38   0:00 -bash
mkorsako 10446  pp4 S    10:38   0:00 -bash
mkorsako 10500  pp4 S    10:38   0:02 telnet bull
```

4.4 Procesy w tle

jeśli polecenie zakończymy znakiem `&`, to shell nie będzie czekał na jego zakończenie, lecz uruchomi go "w tle":

```
zodiac1:~$ find / -name .Xdefaults -print 2>/dev/null &
[1] 12548
/home/sml/dabrowa/.Xdefaults
zodiac1:~$ ps
/home/sml/jakacki/.Xdefaults
  PID TTY STAT  TIME COMMAND
11053 pp0 S     0:00 -bash
12548 pp0 S     0:11 find / -name .Xdefaults -print
12625 pp0 R     0:00 ps
zodiac1:~$ /home/sml/kusmirek/.Xdefaults
```

4.5 Procesy w tle c.d.

- Nie należy uruchamiać w tle procesów interakcyjnych, takich jak edytory (wyjątek: procesy mające swoje własne okno);

- Przy uruchamianiu w tle procesu który cos wypisuje, najlepiej skierowac jego wyjście do pliku;
- Procesy które działają w tle, a których celem jest trwanie, nie zaś zakończenie nazywa się **demonami**
- W UNIX-ie demony świadczą rozmaite usługi systemowe czy sieciowe.

Rozdział 5

Programowanie w shellu

5.1 Skrypty (scenariusze dla interpretera poleceń)

Wykonywane czynności można automatyzować przy pomocy tzw. skryptów. Skrypt to po prostu plik tekstowy zawierający kolejne polecenia dla interpretera.

Przykład:

```
[ben@trawa Unix]$ cat script1
echo $SHELL
ls -l script1

[ben@trawa Unix]$ ./script1
/bin/bash
-rwxrwxr-x   1 ben      ben            27 Oct  9 16:24 script1
```

5.2 Interpreter poleceń — struktura leksykalna

Nazwa — ciąg liter i cyfr zaczynający się od litery

Fraza — ciąg słów oddzielonych blankami; symbole `;` `&` `&&` `||` mają znaczenie specjalne.

Potok — ciąg fraz połączonych symbolami `|`

Lista — ciąg potoków połączonych symbolami `;` `&` `&&` `||`

Proste polecenie to fraza, w której pierwsze słowo jest interpretowane jako nazwa programu do wykonania.

Lista ujęta w nawiasy jest frazą.

5.3 Wynik polecenia — kod stanu

Wykonanie każdej frazy dostarcza *wyniku* (liczbowego *kodu stanu*). Powinien on wynosić 0 jeśli wykonanie się powiodło.

Wynikiem potoku jest wynik ostatniej jego frazy. Jeśli potok jest poprzedzony znakiem `!`, wynik zostanie zanegowany.

Sposób wykonania listy zależy od symboli łączących jej elementy:

- `&` — poprzedni potok jest wykonywany asynchronicznie;
- `;` — wykonanie następnego potoku zostanie rozpoczęte po zakończeniu wykonania poprzedniego;
- `&&` — następny potok zostanie wykonany tylko gdy wynikiem poprzedniego było 0
- `||` — następny potok zostanie wykonany tylko gdy wynik poprzedniego był różny od 0

5.4 Środowisko

Środowisko wiąże nazwy (zmienne) z ich wartościami. Wartość zmiennej możemy uzyskać przez `$`*nazwa*. Pewne zmienne są związane w *środowisku globalnym*, np.

```
zodiac1:~$ echo $MAIL
/var/spool/mail/benke
```

Innym zmiennym możemy nadawać wartość wewnątrz skryptu, np.

```
val=3
echo $val
```

Przez `$`n możemy uzyskać wartość n-tego parametru skryptu (zakładając, że został wywołany z parametrami).

5.5 Konstrukcja `for`

Składnia:

`for` *nazwa* `in` *słowo*`...; do` *lista* `; done`

Wykonuje *listę* dla każdego wymienionego *słowa*; przy każdym wykonaniu listy, wartością związaną z nazwą jest bieżące słowo.

Lista słów zawarta między `in` a `do` może być wynikiem rozwinięcia wzorca.

5.6 Przykład

```
[ben@trawa Unix]$ for i in *.dvi ; do echo $i ; done
TSU.dvi
fs.dvi
oview.dvi
proc.dvi
sh.dvi
shell.dvi
```

5.7 Konstrukcja `select`

Składnia:

`select` *nazwa* `in` *słowo*`...; do` *lista* `; done`

Wzorce występujące wśród *słów* są rozwijane. Wynikowa lista jest drukowana na stderr, każdy element poprzedony numerem. Następnie wyświetlany jest prompt i wczytywana jest jedna linia z wejścia. Jeśli jej zawartość stanowi jeden z wyświetlonych numerów, zmienna *nazwa* przyjmuje wartość elementu oznaczonego tym numerem.

Przykład

```
[ben@trawa Unix]$ select i in *.dvi ; do echo $i ;
break ; done
1) TSU.dvi          3) oview.dvi          5) sh.dvi
2) fs.dvi           4) proc.dvi           6) shell.dvi
#? 5
sh.dvi
```

5.8 Konstrukcja `if`

Składnia:

`if` *lista* `then` *lista1* `else` *lista2* `fi`

Jeśli wynik listy następującej po `if` był pomyślny (0), to wykonuje *listę1* w przeciwnym wypadku *listę2*.

Przykład:

```
zodiac1:~$ if (true) then echo tak ; else echo nie ; fi
tak
zodiac1:~$ if (false) then echo tak ; else echo nie ; fi
nie
```

5.9 Składniki przykładu

`true` — zawsze sygnalizuje sukces nic nie robiąc

`false` — zawsze sygnalizuje niepowodzenie nic nie robiąc

`echo` *s* — wypisuje napis *s*; znaki specjalne (np. *, ?) są interpretowane.

Jaki będzie efekt wykonania polecenia `echo *` ?

5.10 Polecenie `test`

Składnia:

`test` *wyrażenie*
Daje wynik 0 (prawda) jeśli wyrażenie jest prawdziwe.

Przykład:

```
zodiac1:~$ if (test  `expr 2 + 2`  -eq 4)
> then echo cztery ; else echo \? ; fi
cztery
```

5.11 Użyteczne testy

- `-e` *plik* — *plik* istnieje
- `-d` *plik* — *plik* istnieje i jest katalogiem
- `-d` *plik* — *plik* istnieje i jest zwykłym plikiem
- `-r` *plik* — *plik* istnieje i mamy prawo jego czytania
- `-t` *n* — strumień o numerze *n* jest związany z terminalem

Przykład:

```
zodiac1:~$  if (test -t 1) then (echo `tty`) else (echo \?) fi
/dev/ttyp0
[ben@trawa Unix]$ if (test -t 1) then (tty)
> else (echo \?)  fi > wynik
[ben@trawa Unix]$ cat wynik
?
```

Oczywiście polecam `man test`

5.12 Konstrukcja `while`

Składnia:

`while lista` *lista* `do` *lista1* `done`

Tak długo jak wynik listy następującej po `if` był pomyślny (0), wykonuje *listę1*.

Przykład:

```
zodiac1:~$  while (true) do
>  echo "Nacisnij Ctrl-C by przerwac" ; sleep 5
> done
Nacisnij Ctrl-C by przerwac
Nacisnij Ctrl-C by przerwac
Nacisnij Ctrl-C by przerwac
```

5.13 Parametry skryptu

`$`n — n-ty parametr skryptu

`$#` — liczba parametrów skryptu

`$*` — wszystkie parametry

Przykład:

```
#!/bin/sh
echo $0             # wypisz nazwe skryptu...
echo $#             # ...wypisz liczbe parametrow...
for i in $* ; do
  echo $i           # ...wypisz kolejne parametry.
done
```

5.14 Problem praktyczny

Mamy w katalogu pewną (być może dużą) liczbę plików z rozszerzeniem `.dvi`. Chcemy przezwać je tak by każdy miał rozszerzenie np `.dvd`

Spróbujmy:

```
[ben@trawa Unix]$ mv *.dvi *.dvd
mv: when moving multiple files, last argument must be a directory
```

Tak się oczywiście nie da...

```
[ben@trawa Unix]$ for i in *.dvi ; do mv $i $i.dvd ; done
[ben@trawa Unix]$ ls *.dvd
TSU.dvi.dvd    oview.dvi.dvd  sh.dvi.dvd
fs.dvi.dvd     proc.dvi.dvd   shell.dvi.dvd
```

ops!

5.15 Polecenie `basename`

`basename` *ścieżka rozszerzenie*
odcina z argumentu ścieżka wiodące katalogi i podany sufiks, np.

```
[ben@trawa Unix]$ basename TSU.dvi.dvd .dvd
TSU.dvi
[ben@trawa Unix]$ for i in *.dvi.dvd ; do
mv -v $i `basename $i .dvd` ; done
TSU.dvi.dvd -> TSU.dvi
fs.dvi.dvd -> fs.dvi
oview.dvi.dvd -> oview.dvi
proc.dvi.dvd -> proc.dvi
sh.dvi.dvd -> sh.dvi
shell.dvi.dvd -> shell.dvi
```

5.16 Rozwiązanie problemu

```
[ben@trawa Unix]$ for i in *.dvi
do mv -v $i `basename $i .dvi`.dvd ; done
TSU.dvi -> TSU.dvd
fs.dvi -> fs.dvd
oview.dvi -> oview.dvd
proc.dvi -> proc.dvd
sh.dvi -> sh.dvd
shell.dvi -> shell.dvd
```

5.17 Narzędzia do cięcia plików na linie...

Wszystkie wymienione poniżej programy działają też jako filtry

`grep` *wzorzec pliki* — wyszukuje linie zawierające *wzorzec*

`head -`*n plik* — dostarcza *n* pierwszych linii pliku

`tail -`*n plik* — dostarcza *n* ostatnich linii pliku `tail +`*n plik* — dostarcza wszystkie oprócz n pierwszych linii

> **Zagadka:** Jak uzyskać dokładnie n-tą linię pliku

5.18 ...i na kolumny

`cut -c` *m-n plik*
Wycina z pliku (bądź strumienia) kolumny od m-tej do$n$-tej. **Przykład:**

```
$ ps
  PID TTY STAT  TIME COMMAND
 3178 pp7 S     0:00 -bash
 5573 pp7 R     0:00 ps
$ ps | tail +2 | cut -c 1-6 | tr "\012" " " ; echo
 3178   5580   5581   5582   5583
```

> Polecenie `tr` na końcu potoku zamienia znaki końca linii na spacje. To użyteczny trick.

> **Zagadka dla hackerów:** Po co `echo` na końcu?

5.19 Zastosowanie

```
$ kill `ps | tail +2 | cut -c 1-6 | tr "\012" " "`
```

5.20 Wyszukiwanie plików

`find` *katalog wyrażenie*

Obchodzi drzewo katalogów poczynając od wskazanego i wyszukuje wszystkie plik spełniające *wyrażenie*

Przykłady:

`find / -name core` — wyszuka pliki o nazwie `core` w całym systemie

`find . -mtime -7` — wyszuka pliki nowsze niż tydzień

`find /home -atime +300` — wyszuka pliki nieużywane od conajmniej 300 dni w katalogach użytkowników

Bardzo polecam `man find`.

Rozdział 6

Podstawy C

6.1 Skrypt, którym robiłem te slajdy

`envelope`

```
#!/bin/sh
cat header.tex
for i in $* ; do
        echo '\\vbox{\\texttt' {$i}\\hrule'

        echo '\\begin{verbatim}'
        cat $i
        echo '\\end{verbatim}'

        echo '\\hrule}\par\medskip'
        echo
done
cat trailer.tex
```

6.2 Skrypt do testowania programów

`status`

```
#!/bin/sh

prog=$1  # Zapamietaj nazwe programu
shift      # Przenumeruj wszystkie argumenty o 1 w dol
$prog $*   # Wykonaj program z podanymi argumentami
echo $?    # Podaj kod stanu dostarczony przez ten program
```

```
zodiac1:~/C$ status grep ben /etc/passwd
benny:dr7uwN/44vchY:618:600:Grzegorz Grabowski:/home/sml/benny:/bin/bash
benke:ZPuLQe98b6DFM:2041:100:Marcin Benke:/home/staff/teachers/benke:/bin/bash
0
zodiac1:~/C$ status grep nie_ma_mnie /etc/passwd
1
```

6.3 Najprostszy użyteczny program w C

true.c

```
int main ()
{
  return 0 ; /* Dostarcz kod stanu 0 */
}
```

Najprosciej skompilować nasz program używając `make`

```
zodiac1:~/C$ make true
gcc -Wall    true.c   -o true
zodiac1:~/C$ ls -l true
-rwxr-xr-x   1 benke    staff        3776 Apr 18 19:32 true*
zodiac1:~/C$ status true
0
```

6.4 Sprawdzmy, że to nie oszustwo...

seven1.c

```
int main ()
{
  return 7 ;   /* Dostarcz kod stanu 7 */
}
```

Inna metoda kompilacji:

```
zodiac1:~/C$ gcc -o seven1 seven1.c
zodiac1:~/C$ status seven1
7
```

6.5 Zmienne lokalne

seven2.c

```
int main ()
{
  int result ;      /* result jest zmienna lokalna typu int */
  result = 7 ;      /* Nadanie zmiennej result wartosci 7   */
  return result ;  /* Dostarcz wartosc zmiennej result (7) */
}
```

6.6 Inicjalizacja zmiennych

seven3.c

```
int main ()
{
  int result = 7 ;
  return result ;
}
```

6.7 Funkcje

seven4.c

```
int result()
{
  return 7 ;
}

int main ()
{
   return result() ;
}
```

6.8 Pisanie tekstu do strumienia — fprintf

hello1.c

```
#include <stdio.h>
int main()
{
  fprintf( stdout, "Hello, stdout!\n" ) ;
  fprintf( stderr, "Hello, stderr!\n" ) ;
}
```

```
$ hello1
Hello, stdout!
Hello, stderr!
$ hello1 > /dev/null
Hello, stderr!
```

6.9 Pożyteczny skrót — printf

hello2.c

```
#include <stdio.h>

int main()
{
  printf( "Hello, brave new world."
          " The moon is a harsh mistress!\n" ) ;
}
```

```
$ hello2
Hello, brave new world. The moon is a harsh mistress!
```

6.10 Zmienne napisowe; zmienne globalne

hello3.c

```
#include <stdio.h>

char * message = "Hello, brave new world!\n" ;

int main()
{
  printf( message ) ;
}
```

```
$ status hello3
Hello, brave new world!
1
```

6.11 Tablice; pętla `for`

hello4.c

```
#include <stdio.h>
char * message[4] = { "Hello\n", "brave\n", "new\n", "world!\n" } ;
int main()
{
  int i ;
  for ( i = 0 ; i < 4 ; i = i+1 )
    {
      printf( message[i] ) ;
    }
}
```

6.12 Skróty w C

Zamiast `i = i + 1` możemy napisać `i += 1`, a nawet `i++`. Podobnie zamiast `i = i - 1` moglibyśmy napisać `i -= 1` albo `i--`.

Ponadto jeśi ciało pętli składa się z jednej instrukcji, możemy pominąć nawiasy klamrowe:

```
for ( i = 0 ; i < 4 ; i = i+1 ) printf( message[i] ) ;
```

Należy jednak bardzo uważąć, aby nie napisać tak:

```
for ( i = 0 ; i < 4 ; i = i+1 ) ;
  printf( message[i] ) ;
```

To (czyli średnik zaraz po nawiasie zamykającym) spowoduje, że w pętli wykona się instrukcja pusta, a instrukcja `printf` wykona się raz, po zakończeniu „pustej” pętli.

6.13 `hello4` jeszcze raz:

hello4.c

```
#include <stdio.h>
char * message[4] = { "Hello\n", "brave\n", "new\n", "world!\n" } ;
int main()
{
  int i ;
  for ( i = 0 ; i < 4 ; i++ ) printf( message[i] ) ;
}
```

```
$ hello4
Hello
brave
new
world!
```

6.14 Argumenty funkcji

hello5.c

```
#include <stdio.h>
char * message[4] = { "Hello", "brave", "new", "world!" } ;
void writeln(char * s)
{
  printf( s ) ; printf( "\n" ) ;
}
int main()
{
  int i ;
  for ( i = 0 ; i < 4 ; i = i++ ) writeln( message[i] ) ;
}
```

6.15 Struktura napisów

Znaki w C (typ `char`) są do pewnego stopnia utozsamiane z liczbami (`int`), z tym że `char` zajmuje zawsze jeden bajt.

Napisy w C to tablice znaków. Koniec napisu jest sygnalizowany przez znak o kodzie 0.

Napis `"Hello"` jest więc reprezentowany tak:

H	e	l	l	o	\0

a dokładniej:

72	101	108	108	111	0

6.16 Ulepszamy `writeln` — funkcja `puts`

hello6.c

```
extern int putchar(int c) ;
int puts(char * s)
{
  int i;
  for( i = 0 ; s[i] != 0 ; i++ ) putchar(s[i]) ;
  putchar('\n') ;
  return i ;
}
int main()
{
  return puts("Hello, is it me you're looking for?") ;
}
```

6.17 Ulepszamy `puts` — wskaźniki i tablice

hello7.c (fragment)

```
int puts(char * s)
{
  char * t;

  for( t = s ; *t != 0 ; t++ ) putchar(*t) ;
  putchar('\n') ;
  return t - s ;
}
```

`*t` oznacza znak wskazywany przez `t`.
`t++` oznacza przesunięcie wskaźnika o 1 znak do przodu
`t-s` oznacza „odległość” między wskaźnikami `t` i `s`.

Rozdział 7

Używamy C

7.1 Argumenty funkcji `main`: `argc` i `argv`

`args1.c`

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  int i, count = argc - 1 ;
  printf("%d\n", count) ;

  for ( i = 1 ; i <= count ; i++  )
     printf( argv[i] ) ; printf( "\n" ) ;
  return count ;
}
```

7.2 Długość napisu — wskaźniki i pętla `for`

strlen1.c

```
int strlen1(char * s)
{
  char *t ;

  for ( t = s ; *t ; t++ )
    /* empty */ ;

  return (t - s) ;
}
```

7.3 Długość napisu — indeksy i pętla while

strlen2.c

```
int strlen1(char * s)
{
  int i = 0 ;

  while( s[i] != '\0' )
    i ++ ;
  return i ;
}
```

7.4 Przykład programu głównego

lentest.c

```
#include <stdio.h>

extern int strlen1(char * s) ;

int main(int argc, char * argv[])
{
  if ( argc < 2 ) {
    fprintf( stderr, "Usage: %s <string>\n", argv[0] ) ;
    return 1 ;
  }
  printf( "%d\n",  strlen1(argv[1])) ;
  return 0 ;
}
```

7.5 Kompilacja programu w kilku plikach

Razem:

```
$ gcc -o lentest lentest.c strlen1.c
$ ls -l lentest
-rwxr-xr-x   1 ben      zls          5608 Apr 26 10:56 lentest
```

Oddzielnie:

```
$ gcc -c lentest.c
$ gcc -c strlen2.c
$ ls -l *.o
-rw-r--r--   1 ben      zls           980 Apr 26 10:57 lentest.o
-rw-r--r--   1 ben      zls           596 Apr 26 10:57 strlen2.o
$ gcc -o lentest2 strlen2.o lentest.o
$ ls -l lentest2
-rwxr-xr-x   1 ben      zls          5600 Apr 26 10:57 lentest2
```

7.6 Makefile

Makefile

```
CC=gcc
CFLAGS=-Wall

lentest: lentest.o strlen1.o

lentest2: lentest.o strlen2.o
	$(CC) -o lentest2 lentest.o strlen2.o
```

Powyższy plik `Makefile` składa się z definicji zmiennych (`CC, CFLAGS`) oraz reguł.

Reguły mają składnie następującą:

⟨cel⟩ : ⟨zależności⟩
⟨TAB⟩ ⟨polecenia⟩

7.7 Jak to działa?

```
$ make lentest
gcc -Wall   -c lentest.c -o lentest.o
gcc -Wall   -c strlen1.c -o strlen1.o
gcc   lentest.o strlen1.o   -o lentest
$ make lentest2
gcc -Wall   -c strlen2.c -o strlen2.o
gcc -o lentest2 lentest.o strlen2.o
$ make lentest2
make: 'lentest2' is up to date.
```

> `make` kompiluje tylko te pliki które są nieaktualne, tj. starsze od plików od których zależą.

Korzysta w tym celu z wyspecyfikowanych w pliku , oraz z tzw. reguł domyślnych. Np. aby stworzyć plik `.o` z pliku `.c` "domyśla się", by skorzystać z reguły
`$(CC) -c $(CPPFLAGS) $(CFLAGS)`

7.8 Bardziej złożony przykład

Makefile

```
CC=gcc
CFLAGS=-Wall

PROGS = args1 codes count date date1 date2 date3 date4 \
        hello1 hello2 hello3 hello4 hello5 hello6 hello7 hello8 \
        lentest lentest2 lower no1 no2 seven1 seven2 seven3 \
        true wc who yes

all: $(PROGS)
lentest: lentest.o strlen1.o
lentest2: lentest.o strlen2.o
	$(CC) -o lentest2 lentest.o strlen2.o

clean:
	-rm -f core *.o *~ $(PROGS)

PACKAGE=examples-97
ARCHIVE=$(PACKAGE).tar.gz
CFILES := *.c
DISTFILES := Makefile $(CFILES)

dist: $(ARCHIVE)
$(ARCHIVE): $(DISTFILES)
	-rm -rf $(PACKAGE)
	mkdir $(PACKAGE)
	cp $(DISTFILES) $(PACKAGE)
	tar zcf $(ARCHIVE) $(PACKAGE)
	-rm -rf $(PACKAGE)

$(ARCHIVE): $(DISTFILES)
	tar zcf $(PACKAGE).tar.gz $(DISTFILES)
```

7.9 Wyświetlamy datę — przekazywanie wskaźników

date1.c

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

int main()
{
  long clock ;

  time( & clock) ;
  printf( "Date: %ld\n", clock ) ;
  return 0 ;
}
```

date2.c

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

int main()
{
  long clock ;

  time( & clock) ;
  printf( "Date: %s\n", ctime(&clock) ) ;
  return 0 ;
}
```

7.10 Struktury — przykład

```
struct  tm {    /* see ctime(3) */
        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon;
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
```

```
};
struct tm time1 ;
typedef struct tm Time ;
```

date3.c

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

int main()
{
  long clock ;
  struct tm date;

  time( & clock) ;
  date = localtime_r( & clock, & date ) ;

  printf( "Time: %02d:%02d:%02d\n",
          date.tm_hour, date.tm_min, date.tm_sec ) ;
  return 0 ;
}
```

date4.c

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

int main()
{
  long clock ;
  struct tm * date;

  time( & clock) ;
  date = localtime( & clock ) ;

  printf( "Time: %02d:%02d:%02d\n",
          date->tm_hour, date->tm_min, date->tm_sec
        ) ;
  return 0 ;
}
```

7.11 Instrukcja `break`

```
int strcmp(char *s, char *t)
{
  int i, result = 0 ;
  for ( i=0; s[i] || t[i] ; i++) {
    if( s[i] < t[i] ) {
      result = -1 ;  break ;
    }
    if( s[i] > t[i] ) {
      result = 1 ;  break ;
    }
  }
  return result ;
}
```

7.12 Instrukcja `switch`

```
int main(int argc, char *argv[])
{
  int i, err, optg, optl, opth ;
  for ( i = 1 ; i < argc ; i++ )
    if( argv[i][0] == '-' )
      switch( argv[i][1] )
      {
        case 'g' : optg = 1 ; break ;
        case 'l' : optl = 1 ; break ;
        case '\0': break ;
        case 'h' : /* no break! */
        default : usage () ;
      }
/* ... */
```

Rozdział 8

Biblioteka wejścia/wyjścia: stdio

8.1 Program no1

```
#include <stdio.h>

void main()
{
  for(;;)
    { fputc('n', stdout) ; fputc('\n', stdout) ; }
}
```

8.2 Program no2

```
#include <stdio.h>

void main()
{
  for(;;)
    { puts("n") ; }
}
```

8.3 Funkcje `stdio`

Funkcja	Wynik	!	Przeznaczenie
`fopen(`s, t`)`	`FILE*`	`NULL`	Otwiera plik o nazwie s w trybie t.
`fclose(`f`)`	int	`EOF`	Zamyka plik f.
`fgetc(`f`)`	int	`EOF`	Czyta znak z pliku f.
`fputc(`c, f`)`	c	`EOF`	Pisze znak c na plik f.
`fputs(`s, f`)`	$\|s\|$	`EOF`	Pisze string s na plik f.
`puts(`s, f`)`	$\|s\|$	`EOF`	Ditto, dopisuje `\n`
`fgets(`b, n, f`)`	b	NULL	Czyta linię z pliku f do bufora b rozmiaru n.
`fread(`b, k, n, f`)`	int	$< n$	Czyta z pliku n rekordów rozmiaru k.
`fwrite(`b, k, n, f`)`	int	$< n$	Pisze to samo.
`fprintf(`$f, s, \ldots$`)`	int	< 0	Wypisuje na plik f argumenty $(\ldots)$ w formacie s.
`feof(`f`)`	int		Czy koniec f?

8.4 Funkcja `fopen`

`FILE *fopen( char *path, char *mode);`

Otwiera plik o nazwie `path`

Argument `mode` specyfikuje tryb otwarcia pliku:

`"r"` — czytanie; wskaźnik pozycji umieszczany jest na początku pliku

`"r+"` — czytanie i pisanie; wskaźnik pozycji jak wyżej.

`"w"` — pisanie; plik jest zamazywany jeśli istnieje, tworzony wpp.

`"w+"` — czytanie i pisanie, reszta jak wyżej

`"a"` — dopisywanie; wskaźnik pozycji umieszczany na końcu pliku

`"a+"` — czytanie i pisanie; wskaźnik pozycji jak wyżej.

Daje wynik `NULL` w razie błędu.

8.5 Funkcje `fread` i `fwrite`

`size_t fread(  void *ptr, size_t size, size_t n, FILE *stream);`

Czyta z pliku `stream` do `n` rekordów rozmiaru `size`, umieszczając je w buforze wskazywanym przez `ptr`. Daje w wyniku ilość odczytanych rekordów (a nie bajtów!)

`size_t fwrite( void *ptr, size_t size, size_t n, FILE *stream);`

Pisze do pliku `stream` `n` rekordów rozmiaru `size` umieszczonych w buforze wskazywanym przez `ptr`. Daje w wyniku ilość zapisanych rekordów

8.6 Program yes

```
#include <stdio.h>

void main(int argc, char *argv[])
{
  char * s ;

  switch(argc) {
  case 1 :
    s = "y" ; break ;
  case 2 :
    s = argv[1] ; break ;
  default :
    fprintf( stderr,
             "Usage: %s [text]\n", argv[0] ) ;
    exit(0) ;
  }
  for(;;)
    { puts(s) ; }
}
```

8.7 Program count

```
#include <stdio.h>
void main(int argc, char *argv[])
{
  int c, count=0;
  FILE *fp;
```

/* Sprawdź parametry wywołania programu. */

```
  if (argc != 2) {
    fprintf(stderr, "Usage: %s file\n", *argv);
    exit(1);
  }
```

/* Otwórz plik do czytania. Sygnalizuj błąd jeśli się nie udało (np. plik nie istnieje). */

```
  if ((fp = fopen(argv[1], "r")) == NULL) {
    perror(argv[1]);
    exit(1);
  }
```

```
    /* Czytaj i licz kolejne znaki aż do końca pliku. */

  while ((c = getc(fp)) != EOF) count ++;

/* Wypisz wynik. */

  printf("%d\n", count) ;

/* Zamknij plik. */

    fclose(fp);
    exit(0);
}
```

8.8 Program append-by-char

```
#include <stdio.h>

void main(int argc, char *argv[])
{
    int c;
    FILE *from, *to;

/* Sprawdź parametry wywołania programu. */

    if (argc != 3) {
      fprintf(stderr,
        "Usage: %s from-file to-file\n", *argv);
      exit(1);
    }

/* Otwórz plik from do czytania. Sygnalizuj błąd jeśli się nie udało (np.
plik nie istnieje). */

    if ((from = fopen(argv[1], "r")) == NULL) {
      perror(argv[1]);  exit(1);
    }

/* Otwórz plik to do dopisywania. Plik zostanie utworzony jeśli nie istnieje.
*/

    if ((to = fopen(argv[2], "a")) == NULL) {
        perror(argv[2]);
        exit(1);
    }

/* Czytaj kolejne znaki aż do końca pliku from i zapisuj na plik to. */
```

```
    while ((c = getc(from)) != EOF)
        putc(c, to);
```

/* Zamknij pliki. */

```
    fclose(from);
    fclose(to);
    exit(0);
}
```

8.9 Program append-by-line

```
#include <stdio.h>

void main(int argc, char *argv[])
{
    FILE *from, *to;
    char line[BUFSIZ];
```

/* Sprawdź parametry wywołania programu. */

```
    if (argc != 3) {
      fprintf(stderr,
        "Usage: %s from-file to-file\n", *argv);
      exit(1);
    }
```

/* Otwórz plik `from` do czytania. Sygnalizuj błąd jeśli się nie udało (np. plik nie istnieje). */

```
    if ((from = fopen(argv[1], "r")) == NULL) {
      perror(argv[1]);  exit(1);
    }
```

/* Otwórz plik `to` do dopisywania. Plik zostanie utworzony jeśli nie istnieje. */

```
    if ((to = fopen(argv[2], "a")) == NULL) {
        perror(argv[2]);
        exit(1);
    }
```

/* Czytaj kolejne linie aż do końca pliku `from` i zapisuj na plik `to`. */

```
    while (fgets(line, BUFSIZ, from) != NULL)
        fputs(line, to);
```

/* Zamknij pliki. */

```
    fclose(from);
    fclose(to);
    exit(0);
}
```

8.10 Program append-by-buffer

```
#include <stdio.h>

void main(int argc, char *argv[])
{
    FILE *from, *to;
    char line[BUFSIZ];
    int n;
```

/* Sprawdź parametry wywołania programu. */

```
    if (argc != 3) {
      fprintf(stderr,
        "Usage: %s from-file to-file\n", *argv);
      exit(1);
    }
```

/* Otwórz plik `from` do czytania. Sygnalizuj błąd jeśli się nie udało (np. plik nie istnieje). */

```
    if ((from = fopen(argv[1], "r")) == NULL) {
      perror(argv[1]);  exit(1);
    }
```

/* Otwórz plik `to` do dopisywania. Plik zostanie utworzony jeśli nie istnieje. */

```
    if ((to = fopen(argv[2], "a")) == NULL) {
        perror(argv[2]);
        exit(1);
    }
```

/* Czytaj kolejne porcje aż do końca pliku `from` i zapisuj na plik `to`. */

```
    while ((n = fread(buf, sizeof(char), BUFSIZ, from)) > 0)
         fwrite(buf, sizeof(char), n, to);
```

/* Zamknij pliki. */

```
    fclose(from);
    fclose(to);
    exit(0);
}
```

8.11 Program who

```
#include <stdio.h>
#include <utmp.h>
int main()
{
  FILE * fp ;
  struct utmp u ;
  if ((fp = fopen(UTMP_FILE, "r")) == NULL) {
    perror(UTMP_FILE) ; return 1 ;
  }
  while (fread( &u, sizeof(u), 1, fp)) {
    if(  (u.ut_type != USER_PROCESS)
       || !u.ut_name[0] )
      continue ;
    print_utmp_entry(& u) ;
  }
  fclose(fp) ;
  return 0 ;
}
```

8.12 Struktura utmp

```
struct utmp {
        char ut_user[8];
        char ut_id[4];
        char ut_line[12];
        short ut_pid;
        short ut_type;
        struct exit_status ut_exit;
        time_t ut_time;
};
```

8.13 Funkcja print_utmp_entry

```
void print_utmp_entry(struct utmp * u)
{
```

```
  printf("%-8.8s", u->ut_user) ;
  printf(" %-8.8s", u->ut_line) ;
  printf(" %-12.12s\n",
         ctime(&(u->ut_time)) + 4 ) ;
}
```

Rozdział 9

Systemowe funkcje wejścia/wyjścia

9.1 Funkcje `creat` i `unlink`

creat1.c

```
#include <fcntl.h>

void main()
{
   creat("/tmp/temp", 0644) ;
}
```

unlink1.c

```
#include <unistd.h>
void main()
{
  unlink("/tmp/temp")  ;
}
```

```
~/Zajecia/Unix/C> ./creat1
~/Zajecia/Unix/C> ls -l /tmp/temp
-rw-r--r--   1 ben      zls            0 Nov  1 16:39 /tmp/temp
~/Zajecia/Unix/C> ./unlink
~/Zajecia/Unix/C> ls -l /tmp/temp
-rw-r--r--   1 ben      zls            0 Nov  1 16:39 /tmp/temp
~/Zajecia/Unix/C> ./unlink1
~/Zajecia/Unix/C> ls -l /tmp/temp
ls: /tmp/temp: No such file or directory
```

9.2 Typowy błąd przy użyciu `creat`

creat1-bad.c

```
#include <fcntl.h>

void main()
{
   creat("/tmp/temp", 644) ; /* Błąd, mialo byc 0644 */
}
```

```
~/Zajecia/Unix/C> ./creat1-bad
~/Zajecia/Unix/C> ls -l /tmp/temp
--w----r-T   1 ben      zls             0 Nov  1 16:45 /tmp/temp
```

9.3 Jak Poprawnie używać `creat`

creat.c

```
#include <stdio.h>  /* perror */
#include <fcntl.h>  /* creat */
#include <unistd.h> /* close */

int main(int argc, char **argv)
{
  int fd;
  if (argc<2) return 1 ;

  fd=creat(argv[1], 0644) ;
  if(-1==fd) { perror("creat") ; return 2 ; }

  if(-1==close(fd)) { perror("close") ; return 3 ; }
  return 0 ;
}
```

9.4 Poprawne i błędne działanie `creat`

```
~/Zajecia/Unix/C> ./creat /tmp/temp
~/Zajecia/Unix/C> ls -l /tmp/temp
-rw-r--r--   1 ben      zls             0 Nov  1 16:48 /tmp/temp
```

```
~/Zajecia/Unix/C> ./creat .
creat: Is a directory

~/Zajecia/Unix/C> ./creat /etc/temp
creat: Permission denied

~/Zajecia/Unix/C> ./creat ./creat
creat: Text file busy
```

unlink.c

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

int main(int argc, char **argv)
{
  if (argc<2) return 1 ;

  if(-1==unlink(argv[1])) { perror("unlink") ; return 2 ; }

  return 0 ;
}
```

9.5 Funkcja open

```c
#include <fcntl.h>
int open( char *path, int flags);
int open( char *path, int flags, mode_t mode);
int creat( char *path, mode_t mode);
```

Funkcja `open` otwiera plik o podanej nazwie i dostarcza deskryptor (używany w późniejszych operacjach)

Parametr `flags` specyfikuje żadany tryb dostępu do pliku — może przyjąć jędną z wartości `O_RDONLY, O_WRONLY, O_RDWR`, ewentualnie połączoną bitowo (operatorem `|`) z następującymi flagami:

- `O_CREAT` — plik zostanie stworzony jeśli nie istnieje.
- `O_EXCL` — użyte z `O_CREAT` powoduje błąd jeśli plik istnieje
- `O_TRUNC` — plik zostanie zamazany (skrócony do zera) jeśli już istnieje
- `O_APPEND` — wskaźnik pozycji zostanie umieszczony na końcu pliku

- O_NDELAY — wszystkie operacje na pliku będą wykonywane "bez czekania"

Parametr mode mówi jakie prawa dostępu mają być nadane tworzonemu plikowi.

creat jest równoważne open z flagami O_CREAT|O_WRONLY|O_TRUNC

open i creat, podobnie jak większość funkcji systemowych zwraca -1 w razie błędu. Powinniśmy zawsze sprawdzać czy to nie nastąpiło, np

```
if ((from = open(argv[1], O_RDONLY)) < 0) {
    perror(argv[1]);
    exit(1);
}
```

9.6 Funkcje read i write

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t n);
ssize_t write(int fd, void *buf, size_t n);
```

Funkcja read czyta z pliku reprezentowanego przez fd do n bajtów, umieszczając je w buforze wskazywanym przez buf. Daje w wyniku ilość odczytanych bajtów. Jeśli liczba ta jest mniejsza od n, oznacza to z reguły, że dotartliśmy do końca pliku.

Funkcja write pisze do pliku reprezentowanego przez fd do n bajtów z bufora wskazywanego przez buf. Daje w wyniku ilość zapisanych bajtów. Jeśli liczba ta jest mniejsza od n, oznacza to **błąd**.

W obu wypadkach fd musi być deskryptorem otrzymanym od funkcji open (lub pokrewnej) albo jednym ze standardowych deskryptorów: 0,1,2

9.7 Funkcja stat i jej krewniacy

```
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Powyższe funkcje dostarczają informacji o podanym pliku. stat czyni to dla pliku o podanej nazwie (rozwijając dowiazania symboliczne), zaś fstat — dla reprezentowanego przez deskryptor.

Funkcja lstat działa jak stat z tym, że jeśli podany plik jest dowiązaniem symbolicznym, to zwraca informacje o tym dowiązaniu a nie o wskazywanym przez nie pliku.

Wszystkie trzy umieszczają informacje w strukturze stat

9.8 Struktura stat

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

struct stat
{
 dev_t         st_dev;      /* device */
 ino_t         st_ino;      /* inode */
 umode_t       st_mode;
 nlink_t       st_nlink;    /* link count */
 uid_t         st_uid;
 gid_t         st_gid;
 dev_t         st_rdev;    /* device type */
 off_t         st_size;    /* size in bytes */
 unsigned long st_blksize; /* blocksize */
 unsigned long st_blocks;  /* size in blocks */
 time_t        st_atime;
 time_t        st_mtime;
 time_t        st_ctime;
};
```

9.9 Program times

times.c

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

int main(int argc, char **argv)
{
   struct stat info ;

   if(argc < 2) return 1 ;
   if (stat(argv[1], &info) < 0)
     { perror(argv[1]) ; return 2 ; }
   printf("Last accessed: %s",
                 ctime(& info.st_atime)) ;
   printf("Last modified: %s",
                 ctime(& info.st_mtime)) ;
   return 0 ;
}
```

```
~/Zajecia/Unix/C> ./times /etc/passwd
Last accessed: Sat Nov  1 19:05:09 1997
Last modified: Thu Oct 30 15:46:35 1997
```

9.10 Program io-append

io-append.c

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h> /* perror */

void main(int argc, char **argv)
{
    int n;
    int from, to;
    char buf[1024];
```

/* Sprawdź parametry wywołania programu. Ponieważ nie używamy `printf`, każdy kawałek komunikatu musimy wypisać oddzielnie */

```
if (argc != 3) {
    write(2, "Usage: ", 7);
    write(2, *argv, strlen(*argv));
    write(2, " from-file to-file\n", 19);
    exit(1);
}
```

/* Otwórz plik `from` do czytania. Sygnalizuj błąd jeśli się nie udało (np. plik nie istnieje). */

```
if ((from = open(argv[1], O_RDONLY)) < 0) {
    perror(argv[1]);
    exit(1);
}
```

/* Otwórz plik `to` do dopisywania. Plik zostanie utworzony z prawami dostępu 644 (`-rw-r-r-`) jeśli nie istnieje. */

```
if ((to = open(argv[2],
               O_WRONLY | O_CREAT | O_APPEND,
               0644)) < 0) {
    perror(argv[2]);
    exit(1);
}
```

/* Czytaj kolejne porcje aż do końca pliku `from` i zapisuj na plik `to`. Zawsze piszemy tyle ile przeczytaliśmy raczej a nie 1024 bajty */

```
while ((n = read(from, buf, sizeof(buf))) > 0)
    write(to, buf, n);
```

/* Zamknij pliki. */

```
close(from);
close(to);
exit(0);
}
```

9.11 Program `io-who`

io-who.c

```
#include <stdio.h>
#include <utmp.h>
#include <fcntl.h>
#include <unistd.h>
void print_utmp_entry(struct utmp * u) /*...*/

int main()
{
int fd ;
struct utmp u ;

if ((fd = open(UTMP_FILE, O_RDONLY)) == -1) {
  perror(UTMP_FILE) ;  return 1 ;
}
while (read(fd, &u, sizeof(u)) == sizeof(u)) {
  if( ( u.ut_type != USER_PROCESS)
       || !u.ut_name[0] )
    continue ;
  print_utmp_entry(& u) ;
}
close(fd) ;  return 0 ;
}
```

Rozdział 10

Zaawansowane funkcje wejścia/wyjścia

10.1 Pliki specjalne

Oprócz "zwykłych" plików, w systemi Unix są jeszcze tzw. pliki specjalne, takie jak:

- katalogi
- urządzenia
- łącza nazwane

Generalnie można je otwierać funkcją `open`, czytać i pisać przy pomocy funkcji `read` i `write` (choć pliki specjalne mają tu pewną specyfikę). Nie mozna ich natomiast tworzyć przy pomocy `creat` (służy do tego osobna funkcja — `mknod`

10.2 Czytanie katalogów

Katalog najlepiej czytać przy pomocy funkcji `readdir`, po otwarciu go przez `opendir`.

```
  #include <sys/types.h>
  #include <dirent.h>

 DIR *opendir(const char *name);
 struct dirent *readdir(DIR *dir);
 int closedir(DIR *dir);
```

Struktura `dirent` ma w zasadzie tylko jedno interesujące pole: `d_name`. Resztę informacji najlepiej uzyskać za pomocą `stat`.

10.3 Przykład

readdir.c

```
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>
#include <sys/stat.h>
int main()
{
    DIR *dp;
    struct dirent *dir;
    struct stat sbuf ;

    if ((dp = opendir(".")) == NULL) {
        fprintf(stderr, "cannot open directory.\n");
        return 1;
    }


    while ((dir = readdir(dp)) != NULL) {
        stat(dir->d_name, &sbuf) ;
        printf("%07o %20s %ld\n",
               sbuf.st_mode, dir->d_name, sbuf.st_size);
    }

    closedir(dp);
    return 0 ;
}
```

```
[12:31:38] ben@kawa:~/Zajecia/Unix/C> ./readdir | head
0040755                    . 3072
0040755                   .. 1024
0100644               true.c 56
0100644                no2.c 85
0100644              args1.c 222
0100644             Makefile 856
```

10.4 Sprytniejszy dostęp do katalogu: `scandir`

```
int scandir (const char *dir, struct dirent ***namelist,
             int (*selector) (struct dirent*),
             int (*cmp) (const void *, const void *))
```

Funkcja ta przegląda zawartość katalogu `dir`, wybierając tylko pozycje zaakceptowane przez `selector`, wynik jest posortowany według porządku zadanego przez funkcję `cmp`

Jeśli interesuje nas porządek alfabetyczny, można użyć funkcji `alphasort`:

```
int alphasort (const void *a, const void *b)
```

scandir.c

```
static int one (struct dirent *unused){ return 1;}

int main (void)
{
  struct dirent **eps;
  int n;
  n = scandir ("./", &eps, one, alphasort);
  if (n >= 0) {
      int cnt;
      for (cnt = 0; cnt < n; ++cnt)
        puts (eps[cnt]->d_name);
    }
  else
    perror ("Couldn't open the directory");
  return 0;
}
```

10.5 Funkcja `select`

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int  select(int  n,  fd_set  *readfds,  fd_set  *writefds,
fd_set *exceptfds, struct timeval *timeout);

       FD_CLR(int fd, fd_set *set);
       FD_ISSET(int fd, fd_set *set);
       FD_SET(int fd, fd_set *set);
       FD_ZERO(fd_set *set);
```

Funkcja `select` obserwuje podane deskryptory i wraca, gdy któryś z nich zmieni stan (lub upłynie podany czas).

10.6 Przykład użycia `select`

select1.c

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    fd_set rfds;
    struct timeval tv;
    int retval;

    /* Obserwuj stdin (fd 0) */
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);

    /* Czekaj najwyżej 5 sekund */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfds, NULL, NULL, &tv);

    if (retval)
        printf("Coś jest na wejściu!\n");
        /* FD_ISSET(0, &rfds) > 0 */
    else
        printf("Przez 5 sekund nic nie było.\n");

    exit(0);
}
```

10.7 Komunikacja z terminalem

Ponieważ terminal jest urządzeniem, dostępnym za pośrednictwem pliku `/dev/tty`, teoretycznie można komunikować się z nim za pośrednictwem `read`, `write` i `ioctl`. Jest to jednak na tyle skomplikowane, że lepiej skorzystać z gotowych bibliotek, np. `ncurses`. W tym miejscu omówimy tylko dwa problemy, które często stają przed piszącymi programy pod Unixem:

- Jak wczytywać z terminala bezporednio naciśnięte klawisze?
- Jak wczytywać bez echa (np. hasło)?

10.8 Jak wczytywać z terminala bezporednio naciśnięte klawisze?

- Po pierwsze: warto się przekonać, że deskryptor z którego chcemy czytać (czyli zwykle standardowe wejście) jest faktycznie związany z terminalem. Można tego dokonać przy pomocy funkcji `isatty`:

  ```
  if( isatty(0) ) ...
  ```

- Po drugie: wejście z terminala jest buforowane. W tzw. kanonicznym trybie pracy terminala, zawartość bufora jest przekazywana po naciśnięciu `Enter`. Jeśli wyłączymy tryb kanoniczny, będzie ona przekazywana po wypełnieniu bufora, więc...

- ... musimy ustawić rozmiar bufora na 1

`keypress.c`

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

#include <termios.h>
#include <string.h>

static struct termios stored;

void set_keypress(void)
{
    struct termios new;

    tcgetattr(0,&stored);

    memcpy(&new,&stored,sizeof(struct termios));



    /* wyłącz tryb kanoniczny i ustaw rozmiar bufora na 1 */
    new.c_lflag &= (~ICANON);
    new.c_cc[VTIME] = 0;
    new.c_cc[VMIN] = 1;

    tcsetattr(0,TCSANOW,&new);
    return;
}
```

```
void reset_keypress(void)
{
    tcsetattr(0,TCSANOW,&stored);
    return;
}


void main()
{
  int c ;

  set_keypress() ;
  printf("Naciśnij coś: ") ;
  c = getchar() ;
  printf("\nNacisnąłeś: %c\n", c) ;
}
```

Korzystaliśmy tu z funkcji

```
int tcgetattr ( int fd, struct termios *tp );
int tcsetattr ( int fd, int actions, struct termios *tp );
```

10.9 Jak wczytywać bez echa (np. hasło)?

Można oczywiście wyłączyć echo przy pomocy `tcsetattr` (albo wręcz `ioctl`), ale dużo prościej użyć funkcji bibliotecznej `getpass`:

`getpass.c`

```
#include <stdio.h>

char *getpass( const char * prompt );

void main()
{
  printf("%s\n", getpass("Password: ")) ;
}
```

```
 ben@kawa:~/Zajecia/Unix/C> ./getpass
Password:
tajne
```

(Oczywiście w faktycznych zastosowaniach nie ma sensu używać tu funkcji `printf`).

Rozdział 11

Programowanie współbieżne — procesy

11.1 Środowisko

Z programu w C istnieją (co najmniej) trzy metody dostępu do zmiennych środowiska:

- Za pomocą trzeciego argumentu funkcji `main`:

  ```
  int main(int argc, char *argv[], char *envp[])
  ```

- Za pomocą zmiennej `environ`:

  ```
  extern char ** environ ;
  ```

- Za pomocą funkcji `getenv`:

  ```
  char *getenv(const char *name);
  ```

showenv1.c

```
#include <stdio.h>

void main(int argc, char *argv[], char *envp[])
{
  int i ;

  for (i=0; envp[i] != NULL ; i++)
    printf("%s\n", envp[i]) ;
}
```

showenv2.c

```
#include <stdio.h>
extern char **environ ;

void main()
{
  int i ;

  for (i=0; environ[i] != NULL ; i++)
    printf("%s\n", environ[i]) ;
}
```

showenv3.c

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

void main(int argc, char *argv[], char *envp[])
{
  int i ;

  if(argc<2) {
    for (i=0; envp[i] != NULL ; i++)
      printf("%s\n", envp[i]) ;
  }
  else {
    printf("%s\n", getenv(argv[1])) ;
  }
}
```

11.2 Identyfikatory procesów

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

showpid.c

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

void main()
{
   printf( "Mój pid  = %d\n", getpid());
   printf( "pid ojca = %d\n", getppid());
}
```

11.3 Funkcja `fork`

```
#include <unistd.h>
pid_t fork(void);
```

Wywołanie funkcji `fork` powoduje stworzenie kopii (potomka) procesu, różniącej się od niego tylko wartościami PID i PPID oraz wartością dostarczoną przez `fork`. Ojcu `fork` dostarcza PID syna, synowi zaś 0.

Jeśli stworzenie nowego procesu się z jakiegoś powodu (brak pamięci, za dużo aktywnych procesów) nie powiodło, `fork` dostarcza `-1`.

proc_fork.c

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

int main ()
{
  pid_t pid;

  printf("My process id = %d\n", getpid());

  switch (pid=fork()) {
    case -1:
      fprintf(stderr, "Error in fork\n");
      exit(1);
```

```
    case 0:
      printf("Syn: My process id = %d\n", getpid());
      printf("Syn: Value returned by fork() = %d\n", pid);
      exit(0);

    default:
      printf("Ojciec: My process id = %d\n", getpid());
      printf("Ojciec: Value returned by fork() = %d\n", pid);

      if (wait(0) == -1) {
        fprintf(stderr, "Error in wait\n");
        exit(1);
      }
      exit(0);
  } /*switch*/
}
```

11.4 Rodzina funkcji exec

```
#include <unistd.h>

extern char **environ;

int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ...,
            char * const envp[]);
int exect( const char *path, char *const argv[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

proc_exec.c

```
    case 0:
      printf("Syn: My process id = %d\n", getpid());
      printf("Syn: Value returned by fork() = %d\n", pid);

      execlp("ps", "ps", 0);
      fprintf (stderr, "Error in execlp\n");
      exit(1);

    default:
      printf("Ojciec: My process id = %d\n", getpid());
```

```
    printf("Ojciec: Value returned by fork() = %d\n", pid);
```

```
[13:56:47] ben@kawa:~/Zajecia/Unix/C> ./proc_exec
My process id = 2224
Ojciec: My process id = 2224
Ojciec: Value returned by fork() = 2225
Syn: My process id = 2225
Syn: Value returned by fork() = 0
  PID TTY STAT  TIME COMMAND
 2224  ?  S    0:00 ./proc_exec
 2225  ?  R    0:00 ps
13950  ?  S    0:01 -bash
```

11.5 Funkcja pipe

```
#include <unistd.h>
int pipe(int fd[2]);
```

Funkcja `pipe` tworzy i otwiera łącze i dostarcza deskryptorów do jego końców. Deskryptor `fd[0]` służy do czytania, zaś `fd[1]` — do pisania.

Łącza o których tu mowa to dokładnie ten mechanizm, który wykorzystywany jest przez interpreter poleceń do zrealizowania potoków, np. przy wykonywaniu

```
ls | sort
```

interpreter poleceń tworzy dwóch synów (`fork`), tworzy między nimi łącze (`pipe`) i każe im wykonywać odpowiednie programy (`exec`).

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

/* Read characters from the pipe and echo them to stdout. */

void
read_from_pipe (int file)
{
  FILE *stream;
  int c;
  stream = fdopen (file, "r");
  while ((c = fgetc (stream)) != EOF)
```

```
    putchar (c);
  fclose (stream);
}

/* Write some random text to the pipe. */

void
write_to_pipe (int file)
{
  FILE *stream;
  stream = fdopen (file, "w");
  fprintf (stream, "hello, world!\n");
  fprintf (stream, "goodbye, world!\n");
  fclose (stream);
}


int
main (void)
{
  pid_t pid;
  int mypipe[2];

  /* Create the pipe. */
  if (pipe (mypipe))
    {
      fprintf (stderr, "Pipe failed.\n");
      return EXIT_FAILURE;
    }

  /* Create the child process. */
  pid = fork ();
  if (pid == (pid_t) 0)
    {
      /* This is the child process. */
      read_from_pipe (mypipe[0]);
      return EXIT_SUCCESS;
    }
  else if (pid < (pid_t) 0)
    {
      /* The fork failed. */
      fprintf (stderr, "Fork failed.\n");
      return EXIT_FAILURE;
    }
```

```
  else
    {
      /* This is the parent process. */
      write_to_pipe (mypipe[1]);
      return EXIT_SUCCESS;
    }
}
```

parent_pipe.c

```
char message[] = "Hello from your parent!";

void main()
{
  int pipe_fd[2];
  char pipe_read_fd_str[10];

  if (pipe (pipe_fd) == -1) {
    perror("pipe");
    exit(1);
  }

  switch (fork()) {
    case -1:  perror("fork");
              exit (1);


    case 0:  /* child */
      if (close (pipe_fd [1]) == -1) {
        perror("close (pipe_fd [1])");
        exit (1);
      }

      sprintf(pipe_read_fd_str, "%d", pipe_fd[0]);
      execl("./child_pipe", "child_pipe", pipe_read_fd_str, 0);
      perror("execl");
      exit(1);

    default: /* parent */
      if (close (pipe_fd [0]) == -1) {
        perror ("close (pipe_fd [0])");
        exit (1);
      }
```

```
      if (write (pipe_fd[1], message, sizeof(message)) == -1) {
        perror ("write");
        exit (1);
      }

      if (wait (0) == -1) {
        perror ("wait");
        exit (1);
      }

      exit (0);
  } /* switch (fork ()) */
}
```

```
[13:56:48] ben@kawa:~/Zajecia/Unix/C> ./parent_pipe
Reading data from file descriptor 3
Read 24 byte(s): "Hello from your parent!"
```

child_pipe.c

```
#define BUF_SIZE                1024
void main (int argc, char *argv[])
{
  int read_fd;
  char buf [BUF_SIZE];
  int buf_len;

  if (argc != 2) { ...}

  read_fd = atoi(argv[1]);
  printf ("Reading data from file descriptor %d\n", read_fd);

  buf_len = read (read_fd, buf, BUF_SIZE - 1);
  buf [BUF_SIZE - 1] = '\0';



  switch (buf_len) {
    case -1: /* Error in read */
      perror ("read");
      exit (1);

    case 0:  /* Unexpected end-of-file */
      fprintf (stderr, "read: unexpected end-of-file\n");
```

```
      exit (1);

    default: /* Successful read. */
      printf ("Read %d byte(s): \"%s\"\n", buf_len, buf);
      exit (0);
  }
}
```

Rozdział 12

Sygnały

12.1 Wysyłanie sygnałów

Sygnały są prostą metodą komunikacji między procesami. Używane są do komunikowania sytuacji wyjątkowych (jak np. błędna instrukcja, dzielenie przez zero, śmierć potomka) lub prostych żadań (np. zakończ działanie, wczytaj ponownie pliki konfigurację).

```
int kill(pid_t pid, int sig);
```

Sygnały mogą być wysyłane z poziomu shella poleceniem `kill` lub z programu przy pomocy funkcji o tej samej nazwie. W obu wypadkach należy podać numer sygnału oraz identyfikator procesu do którego chcemy go wysłać.

Standardową reakcją na większość sygnałów jest przerwanie działania (czasem połączone ze zrzutem obrazu pamięci. Ważnym wyjątkiem jest `SIGCHLD` (śmierć potomka), który doimyślnie jest ignorowany.

12.2 Obsługa sygnałów — funkcja `signal`

Proces może zmienić sposób reakcji na sygnał przy pomocy funkcji `signal`:

```
#include <signal.h>
void (*signal(int signum, void (*handler)(int)))(int);
```

Parametr `handler` może być **wskaźnikiem** do funkcji obsługi sygnału lub jedną z dwu predefiniowanych wartości

`SIG_IGN` — zignoruj sygnał

`SIG_DFL` — przywróć domyślny sposób obsługi sygnału.

Nie można przechwycić sygnału `SIGKILL` — zawsze powoduje on natychmiastowe zakończenie daiałania procesu.

12.3 Co można zrobić z sygnałem po jego otrzymaniu?

(a) Zignorować

```
main()
{
  signal(SIGINT, SIG_IGN);
  signal(SIGQUIT, SIG_IGN);
  /* itd */
}
```

Należy przy tym pamiętać, że po nadejściu sygnału przywracana jest standardowa reakcja, dlatego jeśli chcemy trwale ignorować sygnał, należy napisać własną funkcję.

(b) Posprzatać i zakończyć działanie

```
int moje_dzieci;
void porzadki(int typ_syg)
{
  unlink("/tmp/plik_rob");
  kill(moje_dzieci, SIGTERM);  wait(0);
  fprintf(stderr, "program konczy dzialanie ...\n");
  exit(1);
}

main()
{
  signal(SIGINT, porzadki);
  open("/tmp/plik_rob", O_RDWR | O_CREAT, 0644);
  moje_dzieci = fork();
  /* itd */
}
```

(c) Dokonać dynamicznej rekonfiguracji

```
void czytaj_plik_konf (int typ_syg)
{
  int fd;
  fd = open("moj_plik_konf", O_RDONLY);
  /* czytanie parametrów konfiguracyjnych */
  close(fd);
  signal(SIGHUP, czytaj_plik_konf);
}
```

```
main()
{
  czytaj_plik_konf();  /* konfiguracja początkowa*/

  while(1) {   /* obsługa w petli */
  ...
  }
}
```

(d) Przekaż raport lub dokonaj zrzutu wewnetrznych tablic

```
int licznik;
void drukuj_info(int typ_syg)
{
  /* drukuj info o stanie */
  printf("liczba skopiowanych blokow: %d\n", licznik);
  signal(SIGUSR1, drukuj_info);
}

main ()
{
  signal(SIGUSR1, drukuj_info);
  for (licznik=0; licznik<DUZA_LICZBA; licznik++) {
    /* czytaj blok z tasmy wejsciowej */
    /* pisz blok na tasme wyjsciowa */
}
}
```

(e) Włącz/wyłącz śledzenie

```
int flaga;
void przelacz_flage(int typ_syg)
{
  flaga ^= 1;
  signal(SIGUSR1, przelacz_flage);
}

main()
{
  /* inicjalnie wylacz sledzenie */
  flaga = 0;
  signal(SIGUSR1, przelacz_flage);
  /* wewnatrz kodu instrukcje implementujace sledzenie
     powinny wygladac nastepujaco: */
```

```
  if (flaga) printf("cos uzytecznego\n");
}
```

12.4 Grupy procesów

Każdy proces posiada identyfikator grupy procesów, do której należy. Początkowo jest on dziedziczony od ojca:

```
[10:28:01] ben@kawa:~/Zajecia/Unix/C> ./proc_pg
Father process. My process id = 3230
Father process. My process group id = 3230
Child process: My process id = 3231
Child process: My process group id = 3230
```

Identyfikator ten można odczytać funkcją `getpgrp`. Proces może też odłączyć się od grupy (tworząc własną) przy pomocy funkcji `setpgrp`:

```
int setpgrp(void);
pid_t getpgrp(void);
```

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

int main ()
{
  switch (fork()) {
    case -1:
      fprintf(stderr, "Error in fork\n");
      exit(1);

    case 0:
      printf("Child process: My process id = %d\n", getpid());
      printf("Child process: My pgid = %d\n", getpgrp());
      exit(0);

    default:
      printf("Father process. My process id = %d\n", getpid());
      printf("Father process. My pgid = %d\n", getpgrp());

      if (wait(0) == -1) {
```

```
      fprintf(stderr, "Error in wait\n");
      exit(1);
    }
    exit(0);
  } /*switch*/
}
```

12.5 Wysyłanie sygnału do grupy procesów

Wysłanie sygnału do grupy procesów odbywa się za pomocą funkcji `killpg`:

```
int killpg(int pgrp, int sig);
```

Wywołanie z parametrem `pgrp` równym 0 powoduje wysłanie sygnału do własnej grupy.

Poniższy program tworzy własną grupę procesów, a potem 10 potomków. Z tych potomków pięciu (o nieparzystych numerach) tworzy własne grupy. Po pięciu sekundach proces macierzyty wysyła sygnał `SIGINT` do wszystkich członków swojej grupy. Zabici zostaja Ci, ktorzy nie utworzyli własnych grup. Pozostali po 15 sekundach kończą pracę.

`sigproc.c`

```
void main ()
{
  register int i;

  if (setpgrp() == -1)  /*ustalenie nowej grupy*/
    syserr("setpgrp");

  for (i = 0; i < 10; i++)
    switch (fork()) {
      case -1:  syserr("fork");
      case 0: /*proces potomny*/
        if (i & 1)
          if (setpgrp() == -1)
            syserr("setpgrp2");
        printf("pid = %d pgrp = %d\n", getpid(), getpgrp());


        if (i & 1)
          sleep(15);
        else
          pause();
```

```
        exit(0);
    }

  sleep(5); /*dajmy czas procesom potomnym na rozpoczecie dzialania*/
  if (kill(0, SIGINT) == -1)
    syserr("kill");
  exit(0);
}  /*main*/
```

```
[10:54:12] ben@kawa:~/Zajecia/Unix/C/07_signal> ./sigproc
pid = 4420 pgrp = 4419
pid = 4421 pgrp = 4421
pid = 4422 pgrp = 4419
...
```

12.6 Sygnały POSIX

Standard POSIX wprowadził bardziej elastyczny sposób obsługi sygnałów:

```
int sigaction(int signum,  const  struct  sigaction  *act,
struct sigaction *oldact);
```

Funkcja `sigaction` ustala nowy sposób obsługi sygnału, opisywany przez strukturę `sigaction`:

```
struct sigaction {
              void (*sa_handler)(int);
              sigset_t sa_mask;
              int sa_flags;
              void (*sa_restorer)(void);
       }
```

Pole `sa_handler` to wskaźnik do funkcji obsługi sygnału (analogicznie jak w `signal`).

Pole `sa_mask` mówi, jakie sygnały powinny być zablokowane na czas obsługi (zawsze blokowany jest sygnał właśnie obsługiwany)

Pole `sa_flags` jest bitową alternatywą flag, z których ważniejsze to:

`SA_RESETHAND` — przywróć domyślny sposób obsługi po użyciu (czyli zachowuj się jak `signal`)

`SA_NOMASK` — **nie** blokuj sygnału na czas jego obsługi

`SA_RESTART` — jeśli sygnał przerwał funkcję systemową, wznów ją po zakończeniu obsługi sygnału. Podobny efekt można uzyskać przy pomocy funkcji

```
int siginterrupt(int sig, int flag);
```

12.7 Operacje na zbiorach sygnałów

```
int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signum);

int sigdelset(sigset_t *set, int signum);

int sigismember(const sigset_t *set, int signum);
```

```
int  sigprocmask(int  how,  const  sigset_t *set, sigset_t
*oldset);
```

Program wypisuje informacje o wszystkich sygnałach i próbuje zmienić ich obsługę; jeśli taka zmiana się nie powiodła, użytkownik jest o tym informowany. Następnie może wysyłać do tego procesu pewne sygnały i oglądać reakcje programu na nie. Funkcja 'psignal' wypisuje stanadardowe komunikaty o sygnałach.

```
void a_handler (int signum)
{
  psignal (signum, "Oto jak wyglada komunikat o aktualnym sygnale");
}

void main ()
{
  int i;
  struct sigaction old, new;

  new.sa_handler = a_handler;
  sigemptyset(&new.sa_mask);


  new.sa_flags = 0;

  for (i = 1; i < NSIG; i++) {
    if (sigaction(i, NULL, &old) == -1)
      syserr("sigaction");

    printf("Sygnal %d ma obsluge %s i komunikat: %s\n", i,
```

```
          old.sa_handler == SIG_DFL ? "SIG_DFL" : "SIG_IGN", sys_siglist[i]);

    if (sigaction(i, &new, &old) == -1)
      printf( "Uwaga!: sygnal %d nie moze miec zmienionej obslugi\n", i);
  }
  while(getchar() != '\n');
  exit(0);
}
```

Poniższy program pokazuje jak niektóre interpretery poleceń mogą w trakcie czytania danych z terminala informować o zakończeniu procesu potomnego. Dzięki funkcji `siginterrupt` ustalamy, ze `SIGCHLD` ma nie przerywać działania funkcji systemowych. Po obsłużeniu sygnału (po 2 sekundach) nastepuje powrót do wykonywania funkcji `read`.

```
void catch_child (int sig)
{
  int status;
  pid_t childpid;

  if ((childpid = wait(&status)) == -1)
    syserr("wait");
  printf("Proces potomny %d zakonczyl sie ze kodem %d\n",
         childpid, status);
}


void main ()
{
  struct sigaction setup_action;
  sigset_t block_mask;
  char buf[100];
  int ile;

  sigemptyset (&block_mask);
  setup_action.sa_mask = block_mask;
  setup_action.sa_flags = 0;
  setup_action.sa_handler = catch_child;
  sigaction (SIGCHLD, &setup_action, NULL);

  if (siginterrupt(SIGCHLD, 0) == -1)
    syserr("siginterrupt");
```

```
  switch (fork()) {
    case -1:
      syserr("fork");
    case 0:
      sleep(2);
      exit(getpid());
    default:
      if ((ile = read(0, buf, sizeof(buf) - 1)) == -1)
        syserr("read");
      buf[ile] = '\0';
      printf("Odczytano: %s", buf);
      exit(0);
  }
}
```

Rozdział 13

Gniazda

Rozdział 14

Flex

14.1 Prościutki przykład

user.l

```
%option noyywrap
%%
username        printf("%s", getlogin()) ;
```

Jak to dziala:

```
ben@kawa:~/Zajecia/Unix/Flex>  echo 'Ach, username!' | ./user
Ach, ben!
```

14.2 Ogólna postać definicji skanera

definicje
%%
reguły (wzorce+akcje)
%%
funkcje pomocnicze (w C)

W sekcji definicji mogą się pojawić definicje klas znaków, np.

```
CYFRA      [0-9]
LITERA     [A-Za-z]
```

a także deklaracje opcji Flexa oraz używanych dalej zmiennych i funkcji (te ostatnie muszą być wcięte lub ujęte między znaczniki `%{` i `%}`.

14.3 Akcje złożone

data.l

```
%{
#include <time.h>
%}
%option noyywrap
%%
data    { long clock ; time( & clock) ; printf(ctime(&clock)) ; }
```

```
 ben@kawa:~/Zajecia/Unix/Flex> echo 'Today is data' | ./data
Today is Mon Nov 24 14:02:42 1997
```

14.4 Wzorce

`x` znak `x`

`.` dowolny znak (oprócz `\n`)

`[xyz]` dowolny znak spośród `x,y,z`

`[abj-o]` klasa znaków: `a,b`, od `j` do `o`

`[^A-Z]` dopełnienie klasy znaków

r`*` zero lub więcej r

r`+` jedno lub więcej r

r`?` zero lub jedno r

r`{2,5}` r od 2 do 5 razy

`{4}` r dokładnie 4 razy

rs konkatenacja r i s

r`|`s r lub s

r`/`s r w kontekście s

`^`r r na początku linii

r`$` r na końcu linii

14.5 Usuwanie zbędnych odstępów

noblanks.l

```
%%

[ \t]+                  putchar(' ') ;
[ \t]+$                 /* ignoruj */
\"[^"]*\"               ECHO ;
```

```
ben@kawa:~/Zajecia/Unix/Flex> ./noblanks
Ala      ma          kota.
Ala ma kota.
 As to "pies     Ali"
 As to "pies     Ali"
```

Aby usunąć spacje na początku linii należy dodać regułę dla `^[ \t]+`.

14.6 Liczenie linii i znaków

lc.l

```
  int linie = 0, znaki = 0 ;

%option noyywrap
%%
\n      ++linie ; ++ znaki ;
.       ++znaki ;
%%
void main()
{
        yylex() ;
        printf("Linii: %d, znakow: %d\n", linie, znaki) ;
}
```

14.7 Zawartość leksemu: zmienna yytext

```
%option noyywrap
        int suma=0 ;
%%
[0-9]+        suma += atoi(yytext) ;
suma            printf("%d\n", suma) ;
```

```
\n
.               /* nic */
```

```
 ben@kawa:~/Zajecia/Unix/Flex> ./dodaj
11
22
33
suma
66
```

14.8 Klasy znaków i wspólne akcje

```
%option noyywrap
CYFRA   [0-9]
        double suma=0 ;
%%
{CYFRA}+(\.{CYFRA}+)?  suma += atof(yytext) ;
^\n             |
suma            printf("%g\n", suma) ;
\n
.               /* nic */
```

Rozdział 15

Bison

15.1 Kalkulator dla RPN

rpn1.y

```
%{
#define YYSTYPE double
%}

%token NUM

%%

input:    /* empty */
        | input line
;




line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

     exp: NUM             { $$ = $1;         }
        | exp exp '+'     { $$ = $1 + $2;    }
        | exp exp '-'     { $$ = $1 - $2;    }
        | exp exp '*'     { $$ = $1 * $2;    }
        | exp exp '/'     { $$ = $1 / $2;    }
%%
```

```
#include <ctype.h>
int yylex ()
{
  int c;
  while ((c = getchar ()) == ' ' || c == '\t')  ;

  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  if (c == EOF)
    return 0;

  return c;
}

void yyerror (s)
     char *s;
{
  printf ("%s\n", s);
}

void main()
{
  yyparse() ;
}
```

15.2 Flex pomoże napisać skaner

rpnflex.l

```
%{
#include "rpn.tab.h"
%}
%%

[:digit:]+(\.[:digit:]+)?   { yylval=atof(yytext) ; return NUM ; }
[\+\-\*\/]                  return *yytext ;
\n                          return '\n'
.
```

15.3 Operatory infiksowe; priorytety i wiązanie

```
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation       */
%%
   ...
exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | '(' exp ')'        { $$ = $2;         }
;
```

makefile

```
LEX=/usr/bin/flex
YACC=/usr/bin/bison
%.tab.c: %.y
	$(YACC) -d $<

noblanks: noblanks.o
	cc -o $@ $< -lfl

dodaj: dodaj.o
	cc -o $@ $< -lfl

dodaj1: dodaj1.o
	cc -o $@ $< -lfl

rpn: rpn.tab.o rpnmain.o rpnclex.o
	cc -o $@ $^ -lm

calc: calc.tab.o
	cc -o $@ $^ -lm

clean:
	-rm -f *.o *.tab.c lex.yy.c
```